Data Layout Optimization for Multi-Valued Containers in OpenCL

Robert Strzodka

Max Planck Institut Informatik
Saarbrücken, Germany

Abstract

Scientific data is mostly multi-valued, e.g., coordinates, velocities, moments or feature components, and it comes in large quantities. The data layout of such containers has an enormous impact on the achieved performance, however, layout optimization is very time-consuming and error-prone because container access syntax in standard programming languages is not sufficiently abstract. This means that changing the data layout of a container necessitates syntax changes in all parts of the code where the container is used. Object oriented languages allow to solve this problem by hiding the data layout behind a class interface, however, the additional coding effort is enormous in comparison to a simple structure. A clever coding pattern, previously presented by the author, significantly reduces the code overhead, however, it relies heavily on advanced C++ features, a language that is not supported on most accelerators. This paper develops a concise macro based solution that requires only support for structures and unions and can therefore be utilized in OpenCL, a widely supported programming language for parallel processors. This enables the development of high performance code without an a-priori commitment to a certain layout and includes the possibility to optimize it subsequently. This feature is used to identify the best data layouts for different processing patterns of multi-valued containers on a multi-GPU system.

Keywords: multi-valued, multi-component, data layout, array of structures, AoS, structure of arrays, SoA, array of structures of arrays, ASA, OpenCL, multi-GPU

Email address: strzodka@mpi-inf.mpg.de (Robert Strzodka)
1. Introduction

DRAM memory chips deliver data most efficiently in data bursts of certain granularity. So a one byte (8 bit) memory request fetches a chunk of many more consecutive bytes at once, typically 32B to 128B. Consequently, fetching data in smaller chunks wastes a lot of bandwidth. Multi-dimensional and multi-valued data containers lead to strided memory accesses in the one dimensional address space and cause exactly this problem. It is particularly severe for SIMD units when they need to collect data from multiple strided addresses for parallel processing. Data permutations improving the data layout are an active area of research [1, 2]. Intel Array Building Blocks [3] also perform dynamic layout transformations of user data for better parallel processing speed. Such software transformations can be costly, therefore, hardware solutions are investigated [4, 5].

Currently, achieving the highest performance requires from the programmer to choose the best suited data layout statically. The two major choices for multi-valued data are array of structures (AoS) and structure of arrays (SoA), see Listing 1. The usual language syntax and standard container types lead naturally to the AoS layout while SIMD units much prefer the SoA format. This is extensively discussed in technical manuals for various many-core devices, e.g., CPU [6], GPU [7] or the Cell processor [8]. The major choices of AoS and SoA can be further refined to form hybrid formats, e.g., arrays of structures of arrays [9] or structures of arrays of structures [10].

The main obstacle to practical data layout optimization is the different syntax of each layout. The container definition appears only once, so syntax differences therein are tolerable, however, the syntax for access to the container is also different, as the last line of code in Listing 1 shows, and this appears thousands of times in large projects. Thus, changing the layout is time consuming and error-prone.

A simple solution for the discrepant container access syntax might seem to define a macro \texttt{at(container, index, component)} that either evaluates to \texttt{container[index].component} if an AoS layout is desired or to the complementary \texttt{container.component[index]} for an SoA layout. However, this does not solve the problem as in practice the access to a multi-valued container often occurs in two stages, e.g., if we want to compute the norm of the container element at position 5, then for AoS layout we would write

\begin{verbatim}
Type computeNorm(Element* elm) {
    return
    abs(elm->comp0)+abs(elm->comp1)+abs(elm->comp2);
}

Type norm = computeNorm(&container[5]);
\end{verbatim}
which first computes the address of the element 5 inside the container and then passes it to a norm function which accesses the components in a second stage. The native SoA layout from Listing 1 does not admit the same two stage process, because the component needs to be selected first. So we would still end up with different syntax for AoS and SoA layout scattered throughout the code.

To obtain the same syntax for the above two stage access using C++, we would have to turn the container and the elements into classes, and the element class would define a member function for each component, allowing to access them as `elm.component()`:

```cpp
class Cpp_Container { ... };  
class Cpp_Element { ... };  
Type computeNorm(Cpp_Element* elm) { return abs(elm->comp0()) + abs(elm->comp1()) + abs(elm->comp2()); } 
Type norm = computeNorm(&container[5]);
```

This would require to change the access to components from `elm.component` to `elm.component()` everywhere in the code (not necessary in C#). Template specializations of `Cpp_Container` and `Cpp_Element` would hide the data layout and allow its selection at class definition. The main problem with this standard object oriented solution is the amount of code that it requires for such a simple multi-valued container as shown in Listing 1. A more compact C++ solution has been successfully employed on CUDA ca-
pable GPUs [11], however, it still relies on classes and templates and thus
is out of reach to most accelerators.

This paper constructs a solution in C using only simple language con-
structs that allow an implementation in OpenCL [12], an industry standard
for parallel programming of accelerators. Section 2 explains the underlying
ideas. Having a tool that can switch the data layout throughout the code
with a single parameter enables a detailed study on how sampling types,
parallelization strategies and data layouts affect the performance of multi-
valued containers on a multi-GPU system. This contribution forms the
second part of the paper, whereby the sampling and parallelization types
are discussed in Section 3, the study results in Section 4, and application
examples in Section 5.

2. Flexible Data Layout in OpenCL

This section introduces flexible containers in C/OpenCL which support
both AoS and SoA layouts.

2.1. Offset Calculation

We assume that our container consists of $M$ identical elements. Each
element is a structure with $N$ components of size $c_0, c_1, \ldots, c_{N-1}$. With these
variables we express the offset computation for AoS ($\text{container}[i].\text{comp}<j>$)
and SoA ($\text{container}.\text{comp}<j>[i]$) as functions of the element index $i$ and
component index $j$:

$$f_{\text{AoS}}(i, j) = \sum_{k=0}^{N-1} c_k \cdot i + \sum_{k=0}^{j-1} c_k , \quad f_{\text{SoA}}(i, j) = c_j \cdot i + \sum_{k=0}^{j-1} c_k \cdot M \cdot (1)$$

The offset added to the base address of the container delivers the location
of the $j$-th component in the $i$-th element.

The two stage index computation discussed in the introduction corre-
sponds to the two addends in the formulae. In case of AoS the offset com-
putation $\text{elm}=&\text{container}[i]$ corresponds to the first addend, and the sub-
sequent component access $\text{elm}->\text{comp}<j>$ corresponds to the second addend.
This separation of indices is possible because the first addend depends only
on $i$ and the second only on $j$. In case of SoA, this separation is not possible
because the first addend depends on both $i$ and $j$.

Let $C := \max_{k=0}^{N-1} c_k$ be the maximum size among the components. We
can change the data alignment such that the storage of each component is
padded to occupy memory size $C$. This means that the container consists of $M \cdot N$ components of size $C$. The offset formulae change to

$$g_{AoS}(i, j) = C \cdot N \cdot i + C \cdot j, \quad g_{SoA}(i, j) = C \cdot i + C \cdot M \cdot j.$$  \hspace{1cm} (2)

These formulae look familiar: $g_{AoS}$ is the offset computation for an $M \times N$ matrix with row-major layout and $g_{SoA}$ is the offset computation for column-major layout. In particular, $g_{SoA}$ now also uses separate indices in the two addends and thus allows the same two stage index computation as $g_{AoS}$. This solves the indexing problem theoretically, however, a practical implementation in C faces multiple challenges concerning the types that realize this indexing strategy, in particular, because we are not dealing with matrix types but structures and named components. Section Appendix A contains the involved technical details. Their study is not required for the understanding of the paper.

A clear disadvantage of the entire solution seems that each component occupies space $C$ rather than its actual size. In general this could lead to a significantly larger memory footprint. But we are concerned with the SIMD processing of scientific data, which mostly utilizes only two different memory sizes for variables: 32 bit for float and int and 64 bit for double and long; so assuming a common size $C$ is not a big problem. Should floats and doubles occur together in a structure then the floats can be grouped in pairs, such that they occupy the same space as a double. Only in the case of an odd number of floats the memory footprint would become slightly larger. Section 2.4 gives a concrete example.

2.2. Simple ASA Example

Listing 2 shows flexible ASA (array of structures of arrays) code for the same simple structure as in Listing 1. ASA describes the high level data organization, in contrast to a SAS (structure of arrays of structures) organization. These patterns do not say much about the implementation or functionality which depend strongly on the programming language and intended usage. The C++ data layout abstraction [11] also uses an ASA pattern but the realization and functionality are quite different because C++ offers much more powerful language features. The first ASA pattern was probably the fine-grained optimization of the AoS layout for SIMD processing [9], but this is a hard-coded layout solution.

Following Listing 2 let us discuss the usage pattern of the flexible containers in C, references to line number will be given as L<n>. The functionality is enabled by including the header ASA.h. The container definition (L3-L8)
/* Flexible container definition in C */
#include "ASA.h"
ASA_STRUCT_CONTAINER
( Container, CONT_SIZE, ASA_AOS, 3, /* or ASA_SOA */
  Type0, comp0, ,
  Type1, comp1, ,
  Type2, comp2, ,
)
typedef Container_value_type Element;
typedef Container_access_type ContElm;
Element single = {0, 1, 2};
Container container;
ASAat(Container,container,i).comp2 = 1; /*direct update*/
Element* pSE = &single;
ContElm* pCE = &ASAat(Container,container,i);
pCE->comp1 = pSE->comp1; /*two stage update*/

Listing 2: A flexible container definition with the array of structures of arrays (ASA) pattern in C. It requires almost the same amount of code as the native AoS or SoA definition in Listing 1, however, here we can change the layout with a single parameter replacing ASA_AOS with ASA_SOA. The second half of the listing shows how the different access syntax from Listing 1 (L11) is unified (L15). The abstraction supports both direct (L15) and two stage updates (L17-L18) of a container value.

is performed with a macro ASA_STRUCT_CONTAINER which in this example defines a container type Container with CONT_SIZE elements in AoS layout (parameter ASA_AOS), and each element consists of three components comp0, comp1, comp2. This is a type definition so all parameters must be known at compile time.

Apart from the container type Container, the macro defines two more types derived from this name: Container_value_type is a structure consisting of the specified components (L9), it can be used to allocate and initialize individual elements with the same components as in the container (L12); Container_access_type is the type of an element residing inside the container (L10), it must not be used for allocation but only for access to the container through a pointer (L17). These two types are equal if the AoS layout is selected, but different for the SoA layout. Therefore, a flexible code that admits both layouts must not assume their equality.

Access to the container is provided through the macro ASAat() to which
the container type, the container instance and the element index are passed (L15, L17). The macro evaluates to the $i$-th element of type `Container_access_type` and can either be used directly to access components in the container (L15) or the address of the indexed element is stored for later use (L17). The second option is advisable if multiple operations are performed on the same element, because then the index computation occurs only once; Sections 3.3 and 3.4 show practical examples. In the last line the same access syntax to a component is used for single and indexed elements although the internally required offset computation is very different.

In summary, the container definition is equally simple as a native AoS or SoA definition. The macro `ASAat(Container,container,i)` replaces the AoS index computation `container[i]`. The user must maintain the distinction between the value type of an individual element and the access type of an element inside the container. Then, the code will work with both layouts and changing the layout becomes a simple parameter change.

2.3. Hybrid Layouts

The macro based container definition presented in the previous section is a type definition. The resulting type `Container` can be used to form new types, e.g., we can create an array of such containers (`A{ASA}`) or a structure of different ASA containers (`S{ASA}`) or even use multiple ASA containers in the definition of another ASA container (`{ASA}{ASA}`). There is no limit to the nesting depth, however, for performance of memory access patterns the order of the A’s and S’s closest to the unit stride dimension matters most. This is similar to multi-dimensional arrays, for which swapping the Y and Z dimensions has little impact on performance if the X dimension is sufficiently large. In our scientific setting we look at large collections of data, so one of the A’s in `{ASA}` contains many thousands of elements and the nesting on the higher levels has little influence on performance. However, we will consider fine-granular grouping of components at the lowest level, i.e., `{ASA}S`, because this strongly affects memory access patterns.

An apparent disadvantage of the proposed solution is the requirement that the ASA container size is a compile time constant. This is simply a consequence of the type definition and applies equally to C array types, e.g., `SIZE` must be a compile-time constant in `typedef int IntArray[SIZE]`. A simple solution is to define a smaller container of certain granularity and allocate a dynamically defined number of them, e.g., if the input data size is in the range $[10^5, 10^8]$, then we would define an ASA container of size $10^4$ and dynamically allocate $10$ to $10^4$ of them depending on the current data size. In the worst case this would waste $10\%$ of memory, however, only for
the small data sizes where the loss does not matter much; for large data sizes the loss would be negligible. This solution corresponds to the aforementioned A\{ASA\} arrangement and since the granularity is large enough the chosen nesting on higher levels is of little importance.

2.4. Abstraction Overhead

In case of the AoS layout the macro \texttt{ASAat(Container,container,i)} performs the same index computation as the native \texttt{container[i]}, which corresponds to the first addend of \(f\textsubscript{AoS}\) in Eq. 1. So for AoS there is no difference to the hand-coded solution. For the SoA layout the macro evaluates the first addend of \(g\textsubscript{SoA}\) in Eq. 2 and the subsequent component access \texttt{elm->comp<j>} corresponds to the second addend. In native SoA code \texttt{container.comp<j>[i]} this index computation happens in the reversed order, because the component is selected first. However, changing the evaluation order does not change the performance. So for both layouts we obtain the same performance as for the native code and there is zero overhead in the data access of the abstraction.

The container definition creates a performance difference for the SoA layout if the component sizes are different and their grouping leaves a rest, e.g., one double and three floats would form three 64 bit large groups in ASA, namely: one double, one pair of floats, and a single float with padding. This is 32 bit or 32/160=20\% larger than the native SoA container and stresses the bandwidth by the same factor. If performance depends solely on bandwidth then this is also the performance reduction in practice.

On the other hand, the indexing of multiple components with the abstraction is more efficient than the native SoA indexing. The following code uses the type definitions from Listing 1 and 2.

```c
/* ASA */
Container container;
ContElm* elm = &ASAat(
    Container,container,5);
elm->comp0 = 1;
elm->comp1 = 0;
elm->comp2 = 1;

/* SoA */
SoA_Container container;
container.comp0[5]= 1;
container.comp1[5]= 0;
container.comp2[5]= 1;
```

While the native SoA code evaluates \(f\textsubscript{SoA}\) in Eq. 1 three times before adding them to the base address (overall 3 MULs and 6 ADDs), the ASA code computes the first addend of \(g\textsubscript{SoA}\) in Eq. 2 only ones and afterwards adds only constants to the element address (overall 1 MUL and 4 ADDs). In both cases the second addend does not require a multiplication because it
is a compile-time constant. This more efficient indexing is of particular advantage in computation bound algorithms.

3. Sampling and Parallelization Types

This section discusses different sampling and parallelization types applied to multi-valued containers, including code examples in OpenCL. All readers familiar with C should be able to understand them, because the only differences to C are some variable and function attributes preceded by ‘_’.

3.1. Sampling Irregularity

In the introduction we have discussed the negative effects of strided memory accesses mainly with respect to inefficient memory bandwidth and SIMD utilization, e.g., parallel access to the first component comp0 in sixteen elements of the container AoS_CONTAINER from Listing 1 would occur at only one third of the peak bandwidth, because the intermediate components comp1 and comp2 would also be fetched from DRAM. Clearly, if the components comp1 and comp2 are cached and processed subsequently then the bandwidth has been utilized efficiently, however, SIMD processing of the sixteen comp0 components may not occur at full speed if the access to on-chip memory has a stride larger than one.

In case of the SoA_CONTAINER from Listing 1 parallel access to comp0 in sixteen consecutive elements suits SIMD processing well, because the elements are laid out consecutively in memory. However, access to a different component, for example comp1, occurs with a large stride in memory. This is not a problem if again multiple consecutive comp1 are fetched, such that there are two data streams one with comp0 and the other with comp1 components. However, the number of consecutively requested components depends on the application. If we draw a random sample of elements from a container then at each address only one component is read and the large stride between comp0 and comp1 leads to a memory latency overhead in addition to poor bandwidth utilization. An AoS layout has also a latency problem in random sampling of elements, however, the latency occurs only once for each element because its components are consecutive.

So the decision whether an AoS or an SoA layout is better depends strongly on the sampling type that the application uses. The strong advocation of SoA layouts in SIMD manuals for many-core devices [6, 7, 8] is based on the assumption that the application utilizes linear access to elements in the container. However, in practice this is often not the case and
sampling types of various irregularity occur ranging from linear over irregular to completely random. In the following we construct a versatile test scenario which produces a large number of different sampling types.

For this purpose we first define the periodic distance on an index set $I$ with $|I|$ indices as though the ends of the index set were connected.

\[\forall i, j \in I : \text{dist}(x, y) := \min(|x - y|, |I| - |x - y|).\]

Let $M$ be the number of elements in our container and $I_M := \{0, \ldots, M - 1\}$ the corresponding index set. We define the sampling of diameter $d$ as a random permutation $F_d : I_M \to I_M$ that fulfills

\[\forall i \in I_M : \text{dist}(F_d(i), i) \leq d/2.\]

For $d = 1$ we obtain a linear map ($F_1(i) = i$) and with growing $d$ the sampling becomes more irregular, because the average distance between two consecutive indices $\text{dist}(F_d(i), F_d(i + 1))$ becomes larger, until we reach a completely random permutation for $d = M$.

### 3.2. Sampling Sparsity

Besides the regularity of the sampling we also want to vary its sparsity, because in practice the number of sample values is often smaller than the entire set from which they are drawn. So let $s \geq 1$ be the sparsity factor, $S := M/s$ the number of sample values and $I_S := \{0, \ldots, S - 1\}$ the corresponding index set. We define our sampling of diameter $d$ with sparsity $s$ to be an injective map $F^s_d : I_S \to I_M$ that fulfills

\[\forall i \in I_S : \text{dist}(F^s_d(i), s \cdot i) \leq d/2.\] (3)

For $d = 1$ we have $F^s_d(i) = s \cdot i$ so the sparsity factor corresponds to a stride $s$ in the element access. However, this stride between elements is different from the stride between components, which was discussed in the first paragraph of Section 3.1.

We require an injective map because in our analysis we want to exclude the effects of fine grained synchronization, which is required when multiple parallel threads write to the same index. We already have multiple parameters for data layout, sampling, and parallelization strategies, so to point out their relations among each other we need to exclude the perturbing effects of other parameters.
3.3. General Multi-Valued axpy

As scientific data requires large containers that do not fit in on-chip memory we want an operation that fetches data from DRAM memory without on-chip data reuse. For this purpose we generalize the saxpy (\( Y = \alpha \times X + Y \)) operation to a general axpy (gaxpy) which operates on structures rather than scalars. The multiplication and addition are performed on each structure component independently. The gaxpy operation is applied to all container elements selected by the sampling map \( F^s_d \). The computation on each element is also independent.

In summary, this is an embarrassingly data parallel operation with no need for synchronization and no on-chip data reuse. Its performance is determined solely by the memory access patterns. By measuring the execution time this fact allows us to study which combinations of data layout, sampling and parallelization types create the best memory access patterns.

Listing 3 shows an OpenCL kernel that implements the gaxpy operation for the simple flexible ASA container from Listing 2. The kernel is executed by the OpenCL runtime for all sample values \( is=0..S-1 \). Each kernel execution is processed by one so called work-item; one can think of it as a very light-weight thread. Work-items in OpenCL execute independently of each other and in parallel to the extent that the hardware supports it. The built-in function \( \text{get\_global\_id(0)} \) returns the unique global ID of the current work-item and we use it as the current sample value that this work-item has to process (L5).

Listing 3: OpenCL kernel of a vertically parallelized gaxpy operation.

```c
__kernel void gaxpy_V(int S, __global int* F, Element alpha, __global Container* X, __global Container* Y )
{
    typedef __global Container_access_type ContElm;
    int is = get_global_id(0); /* ID of sample value */
    if( is < S ) {
        int i = F[is]; /* index of sample value */
        ContElm* elmX = &ASAat(__global Container, *X, i);
        ContElm* elmY = &ASAat(__global Container, *Y, i);
        elmY->comp0 += alpha.comp0 * elmX->comp0;
        elmY->comp1 += alpha.comp1 * elmX->comp1;
        elmY->comp2 += alpha.comp2 * elmX->comp2;
    }
}
```
__kernel void gaxpy_H( int S, __global int* F, Element alpha, __global Container* X, __global Container* Y )
{
    typedef __global Container_access_type ContElm;
    int is= get_global_id(0); /* ID of sample value */
    int ic= is%3; is= is-ic;
    for ( int g= 0; g < 3 && is < S; g++ , is ++) {
        int i= F[is]; /* index of sample value */
        ContElm* elmX = & ASAat ( __global Container , *X, i);
        ContElm* elmY = & ASAat ( __global Container , *Y, i);
        switch (ic) {
        case 0: elmY->comp0 += alpha.comp0*elmX->comp0; break;
        case 1: elmY->comp1 += alpha.comp1*elmX->comp1; break;
        case 2: elmY->comp2 += alpha.comp2*elmX->comp2; break;
        }
    }
}

Listing 4: OpenCL kernel of a horizontally parallelized gaxpy operation.

The sampling map F (L1) and the containers X, Y (L2) reside in the device DRAM memory (attribute __global), while the sample size S (L1) and the coefficient alpha (L2) are passed by value to the kernel and are stored on-chip. Each work-item performs one gaxpy operation independently of the others. It retrieves the index of the current sample value from the sampling map F (L7), uses it to compute the addresses of the elements in the containers (L8, L9) and finally performs the gaxpy operation on these elements (L10-L12). Depending on which layout is chosen for the container type Container (definition in Listing 2) this kernel produces completely different memory access patterns.

3.4. Vertical and Horizontal Parallelism

The kernel presented in the previous section utilizes so called vertical parallelism. It is the natural parallelization strategy when parallelization is exposed through kernel processing of independent work-items. Each work-item first operates on component comp0 then on comp1 and so on. Since multiple work-items execute in parallel, multiple components comp0 are requested and processed at the same time.

The alternative approach of horizontal parallelism lets each work-item operate on only one type of component. Parallel processing of multiple work-items then leads to the parallel processing of multiple components (as far
as the SIMD unit allows this), e.g., processing all saxpy operations on one element at the same time. Horizontal parallelism is more difficult to express in a kernel formulation because we must explicitly construct an assignment of work-items to components. Listing 4 implements the horizontally parallelized version of the same gaxpy operation as in the previous section. Although horizontal parallelism is alien to a kernel formulation and in literature only formulated with explicit vectorization, this listing is surprisingly not much longer than the standard vertical implementation in Listing 3.

The main difference between the two types of parallelism in a kernel formulation is that vertical parallelism naturally assigns work-items to elements, i.e., work-item is operates on the three components comp0, comp1, comp2 of the element is; whereas horizontal parallelism unnaturally assigns work-items to components, i.e., work-item is operates on component comp<is%3> of the three elements is-is%3, is-is%3+1, is-is%3+2. In Listing 4 the modulo 3 index computation (L6), the for-loop (L7) and the switch statement (L11) implement this assignment of work-items to components. Clearly, the code easily generalizes to higher number of components. The computation of the addresses in the ASA container (L9, L10) and the gaxpy operations themselves (L12-L14) are the same as in Listing 3.

From the SIMD perspective the switch statement is a performance killer. The three different cases are serialized leading to slower instruction processing and also three times more memory requests. With registers and many type casts one could coalesce the memory request in case of an AoS layout, however, the SIMD inefficiency would remain. A shorter solution is to specialize the kernel. The number of cases that are strictly required in the switch statement depends on the number of different component types in the container. Often there are only two different types, e.g., float and int, so it suffices to have one case for each of them. It is difficult for the compiler to perform this optimization automatically, but replacing the switch statement with an if-else construct does the trick. Figure 6 shows how the general switch version compares against the faster if-else version, which we use for all other tests.

Even after the specialization we still have a two-fold serialization and the overhead of the loop and branch instructions, so it appears impossible that this version is faster than the vertically parallelized kernel from the previous section. Nevertheless, the bandwidth study in Section 4 shows that the question which parallelization strategy is faster depends strongly on the data layout and sampling type. In fact, this version has a clear advantage for AoS layout in case of highly irregular sampling because parallel work-items maintain spatial locality among the components of the same element.
Table 1: Hardware configuration of the four GPU devices. The memory is distributed, each GPU having its own address space.

<table>
<thead>
<tr>
<th>GPU device</th>
<th>NVIDIA GTX295</th>
<th>Number of devices</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores (SMs) per GPU</td>
<td>30</td>
<td>Memory per GPU</td>
<td>896 MiB</td>
</tr>
<tr>
<td>Flops per core</td>
<td>8 MAD &amp; 8 MUL</td>
<td>Bus width per GPU</td>
<td>448 bit</td>
</tr>
<tr>
<td>Core frequency</td>
<td>1242 MHz</td>
<td>Memory frequency</td>
<td>2000 MHz</td>
</tr>
</tbody>
</table>

3.5. OpenCL Devices

Table 1 shows the configuration of our four GPU devices. Our gaxpy operation that we use for the bandwidth study is embarrassingly parallel and does not require any communication (Section 3.3), however, the fact that the memory of the GPUs is distributed has still an effect on the processing. First, the type of data distribution is different for the containers and the sampling map. The sampling map is generated on the CPU and only one quarter of it needs to be transferred to each GPU. The containers, however, cannot be split in this way because the sampling may access any part of the container, therefore, they must be duplicated for each GPU. The duplication has the effect of artificially increased sampling sparsity. Each GPU processes only one quarter of the sampling map, however, the indices may point to any element in the container, such that the ratio $M/S$ is bigger the more GPUs are used, cf. Section 3.2.

The software environment is a Linux 64-bit operating system with GNU g++ compiler 4.3.2 and CUDA 3.1.

4. Bandwidth Study

This section empirically explores the large parameter space created by the different data layout, sampling and parallelization types with the help of the gaxpy operation.

4.1. Parameter Space

For the performance measurement we use containers with one million elements, in which each element consists of ten floats and ten ints. On these containers we repeatedly perform the gaxpy operation from Section 3.3 parameterized by two data layouts (AoS, SoA), two parallelization strategies (vertical, horizontal), twenty one sampling diameters ($d \in \{2^0, ..., 2^{20}\}$) and seven sampling sparsities ($s \in \{2^0, ..., 2^6\}$). The computation time
Figure 1: Comparison of all four combinations from two data layouts (AoS, SoA) and two parallelization strategies (Vertical, Horizontal) on sampling permutations of increasing irregularity.

is measured on multiple iterations until all GPUs have finished. From the timings the effective bandwidth is computed and displayed in the figures.

For the gaxpy operation we study the pure impact of strided and irregular memory access on global bandwidth due to the layout and usage of multi-valued containers, so the data transfer is not included. In practice, one would never execute a single operation of this type on an accelerator because the low arithmetic intensity means that the data transfer takes longer than the computation. In Section 5 we look at applications and there the time for the entire process is considered.

4.2. Sampling Irregularity

Figure 1 answers the question what is the best combination of data layout and parallelization strategy in dependence on the sampling irregularity. The sparsity is fixed at $s = 1$, so each sampling map is a permutation. We see that in case of vertical parallelism (V-parallel) one should use an SoA layout (solid line) and in case of horizontal parallelism (H-parallel) one should use an AoS layout (solid line) because the other options (dashed lines) are clearly worse for all sampling types. However, whether the SoA V-parallel or the AoS H-parallel configuration is better depends on the irregularity of the sampling. For linear access the former is up to 7.4x faster whereas for random sampling it is up to 2.7x slower. So the layout should be chosen depending on the sampling type in the application.

Figure 2 answers the question if the results of the above comparison change for varying sparsity of the sampling. Qualitatively, the result is the same, for all tested sparsities: the SoA layout is better on regular sampling
Performance for Increasing Sampling Sparsity on four GTX295

SoA layout V-parallel
AoS layout H-parallel
AoS layout V-parallel
SoA layout H-parallel

Figure 3: Comparison of all four combinations from two data layouts (AoS, SoA) and two parallelization strategies (Vertical, Horizontal) on element indexing of increasing stride.

Performance for Increasing Sampling Sparsity on four GTX295

SoA-V linear
SoA-V irregular
SoA-V random
AoS-H linear
AoS-H irregular
AoS-H random

Figure 4: Impact of irregularity (linear $d = 1$, irregular $d = 2^{10}$, random $d = 2^{20}$) on the sparsity sampling graphs (sparsity $s \in \{2^0, \ldots, 2^6\}$) of AoS and SoA layouts with best parallelization.

and worse on highly irregular sampling, however, the quantitative advantage of the SoA layout decreases rapidly with growing sparsity. For small sparsity of $s = 4$ the advantage on regular sampling remains large (graphs marked by squares), for sparsity of $s = 16$ this advantage melts down to 1.3x on average (graphs marked by circles). To obtain very high bandwidth both the sparsity and the irregularity must be low for the SoA layout. The AoS layout is insensitive to both parameters, but slower on average.

4.3. Sampling Sparsity

Figure 3 determines the best combination of data layout and parallelization strategy in dependence on the sampling sparsity. The diameter is fixed here at $d = 1$ so the sampling sparsity is equivalent to a regular stride. The conclusion is similar to Figure 1 discussed in the previous section: V-parallelism should be paired with SoA layout and H-parallelism with AoS layout. In case of regular strides the SoA layout is always better than the AoS layout.

Figure 4 shows that the above conclusion is not true any more if we introduce sampling irregularity. For irregular sampling with $d = 2^{10}$ the SoA layout remains superior across all sparsities (graphs marked by squares) although by a much smaller margin. For random sampling the picture changes completely and the AoS layout is clearly superior for all sparsities (graphs marked by circles). We conclude that for the correct selection of the data layout the sampling irregularity in the application is of primary importance, the sampling sparsity changes the speedup factors but has little impact on the decision which layout is better.
4.4. Component Types

Grouping of components leads to a hybrid format (\{ASA\}S), see Section 2.3. The idea is to increase the granularity of memory accesses from 32 bit (float, int) to 64 bit (float2, int2) to 128 bit (float4, int4) per work-item. Figure 5 shows that the AoS layout with H-parallelism significantly benefits from this procedure which is solely due to a more efficient execution of the horizontally parallelized kernel (Section 3.4), because the layout of the container in memory does not change. For the SoA layout with V-parallelism the bandwidth improves more for higher sampling irregularity, as expected, because the larger components better preserve data locality. Notably, the cross-over point between the AoS and the SoA solution remains constant at $d = 10^4$.

Figure 6 demonstrates that the performance of the horizontally parallelized kernel is sensitive to the used component types. While the general version of the kernel is very slow, a specialization with two types performs much better and the one type specialization is ten times faster, because code serialization is eliminated.

4.5. Conclusions of Bandwidth Study

We summarize the observations of this study. The SoA layout should be paired with vertical parallelism, while the AoS layout should use horizontal parallelism. If the access pattern is linear or moderately irregular then the SoA layout is preferable, the AoS layout wins for highly irregular access patterns. The sparsity factor does not change the above recommendation, however, it reduces the performance advantage of SoA over AoS very
quickly, because it corresponds to a growing stride in the DRAM access. Grouping of components into larger types has a very positive impact on the horizontally parallelized kernel with AoS layout; the SoA layout with vertical parallelization also benefits in case of irregular access patterns. Because the horizontal parallelization in general collides with the SIMD processing pattern, it benefits greatly from specialized implementations targeted at the actual number of different component types in the structure.

The main reason for the observed performance differences and therewith derived recommendations has been already discussed in the introduction: DRAM chips always deliver data in large chunks, no matter how many bytes are actually requested. Therefore, a strided or irregular access to DRAM always reduces the bandwidth efficiency and irregularity causes latency problems irrespective of the hardware and software. However, this does not necessarily mean a reduction in overall application performance by the same factor, because other resources might be more limiting for the application. From a hardware perspective one can improve on latency and bandwidth efficiency despite an irregular access by multi-threading and on-chip data reuse, respectively. However, this does not help much for highly irregular access for which data reuse on-chip is unlikely and latency too high to hide. The following applications shine a little light on these mixtures of effects.

5. Impact on Applications

The bandwidth study in Section 4 has explored the parameter space of different data layouts, sampling and parallelization types in detail. In real-world application the parameter space is much larger and cannot be explored exhaustively. In this section two applications from data analysis exemplify some relevant and irrelevant additional effects.

5.1. Cross-Correlation of Data Series

Given two data series $A$ and $B$ we want to compute their cross-correlation

$$C[i] = (A \ast B)[i] := \sum_{j=-\infty}^{\infty} A[j] \cdot B[i + j], \quad (4)$$

where we assume that $A[i]$ and $B[i]$ are zero for all undefined $i$. This problem has a regular access pattern and high arithmetic intensity, therefore, an efficient implementation on the GPU proceeds by loading a block of $A$ and $B$ on-chip, forming partial sums in Eq. 4, and repeating this load and compute
process until a block of results has been obtained and written out into $C$. In case of four GPUs, each GPU computes one quarter of $C$ and the results are merged on the host. We measure the time for the entire process, including the transfer of $A$, $B$, $C$ and the post-processing on the host.

The code is parameterized such that we can change the layout and the parallelization of the global and the local data arrays independently. Overall, this gives $2^4 = 16$ different configurations and the native coding approach would require as many code variants of the entire project. With the ASA abstraction we need only one version of the host code, two OpenCL kernels implementing vertically and horizontally parallelized data copying to and from the chip, and two short compute functions implementing the partial summation on-chip with vertical or horizontal parallelism.

Figure 7 displays the effective bandwidth for a block size of 256, $A$ with one million elements and $B$ of varying size, each with four floats per element. For clarity only the four combinations with matching layout and parallelism are shown, the other combinations suffer from stride in global memory access or bank conflicts in local memory. For larger $B$ the computation to communication ratio is better, therefore, all variants execute faster. The two variants with vertically parallelized SoA layout (local SoA-V) perform around 1.4x better than the horizontally parallelized AoS layout (local AoS-H) because of the simpler indexing. This advantage decreases for smaller $B$ where communication plays a bigger role.
The 1.4x speedup factor is much smaller than the 7.4x observed between SoA-V and AoS-H in Section 4.2, since speed of local computation rather than global bandwidth dominates the performance in this case. Moreover, because local computation is dominating, the choice of global SoA-V vs. global AoS-H, i.e., the choice of data transfer to and from the chip, is less important; the simpler SoA-V produces only a small advantage.

The best configuration is SoA-V-SoA-V as expected from Section 4.5. The main new insight from this application is that in case of many local computations the choice of the global data layout is less relevant. So with the ASA abstraction we could easily switch to the slightly slower AoS-H-SoA-V configuration, if the data input is produced by code for which we do not have the freedom to modify its global data layout to SoA.

5.2. Statistical Inference with Bootstrapping

Given a sample $A$ we want to gain information about the sampling distribution of a statistic by computing its bootstrap distribution; in this example the desired statistic is the mean value. The required procedure is simple: we randomly resample $A$ $b$-times, creating $b$ samples each of the same size as $A$ (certain elements of $A$ will occur multiple times in the new samples), and compute the mean value on each of the $b$ samples. In this way we obtain $b$ mean values and can study their distribution, e.g., compute its center and standard deviation to infer the bias and standard deviation of the sampling distribution.

The task of the GPUs is to compute $b$ mean values from $b$ random resamples of $A$. A simple scheme is to have each work-item run a random number generator, we use a multiply-with-carry lag-1 32 bit generator [13], and compute one mean value by resampling $A$ with the generated random indices. In case of four GPUs, each GPU executes $b/4$ work-items to produce $b$ mean values overall. If the user defined number of resamples $b$ is small, such that $b/4$ is smaller than $W$, the minimum number of work-items to fill up our GPU; then $W$ work-items are executed forming only partial sums of the mean values, and the host finishes the mean value computation while merging the results from different GPUs into one output array. The entire time is measured including the transfer of $A$ and the partial sums, and the post-processing on the host.

The code parameterizes the layout of the input and output array and the parallelization strategy. In Figure 8 the effective bandwidth of the four main variants is shown. The size of $A$ varies from $2^{18}$ to $2^{25}$ elements and with it the number of resamples $b$ from $15 \cdot 2^{12}$ to $15 \cdot 2^5$ such that the overall
number of randomly drawn sample values $|A| \cdot b$ remains constant. Each element in $A$ consists of four floats.

In Figure 8 we see that the layout of the output array is irrelevant for the performance, as long as the parallelization suits with the input layout, i.e., SoA-SoA-V and SoA-AoS-V perform equally bad, AoS-AoS-H and AoS-SoA-H perform equally good. The average speedup of AoS-AoS-H over SoA-SoA-V is 3.8x reflecting the theoretically four times higher bandwidth efficiency. The absolute performance of both schemes improves for smaller $A$ and larger $b$, because no partial sums need to be formed and more work-items are launched on the GPU.

The AoS-AoS-H configuration is the best choice for this application as expected from Section 4.5. In contrast to the previous section the speedup is close to the theoretical value because other effects do not interfere. If desired, we can modify the output layout to SoA without penalty, however, a quick test with the ASA code reveals that a SoA input will reduce the performance dramatically. Such information is important for the integration of a bootstrapping procedure in a larger software project.

6. Conclusions

The paper presents the first solution for flexible multi-valued containers in C/OpenCL which support both AoS and SoA layouts. They can be easily used to create different hybrid formats. The flexible containers can be parallelized either in vertical or horizontal fashion and both solutions are shown in full generality as kernels without explicit vectorization. A versatile test scenario is constructed for the global bandwidth analysis of multi-valued containers on a multi-GPU system exploring the large parameter space created by the different data layout, sampling and parallelization types. Results identify the correct pairings of layout and parallelization, the quantitative effects of sampling sparsity and component types, and the primary influence of sampling irregularity on the best choice of data layout. The impact on real-world applications is exemplified in two cases as it depends on layout configurations of multiple containers and the presence of additional effects.

This paper realizes the flexible container solution in C/OpenCL for widest accelerator support and applicability to existing code. An integration of this functionality into higher level programming models for accelerators like HMPP Workbench [14], PyOpenCL [15] or Copperhead [16] would further raise programmer productivity on new projects dealing with multi-valued data. In particular, usage of horizontal parallelism in kernels could become less complicated.
References


Listing 5: For the AoS layout a pointer to a single element and an indexed element have the same type because both are laid out consecutively in memory. For the SoA layout the components of an indexed element are not consecutive in memory therefore a different pointer type SoA_Elm* and pointer calculation P(container,i) are necessary. Section Appendix A.1 explains how to define them and Section Appendix A.2 how to obtain a unified syntax for both layouts.

Appendix A. Implementation Details

Appendix A.1. Value and Access Type

When operating on containers with flexible data layout it is important to distinguish the value type that is used for storage of individual elements and the access type that is used for accessing elements in the container. The value type is the same for both layouts, namely Element from Listing 1, however, the access types are different as Listing 5 demonstrates.

If we see the container again as an $M \times N$ matrix then a container element corresponds to a matrix row. Consequently, $\text{container}[i]$ is simply the $i$-th element/row. In case of an AoS layout (row-major storage) the components within this element/row are laid out consecutively in memory in the same way as for a singleton element/row. A pointer to both of them has type Element* as shown in Listing 5. However, for an SoA layout (column-major storage) the $i$-th element/row is not laid out consecutively in memory, to the contrary, there is a large stride between the components $\text{comp0}$ and $\text{comp1}$ of the $i$-th element/row in the container. We must define a type in C (named SoA_Elm in Listing 5) that can represent the $i$-th row of a matrix which is stored in column-major order, and a function (named $\text{P()}$ in Listing 5) to identify its location in the matrix.

In high level numerical languages, like MATLAB [17], the above operation is easily performed as $\text{row=container(i,:)}$ which retrieves the $i$-th row no matter if row-major or column-major storage is used internally. Matrix libraries in C++, like Blitz++ [18], offer similar functionality, allowing the specification of internal storage orders and the definition of so called
views on subsets of a matrix, e.g., a row. However, we require a solution that works in C, and in addition works with named structure components and not just matrices.

The problem we face in case of the SoA layout is that pCE->comp0 and pCE->comp1 must reference values that are far apart in memory, and the type of pCE should disallow access to anything between these values. Since we cannot overload operators this is impossible in a strict sense in C. However, in practice we can solve the first problem by introducing a large padding between comp0 and comp1, and the second by hiding the name of the intermediate padding values.

```
struct SoA_Elm {
    Type0 comp0;
    PadType0 hidden_pad_name0;
    Type1 comp1;
    PadType1 hidden_pad_name1;
    Type2 comp2;
    PadType2 hidden_pad_name2;
};
```

The padding ensures for all $j$: sizeof(Type$j$)+sizeof(PadType$j$) = $C \cdot M$, which is the constant in the second addend of $g_{SoA}$, see Eq. 2. Accordingly, the undefined function $P()$ in Listing 5 is the offset calculation that corresponds to the first addend in $g_{SoA}$, it adds $C \cdot i$ to the base of the container and casts it to SoA_Elm*. Thus, the pointer pCE is of type SoA_Elm* and pCE->comp$j$ evaluates to the different components of the same element in the container. This realizes the same two stage indexing for the SoA layout as for the AoS layout. Now we only need to unify the syntax.

Appendix A.2. Macro Implementation

Listing 2 shows an example of a flexible container definition that can be configured with a single parameter to use the AoS or the SoA layout. The main functionality comes from the macro

```
#define ASA_STRUCT_CONTAINER \
(ContType, contSize, layout, N, t0, c0, ...) ...
```

in which the user specifies the container type name ContType, the container size contSize, the container layout layout, number of components $N$, and the types and names of the individual components. Similar to a template
specialization, macro processing allows macro specialization based on parameters. We use this functionality to invoke the correct processing in dependence on \( N \) and to branch into two specialized macro versions depending on the layout.

Both specialized versions implement three structures: the value and access types discussed in the previous section, and the actual storage type of the container. The internal names are created from the user specified name `ContType` with appropriate appendices.

```c
typedef struct { ... } ContType##_value_type;
typedef struct { ... } ContType##_access_type;
typedef struct { ... } ContType##_storage_type;
typedef ContType##_storage_type ContType;
```

The value type is defined for both layouts exactly like the type `Element` in Listing 1. For AoS the access type is the same as the value type and the storage type is simply an array of the value types as in Listing 1. For SoA the access type is implemented as shown in the previous section (`SoA_Elm`), while the storage type is an SoA container as in Listing 1.

Apart from the macro for the container definition the user only needs to know the macro `ASAat()` for the container access. It implements the offset computation according to the first addend in \( f_{AoS} \) for AoS (Eq. 1) and the first addend in \( g_{SoA} \) for SoA (Eq. 2).

```c
#define ASAat(ContType, cont, i) 
  ( /* AoS layout */
    ( (cont)[i] )
  )
#define ASAat(ContType, cont, i) 
  ( /* SoA layout */
    ( *(Cont##_access_type*)((char*)(cont)+C*i) )
  )
```

In contrast to the function `P()` from Listing 5 this implementation requires a macro because the offset computation must work for different container types and C does not support templates. Apart from the container instance `cont` and the index `i` we must also pass the container type `ContType` because the offset computation performs a type cast and the `typeof()` operator is not a standard C feature. The macro evaluates to the container access type `ContType##_access_type` defined by the container definition macro. This enables the two stage container access that is often used in practice, as demonstrated in Listing 2.