Data Layout Transformation Using LLVM/Clang

Omri Mor

August 18, 2019

Abstract

Modern systems contain a variety of data-parallel compute components, such as CPU vector units and GPU cores. Addressing these components requires a restructuring of application data so that it can be efficiently loaded from memory. In this paper, we present a backwards-compatible compiler-based data layout transformation technique that does not require preprocessing or runtime marshalling. Using our proposed extension, experimental results demonstrate a significant improvement over using an array-of-structures and can achieve near-performance-parity with manual transformation.

1 Introduction

Nearly all applications use *records* to organize related fields in a single composite structure. This allows for simple access to logically-connected data, such as the coordinates of and forces acting on an atom or molecule. Due to its structuring of data, records are sometimes referred to as *structures*, particularly when discussing C-derived language, due to its use of the `struct` keyword for defining such datatypes; we will henceforth use this term. Object-oriented languages expand this concept with objects that encapsulate behaviors in addition to state. Another aggregate, used when needing to address a collection of data, is an *array*, allowing index-based access to the constituent elements.

When the data addressed is itself a structure, it is often most straightforward to composing them as an *Array of Structures* (AoS). In this data layout, the fields of the structure are laid out consecutively in memory, with individual structures laid out one after the other as an array. An alternative layout is termed *Structure of Arrays* (SoA), wherein the order is reversed: the fields of successive structures are contiguous in memory and the arrays of fields laid out in the structure order. An example in C of the two is demonstrated in Listing 1.

When iterating over such aggregates, it is common to reference and operate on many or all fields of the structure in each iteration. An example from molecular dynamics is the computation of forces acting on an atom or molecule; in each iteration, the interactions from nearby molecules must be collected for each force and computed. Modern CPUs are highly optimized for latency and

Listing 1: Three variables demonstrating the `AoS`, `SoA`, and `ASA` layouts.

```
struct {          struct {              struct {
    double a;         double a[100];        double a[4];
    int    b;         int    b[100];        int    b[4];
    long   c;         long   c[100];        long   c[4];
} AoS[100];       } SoA;                } ASA[25];
```

have advanced hardware memory prefetching and large caches; since instructions will only address a single data stream, the AoS layout is beneficial for these machines, due to the data locality that results from storing all the fields of a structure nearby.

However, architectures more suited to high throughput, such as GPUs and CPUs that are able to take advantage of vector operations, will have much better performance when using the SoA layout. These architectures typically rely on applications written with massive data-parallelism, wherein the same operation is executed on a large amount of data. GPUs attempt to coalesce memory operations so as to achieve the high throughput required to keep its execution units furnished with data; when it is unable to coalesce memory operations due to the data being to far apart, as is common with the AoS layout (particularly with structures containing many fields), performance suffers greatly. CPUs with vector operations may be entirely unable to take advantage of them when the data isn't laid out in memory with the same fields in consecutive locations, at least not without significant preprocessing and manual effort that a vectorizing compiler cannot usually do automatically. When targeting CPUs supporting gather and scatter vector operations, compilers may still be able to vectorize the code, but such operations are expensive in terms of memory bandwidth and, similarly as in GPUs, performance suffers.

A third technique combines the advantages of both AoS and SoA layouts. Here, the structure contains small arrays for each field, usually some small multiple of the vector length of the system. Larger arrays can be constructed arrays of this structure; hence, the technique is *Arrays of Structures of Arrays* (ASA). An example of this layout is shown in Listing 1. The inverse of this, a *Structure of Arrays of Structures* (SAS), is also possible, and is occasionally useful; however, we here focus on the ASA data layout.

2 Design

We propose using GNU attribute syntax supported by LLVM/Clang to annotate specific structures as "array" types, wherein each field is transformed into an array of the specified length. Arrays of such structures are decomposed into ASA types, maintaining the total number of fields as necessary. References to fields of scalar variables of the type become references to the first element of the constituent arrays. Users are assumed to dynamically allocate the required

Listing 2: An example of a data structure using the proposed attribute.

```
struct __attribute__((array(4))) SoA {
    double a;
    int    b;
    long   c;
};

void foo(size_t len) {
    struct SoA A[100];
    struct SoA *B = malloc(sizeof(struct SoA[len]));

    for (size_t i = 0; i < 100; i++) {
        A[i].c = i * i;
    }
}
```

amount of memory; using the `sizeof` operator simplifies this process and allows the code to be portable, but doing so is otherwise the responsibility of the user. Pointer arithmetic is more complicated and requires additional thought, but our initial concept will disallow this. Type casting between pointers is, of course, problematic, but doing so violates the C standard due to type punning rules in any case.

Listing 2 demonstrates the use of our proposed syntax. An `array` attribute is attached to the definition of `struct SoA`. The compiler will note this and transform the `a`, `b`, and `c` fields into arrays of length 4 accordingly. The lengths of statically-sized arrays can be trivially computed at compile-time by using the fact that $\lceil \frac{n}{m} \rceil = \lfloor \frac{n+m-1}{m} \rfloor$. When the compiler supports variable-length arrays (VLA), it must do the same computation at runtime. By only computing the size of a VLA, it is possible to allocate the correct amount of memory on the heap rather than the stack, thus allowing supporting larger arrays; most system limit the stack memory of a process to a couple MiB at most.

Listing 3 shows the same program post-transformation. Note that this would never be seen by the user; rather, when the compiler transforms the C code to its internal representation, it will take into account the `array` attribute on structures.

3 Implementation

We build on top of LLVM/Clang 8.0.0, implementing the transformation by manipulating the Abstract Syntax Tree (AST). While such a design might be more suited to a source-to-source compiler such as ROSE, we decided that such a design is more difficult to use due to require multiple compilation steps. We also considered that it may be possible to expose the attribute to LLVM at the IR level and doing the transformation there, which would open up the

Listing 3: Listing 2 after being transformed.

```
struct SoA {
    double a[4];
    int    b[4];
    long   c[4];
};

void foo(size_t len) {
    struct SoA A[25];
    struct SoA *B = malloc(sizeof(struct SoA[(len+3)/4]));

    for (size_t i = 0; i < 100; i++) {
        A[i/4].c[i%4] = i * i;
    }
}
```

technique languages other than those supported by Clang; this, however, was not implemented in our current design.

The transformation consists of several steps:

1. Redefine the base type:
   (a) A declaration for record type T with an `array` attribute is parsed.
   (b) The length L supplied to the `array` attribute is recorded as an attribute on the declaration for T.
   (c) Each field in the declaration for T is replaced with an array of length L with the same base type.
2. Compute new bounds for arrays of type T, as described in Section 2.
3. Compute new indices for subscripts of arrays of type T. The new index is the value of the original index i divided by L: `i/L`. The index of the inner array is the remainder, `i%L`; it is stored as a subexpression inside the subscript expression object.
4. Member access expressions with a base expression subscripting an array of type T are replaced with a subscript expression. The base of the new subscript is the original member expression and the index is the subexpression stored previously.

Thus the compiler does the same transformation internally as demonstrated in the manual transformation of Listing 2 to Listing 3. Additional work was done to support the attribute syntax introduced in C++11 [8] and parameterizing the length base on C++ templates. A similar attribute syntax may be supported in a future version of the C standard [9, 1, 5]. An example demonstrating these capabilities is in Listing 4.

Listing 4: `array` attribute using C++11 syntax and a templated length.

```
template<std::size_t N>
struct [[clang::array(N)]] SoA {
    double a;
    int    b;
    long   c;
};
```

4 Evaluation Methodology

4.1 Benchmarks

We target NVIDIA general-purpose graphics processors using the CUDA [11] platform to demonstrate our data layout transformation. The following CUDA benchmarks were used:

***n*-Body:** a simple gravitational n-body simulation benchmark

Black–Scholes: an option pricing algorithm

SRAD: a diffusion method for ultrasonic and radar imaging applications [22]

The n-body simulation was derived from [6]. Black–Scholes was based on code from samples distributed with CUDA [13, 12, 16]. The speckle reducing anisotropic diffusion (SRAD) kernel was taken from the Rodinia benchmark suite [2, 3].

For each of the three benchmarks we compare a version using separately-allocated parallel arrays for each field in the record, a version using the `array` attribute with an optimized `length` parameter[1], and one using the `array` attribute with a `length` of 1. These versions use structure-of-arrays, array-of-structure-of-arrays, and array-of-structures, respectively. We additionally compare an explicit AoS version available in the n-body benchmark.

4.2 Hardware and Software

Benchmarks were run on a single NVIDIA Tesla V100 with 16 GiB of second-generation High Bandwidth Memory (HBM2). An allocation on the Bridges AI cluster [15] at the Pittsburgh Supercomputing Center was used.

CUDA 9.2 and our fork of LLVM/Clang 8.0.0 were the primary software used; the benchmarks did not have external dependencies.

5 Results

Figure 1 shows the performance of discrete arrays and that of structures with `array` attribute, normalized against the traditional AoS programming style. In

[1] This is usually the size of the CUDA thread block or a multiple thereof.

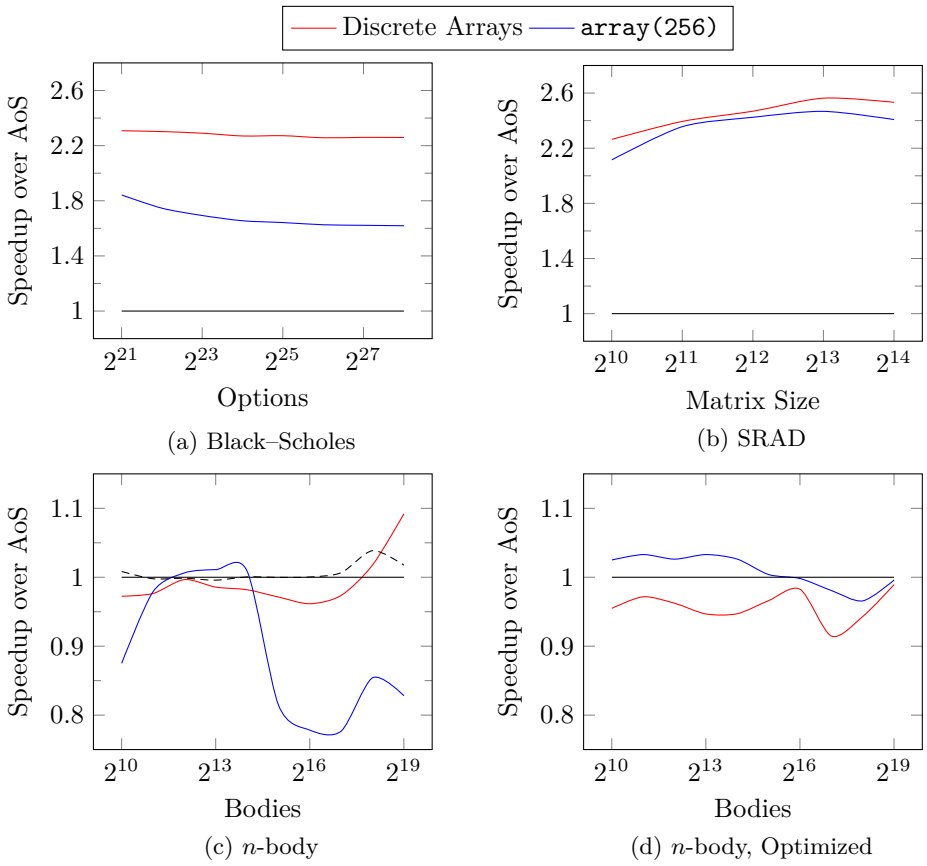

Figure 1: Application Speedup

Figure 1c the dashed line represents the speedup of the explicit AoS version as compared to the version derived from using a structure-of-arrays with an array length of 1. As shown, this validates our testing methodology of comparing against this baseline: there is little overhead, as the compiler can eliminate nearly all the extraneous operations.

As can be seen in Figures 1a and 1b, the speedup of simply annotating the structures with the `array` attribute is significant; in the case of the SRAD benchmark, we achieve within 95% of the performance as using a SoA technique with discrete arrays. While we still obtain a speedup of up to 1.8× in the Black–Scholes benchmark, we believe that the compute-bound nature of the algorithm (as compared to SRAD) is the reason that using discrete arrays is able to achieve an even greater speedup: the additional operations required to compute the indices becomes a more significant overhead.

The n-body benchmark shown in Figures 1c and 1d has some curious results. The net speedup for the optimized code is within 10% and the versions using AoS

can actually achieve better performance for some input sizes. We believe that this particular benchmark has enough computation that the device scheduler is able to sufficiently hide the memory access latencies; the additional compute requirements imposed by using SoA in any form can occasionally result in lower performance. A different compute architecture may be able to expose a greater performance difference; we are actively working on understanding and characterizing the phenomenon further. In any case, it is clear that our data layout transformation solution is able to meet and exceed the performance of hand-optimized code.

6 Related Work

Some languages, such as Intel's `ispc` [14], have explicit support for SoA data layout transformation for structures as part of their declaration. This can result in significant performance improvements when vectorizing, do to it no longer being necessary to use gather and scatter operations. However, `ispc`, while derived from C, is supported only by Intel. Our goal is to integrate a similar mechanism into otherwise standards-compliant C, such that the same code could be compiled regardless of whether the compiler supports the extension. Using types with a templated `array` attribute length parameter results in behavior similar to that available in `ispc`.

Strzodka [19] demonstrates a method of transparently converting between a traditional AoS and an SoA layout using C++ templates. Similarly, though specifically targeting arrays, Kokkos [4] also uses templates to achieve a comparable effect. These techniques are valuable and useful when using C++, but do not easily allow interoperation with other languages and porting existing applications to them is difficult and time-consuming.

The DL data layout transformation system [20] and work by Hoshino et al. [7] both suggest a runtime data layout transformation, though using different mechanisms. DL works by using a preprocessor to parse specially-formatted comments to generate marshalling code that converts structures in AoS or SoA formats to the intermediate ASA, allowing one to use the optimal layout for each device. Hoshino et al. build on top of OpenACC [21], introducing new pragmas that instruct the compiler to transform the data structures in a similar manner as DL. Both of these works focus on GPUs and don't consider vector instructions on CPUs as a viable target for their work. Additionally, both mechanisms used are more intrusive and require more manual control than our proposed design.

Majeti, Barik, Zhao, Grossman, and Sarkar [10] propose a compile-time mechanism for determining data layout by describing the structures in an external metadata file. Building on top of Habanero-C [17], they use the ROSE compiler infrastructure [18] to implement their transformations. A noted difference is that they require the use of the external metadata file and are not directly compatible with standards-compliant C.

7 Conclusions

We propose a data layout transformation extension to programming languages in the C and C++ families that can be implemented by compilers in a backwards-compatible fashion by using the standards-compliant attribute syntax available in C++11 and C2x. Our experimental results across a variety of CUDA benchmarks demonstrates that applications are able to obtain a speedup of over 2× when using the familiar AoS style of programming without requiring a restructuring of the application to use a structure-of-arrays. Applications that have a greater ratio of memory operations to computation can achieve performance that is within 95% of manual techniques. As the prevalence of GPU computation grows, allowing users to extract efficiency while maintaining the same application design becomes increasingly important. We continue to explore methods that may reduce the overhead required and that allows the same technique to be applied to CPU vectorization.

References

[1] A. Ballman. *N2335. Attributes in C*. Proposal for C2x. WG14, 2019.

[2] S. Che, M. Boyer, J. Meng, et al. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC '09. IEEE, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[3] S. Che, J. W. Sheaffer, M. Boyer, et al. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads". In: *Proceedings of the 2010 IEEE International Symposium on Workload Characterization*. IISWC '10. IEEE, 2010, pp. 1–11. DOI: 10.1109/IISWC.2010.5650274.

[4] H. Edwards, C. R.Trott, and D. Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216. DOI: 10.1016/j.jpdc.2014.07.003.

[5] J. Gustedt. *N2385. ISO/IEC 9899:202x (E)*. Information technology—Programming languages—C. Working Draft. WG14, 2019.

[6] M. Harris. *mini-nbody*. 2014. URL: https://github.com/harrism/mini-nbody.

[7] T. Hoshino, N. Maruyama, and S. Matsuoka. "An OpenACC Extension for Data Layout Transformation". In: *Proceedings of the First Workshop on Accelerator Programming using Directives*. WACCPD '14. IEEE, 2014, pp. 12–18. DOI: 10.1109/WACCPD.2014.12.

[8] International Organization for Standardization. *ISO/IEC 14882:2011*. Information technology—Programming languages—C++. Standard. 2011.

[9] International Organization for Standardization. *ISO/IEC 9899:2011*. Information technology—Programming languages—C. Standard. 2011.

[10] 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. DOI: 10.1007/978-3-642-54420-0_19.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA". In: *ACM Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[12] NVIDIA. *CUDA Samples*. Version 9.2. 2018.

[13] NVIDIA. *CUDA Samples Reference Manual*. TRM-06704-001_v9.2. Version 9.2. 2018.

[14] M. Pharr and W. R. Mark. "ispc: A SPMD Compiler for High-Performance CPU Programming". In: *2012 Innovative Parallel Computing*. InPar '12. IEEE, 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[15] Pittsburgh Supercomputing Center. *Bridges User Guide*. 2019. URL: https://psc.edu/bridges/user-guide.

[16] V. Podlozhnyuk. *Black–Scholes option pricing*. White Paper. Version 1.0. NVIDIA, 2013.

[17] V. Sarkar et al. *Habanero-C*. 2013. URL: https://wiki.rice.edu/confluence/display/habanero/habanero-c.

[18] M. Schordan and D. J. Quinlan. "A Source-to-source Architecture for User-Defined Optimizations". In: *Joint Modular Languages Conference*. 2003, pp. 214–223. DOI: 10.1007/978-3-540-45213-3_27.

[19] R. Strzodka. "Abstraction for AoS and SoA Layout in C++". In: *GPU Computing Gems: Jade Edition*. Ed. by W.-M. W. Hwu. Applications of GPU Computing series. Elsevier, 2012. Chap. 31, pp. 429–441. DOI: 10.1016/B978-0-12-385963-1.00031-9.

[20] I.-J. Sung, G. D. Liu, and W.-M. W. Hwu. "DL: A data layout transformation system for heterogeneous computing". In: *2012 Innovative Parallel Computing*. InPar '12. IEEE, 2012, pp. 1–11. DOI: 10.1109/InPar.2012.6339606.

[21] The OpenACC Organization. *The OpenACC Application Programming Interface*. Standard. Version 2.7. 2018.

[22] Y. Yu and S. T. Acton. "Speckle Reducing Anisotropic Diffusion". In: *IEEE Transactions on Image Processing* 11.11 (2002), pp. 1260–1270. DOI: 10.1109/TIP.2002.804276.