Extending SHAPES for SIMD Architectures

An approach to native support for Struct of Arrays in languages

Alexandros Tasos[1] Juliana Franco[2] Tobias Wrigstad[3] Sophia Drossopoulou[1] Susan Eisenbach[1]

[1] Imperial College London [2] Microsoft Research [3] Uppsala University

{a.tasos17,s.drossopoulou,s.eisenbach}@imperial.ac.uk juliana.franco@microsoft.com tobias.wrigstad@it.uu.se

Abstract

SIMD (Single Instruction, Multiple Data) instruction sets are ubiquitous on modern hardware, but rarely used in software projects. A major reason for this is that efficient SIMD code requires data to be laid out in memory in an unconventional manner, forcing developers to explicitly refactor their code and data structures in order to make use of SIMD.

In previous work, we proposed SHAPES, an abstract layout specification for enabling memory optimisations for managed, object-oriented languages. In this paper, we explain how, by extending SHAPES with well-known constructs from the literature, which are not specific to SIMD, we can extend SHAPES to compile programs to use SIMD instructions.

The resulting language (sketch) seems able to exploit SIMD capabilities without sacrificing ease of development.

1. Introduction

The premise of SIMD is that an operation is simultaneously applied to multiple units of data. In a CPU where a SIMD register can fit N elements, a SIMD addition instruction will perform N additions simultaneously. This implies a speedup bound of N in single-core performance.

A barrier to the wider exploitation of SIMD is the need to lay out data in a manner that can be at odds with application logic and good software engineering practise, such as abstraction. The semantics of SIMD memory instructions in most architectures require data to be laid out in a *contiguous fashion* in memory. That is, there are no memory scatter–gather operations available most of the time[1].

As an example of the impact of layout on SIMD, consider using SIMD for calculating the dot products of n 4D vectors, with fields x, y, z, and w. Iterating over an array of 4D vectors laid out as depicted in the top of Figure 1 runs only slightly faster than the scalar counterpart, as it iterates over one vector pair at a time and takes slightly less instructions per iteration. Changing the representation of the data to *Struct of Arrays (SoA)*—so that vector coordinates are stored in their own subarrays, as depicted in the bottom of Figure 1—enables the iteration over 4 vector pairs at a time.

[1] Even when scatter–gather instructions are available (*e.g.* on AVX), they will suffer multiple cache misses and have a cycle latency of 10 or more [6].

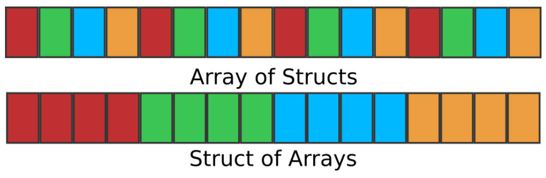

Figure 1. Array of structs (AoS), Struct of arrays (SoA).

Due to the contiguity requirement of SIMD instructions, SoA is better suited for SIMD. AoS, however, is "easier" for developers to use in virtually all imperative languages as it allows the development of *object abstractions* that group data together in a capsule. Representing a 4D point as a *4-tuple* of floats is superior to manipulating four separate floats and keeping track on which one is x and which one is y, etc. Translating into SoA therefore breaks object abstraction. This makes it impossible to pass a point as argument to a function, without creating a specialised proxy object wrapping the array and an index, and maintaining parallel structures in code. Thus, SoA transformations lead to increased complexity, code duplication, and bugs. The code for the example above (Figures 5 and 6 in the Appendix) demonstrates that shifting from AoS to SoA yields can yield significantly different—and more complicated—code, with little possiblity for reuse.

We postulate that automatic SoA transformations in a high-level language with constructs tailored to SIMD would allow developers to exploit the SIMD capabilities of CPUs in a convenient manner. With that in mind, we extend the SHAPES layout annotations [1] with programming concepts that are useful for writing programs with SIMD capabilities. This paper presents the main ideas and the design space.

2. Motivating Example

As an example of how automatic SoA transformations can be beneficial to leveraging SIMD, consider a (simplified) skeletal animation of 3D models in the MD5Anim format [8]. MD5Anim models consist of data, *e.g.* joints, weights and vertices. Animation changes the position of vertices in 3D space. Joints are organised in a tree. There is a 1-N relationship between joints and weights, and vertices and weights, respectively. Animation goes through three phases:

```
struct Joint { quaternion loc_orient, glob_orient;
  std::vector<Joint*> children; };
struct Weight { Joint* jnt; float x, y, z; float wv; };
```

Listing 1. MD5Anim structures definitions in C++.

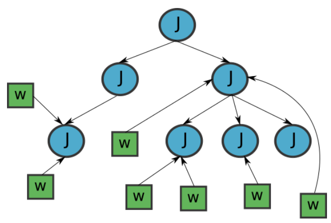

Figure 2. Memory layout of joints and weights.

Phase 1 Calculate the joints' new orientations in a top-down recursive manner.

Phase 2 (the focus of this paper) Calculate the weights' new positions from the weights' current positions and joints' new orientations.

Phase 3 Calculate the vertices' new positions from the weights' positions.

The data structures used are shown in Listing 1 and an example memory representation is shown in Figure 2. Although easily understandable, the implementation in Listing 1 inhibits optimisation opportunities regarding cache coherency and the use of SIMD instructions.

Shapes To address locality and AoS–SoA transformation, we have in the past proposed SHAPES [1], which is a set of abstract layout annotations for managed languages to enable memory optimisations under abstract memory. SHAPES leverages the following concepts: *pools* and *layouts*.

Pools are arenas for allocating objects of a specific type in contiguous memory. Pools are a grouping construct and not first-class citizens. *Layouts* describe the pools' memory organisation, for example how fields of objects should be ordered, or grouped for more efficient memory use.

The pools and layout annotations of the SHAPES system allow placing objects in structures close to each other (but not necessarily in order or even contiguously), on even strides and with hot fields grouped together. By adding pool and layout annotations to Listing 1, we obtain Listing 2.

Data structure declarations in SHAPES are parameterised by *pool parameters* that abstractly track the location of instances of the structures at run-time. In Listing 2, `wp` is a pool of `Weight` elements and the `Joints` the weights refer to are stored in the pool `jp` (Line 1). The `wp` pool is organised according to the layout `WeightSoa` (Line 11). This layout splits the objects in the pool into a SoA representation, effectively creating five contiguous subpools, one for eah field in the `Weight` struct. The respective (to Figure 2) memory layout we obtain is depicted in Figure 3.

```
1  struct Joint[jp: Pool(Joint[jp])] { // jp is a pool parameter
2      quaternion loc_orient;
3      quaternion glob_orient;
4      Array<unique Joint[jp]> children; }; // array of unique ptrs
5  struct Weight[wp:Pool(Weight[wp,jp]),jp:Pool(Joint[jp])] {
6      Joint[jp] jnt;
7      float x, y, z;
8      float wv; };
9
10 // Layout declaration used for wp (not shown)
11 layout WeightSoa = rec{jnt} + rec{x} +
12                    rec{y} + rec{z} + rec{wv};
```

Listing 2. MD5Anim structures definitions in SHAPES.

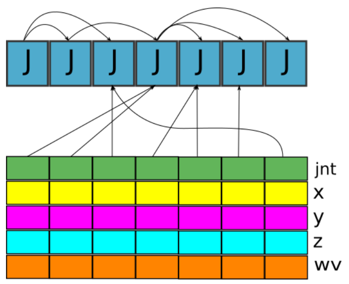

Figure 3. Memory layout achieved by using SHAPES.

3. SHAPES$^{\text{SIMD}}$

Although the layout obtained from Listing 2 is now eligible for SIMD optimisation (the only data dependencies for the new position of a weight are the joint's global orientation and the position coordinates of the weight), SHAPES is lacking in features suitable for SIMD. To that extent, we add the following extensions:

Arrays Arrays give an order for the placement of objects in memory, but considering the object-oriented, managed nature of SHAPES, arrays come with a caveat: Storing objects in arrays will store *pointers* to the objects contiguously, but not the *objects*. Using value semantics to overcome this problem, introduces a need for "default object values", likely incurs copying overhead, requires extra complexity to allow referencing, and are not suitable for recursive data structures. Instead, we rely on *unique pointers* to lay out array elements as if they had value semantics in the pool backing it.

Unique pointers A unique pointer [9] is the only pointer to an object in the system. Uniqueness is a strong property that allows many optimisations, including automatic memory management without a GC. We can leverage uniqueness for data placement—an array of unique pointers can be efficiently mapped to a form of value semantics under the hood. Since replacing a unique element drops the unique element entirely, a possible implementation strategy is to copy the source value into the designated memory placement in the array.

Borrowing Borrowing [10] allows relaxing uniqueness temporarily, usually for a well-defined lexical scope. Borrowing is key to reducing the pain of programming with unique references, as unique values are often destroyed upon reading to preserve uniqueness. We introduce borrowing as a means of accessing elements of an array directly without having to copy the element out of the array.

Exclusive pools Making a pool exclusive to one data structure is important to control object placement. If array A has an *exclusive* pool P for storing its objects, it also implies that objects in P do not have the right to further create or reference other objects in P, as that might lead to mismatches between the array and the pool backing the array and unclear object placement. Normal SHAPES pools permit this.

`SIMD` ***Environment*** We further extend SHAPES with an explicit `simd` environment, similar to ISPC [2] and Sierra [4]. The execution model of these languages with respect to SIMD is to consider multiple program instances that have the same program counter and are running in a lock-step manner. For now, this is the execution model we are considering for SHAPES' `simd` environment.

High-level Iteration For iteration over an array's elements, we add a `foreach` construct as syntactic sugar, which iterates over N array elements per iteration where N is controlled by target hardware. This is similar to `foreach` in ISPC. High-level iteration facilitates reasoning about code and thus simplifies optimisation.

3.1 SHAPES$^{\text{SIMD}}$ Example

Listing 2 shows the definitions in Figure 1 expressed using SHAPES.

The SHAPES$^{\text{SIMD}}$ the code for Phase 2—which calculates the weights' new positions from the weights' current positions and joints' new orientations—is shown in Listing 3. For comparison, an equivalent SIMD version that uses SoA manually is presented in Listing 7 in the Appendix. The code is derived from [7].

The `foreach` statement on Line 3 indicates that weights are processed in groups of N, where N depends on the SIMD instruction set being targeted. When targeting the SSE instruction set, for instance, 4 contiguously allocated weights in the pool are processed per loop iteration.

By changing the declaration of the `jp` pool to use the `WeightSoa` layout, we expect the following performance improvements:

- All joints are stored in a pool, thus we expect them to remain in the cache. This is important for the gather that is performed when fetching the quaternion components from each joint (Lines 5–8).
- More importantly, without polluting our code, we have ensured that the weights are stored in an SoA format, hence we were able to exploit SIMD parallelism without requiring major refactorings or sacrifices in readability.

```
1  [wp : Pool(Weight[wp,jp]), jp : Pool(Joint[jp])]
2  void move_weights(weights:Array<unique Weight[wp,jp]>){
3    simd {
4      foreach e <- weights { // e borrowed at each iteration
5        float x = e.jnt->glob_orient.x; // One gather
6        float y = e.jnt->glob_orient.y; // per quaternion
7        float z = e.jnt->glob_orient.z; // component
8        float w = e.jnt->glob_orient.w;
9
10       float px = e.x;
11       float py = e.y;
12       float pz = e.z;
13
14       float x2 = x + x; float xx2 = x * x2;
15       float y2 = y + y; float yy2 = y * y2;
16       float z2 = z + z; float zz2 = z * z2;
17
18       float xy2 = x * y2; float wx2 = w * x2;
19       float xz2 = x * z2; float wy2 = w * y2;
20       float yz2 = y * z2; float wz2 = w * z2;
21
22       float a11 = 1 - yy2 - zz2;
23       float a12 = xy2 + wz2;
24       float a13 = xz2 - wy2;
25       float a21 = xy2 - wz2;
26       float a22 = 1 - xx2 - zz2;
27       float a23 = yz2 + wx2;
28       float a31 = xz2 + wy2;
29       float a32 = yz2 - wx2;
30       float a33 = 1 - xx2 - yy2;
31
32       e.x = a11*px + a12*py + a13*pz;
33       e.y = a21*px + a22*py + a23*pz;
34       e.z = a31*px + a32*py + a33*pz;
35     }
36   }
37 }
```

Listing 3. Calculation of weights' new positions in SHAPES

3.2 Discussion

The combination of arrays and pools allows a developer to specify an array object that uses the already familiar syntax of accessing/modifying array elements, and also gives them the ability to change the underlying representation from AoS to SoA by simply changing a pool's layout.

Now, an array A with an exclusive pool P whose elements have type `unique` T, can *automatically* be represented in memory as a contiguous storage of objects (laid out according to the layout specification of P) in the same order as they are held by the array. (Unique pointers even make it possible to obtain the same layout for a singly linked list.)

As an alternative to the `simd` environment, we could rely on the autovectoriser. However, the "black box" behaviour of autovectorisers means that developers have no confidence over the machine code generated. Moreover, given the mul-

```
struct IspcSoa { uint64_t a; char b; uint64_t c; };
typedef soa<4> IspcSoa IspcSoaX4;
// Equivalent to
struct IspcSoaX4 {
  uniform uint64_t a[4];
  uniform char b[4];
  uniform uint64_t c[4]; };
```

Listing 4. ISPC Struct of Arrays

titude of optimising compilers available, it seems futile to attempt to cater to the lowest common denominator of autovectorisers. Instead, wrapping an operation in a `simd` block construct clearly states a programmer's intentions that the code should be compiled using SIMD instructions. Unless the data operated on has the type sketched above—an array A with an exclusive pool P whose elements have type `unique` T, compiling to SIMD is not sensible and the compiler can act accordingly—this may include emitting a warning.

Inside the `simd` block, the borrowing construct (Line 4) allows us to directly manipulate elements in arrays, voiding the need to assign to all of the fields of an object simultaneously.

Gaps in the Data The implementation details of pools in SHAPES are abstracted away from the developer. One such detail is the possibility of gaps between objects inside a pool. That is, these gaps can be filled in later when a new object is constructed inside a pool. We believe that making pool-backed arrays expose this feature is beneficial to the developer. The ability to guarantee that no gaps exist in the array (*e.g.* by leveraging moving GC) can be added as a later extension.

4. Related work

Languages with Explicit SIMD Support We have already discussed ISPC [2] and Sierra [4] and how SHAPES aims to adopt a subset of their features.

ISPC can transform a structure type into a layout that resembles SoA, via the keyword `soa<N>`. This defines a new structure type at compile time with the same fields, but the type of the field is changed from T to $T[N]$. Listing 4 shows such an example. Sierra has an equivalent keyword, `varying`.

We aim to improve upon ISPC and Sierra thusly:

– Because pools are part of the SHAPES runtime, padding will likely be avoided (there is padding between fields `b` and `c` of `IspcSoaX4` in Listing 4).
– In ISPC and Sierra, pointers to elements in an `soa` type need to be represented by a pointer to an array and an index. However, because SHAPES captures pool membership at the type-level, we can represent pointers by only an index, plus one pool pointer per activation record.
– We plan to optimise the pool representation so as to make it more amenable to the hardware prefetcher.

Layout Changing Constructs A package [5] for the Julia language allows the transformation of `AbstractArrays` into ones that uses SoA under the hood. These are however restricted to base types that are immutable `isbits` (*i.e.* a type with no reference fields). Unlike our approach, however, in this implementation all fields will be fetched at indexing and all fields must be modified when assigning into an array.

Mattis et al.[11] describe and implement an object layout for column-based databases intended to be easily optimisable by the PyPy JIT interpreter.

The Jai language design [3] targets game development, and aims to allow the layout of arrays to be changed to SoA.

5. Conclusion

All of our proposed extensions to SHAPES are useful on their own, with or without SIMD. By combining them and the `simd` environment, we hope to provide an environment that allows better exploitation of SIMD compared to existing approaches.

Moreover, despite the possibility of gaps in our implementation, we expect that the code generated will not be suboptimal, as we can exploit the fact that reading from and writing into the fields of empty slots should not affect the semantics of well-behaving programs and that conditional statements will not be translated into suboptimal code.

References

[1] Franco, Juliana, et al. "You Can Have it All: Abstraction and Good Cache Performance." Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, 2017.

[2] Pharr, Matt, and William R. Mark. "ISPC: A SPMD Compiler for High-Performance CPU Programming." Innovative Parallel Computing (InPar), 2012. IEEE, 2012.

[3] Rodriguez, Jose. A Jai Primer. URL: https://github.com/BSVino/JaiPrimer/blob/fc9ff9e722c190eeabe33a78d50d2588ae6b7e49/JaiPrimer.md Accessed: 2018-05-18.

[4] Leißa, Roland, Immanuel Haffner, and Sebastian Hack. "Sierra: a SIMD Extension for C++." Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing. ACM, 2014.

[5] Kornblith, Simon. Julia Structs of Arrays. URL: https://github.com/simonster/StructsOfArrays.jl/blob/v0.0.3/src/StructsOfArrays.jl Accessed: 2018-05-18.

[6] Fog, Agner. "The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers." Copenhagen University College of Engineering (2012): 02-29.

[7] Van Waveren, J. M. P. "From Quaternion to Matrix and Back." Id Software, Inc (2005).

[8] Henry, David. "MD5Mesh and MD5Anim Files Formats." URL: http://tfc.duke.free.fr/coding/md5-specs-en.html. Accessed: 2018-05-18.

[9] John Hogg. Islands: Aliasing Protection in Object-oriented Languages." *SIGPLAN Not.* 26(11):271–285. 1991.

[10] John Boyland. "Alias burying: Unique variables without destructive reads." *Software: Practice and Experience*, 31(6):533–553. 2001.

[11] Mattis, Toni, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. "Columnar objects: Improving the performance of analytical applications." In 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), pp. 197-210. ACM, 2015.

A. Appendix

```
#include <xmmintrin.h>
struct Vec4f { float x, y, z, w; };
void dp(float* dest, Vec4f* lhs, Vec4f* rhs, size_t len) {
 for(size_t i = 0; i < len; i++) {
    __m128 a0 = _mm_mul_ps( _mm_loadu_ps((float*) &lhs[i]), _mm_loadu_ps((float*) &rhs[i]) );
    __m128 a1 = _mm_movehl_ps(a0, a0);
    __m128 a2 = _mm_add_ps(a1, a0);
    __m128 a3 = _mm_shuffle_ps(a2, a2, 0); // Constant 0 broadcasts dot product to all SIMD lanes
    __m128 dot = _mm_add_ps(a3, a2);
    dest[i] = _mm_cvtss_f32(dot);
  }
}
```

Listing 5. Naive SIMD dot product calculation (1 dot product at a time). Written for the Intel SSE SIMD instruction set.

```
#include <xmmintrin.h>
struct Vec4fSoa { float *x, *y, *z, *w; };
void dp(float* dest, Vec4fSoa* lhs, Vec4fSoa* rhs, size_t len)
{
  // Assume for simplicity that len is a multiple of 4 (we do not show the code
  // responsible for handling the residual vector pairs).
  for(size_t i = 0; i < len; i += 4) {
    __m128 px = _mm_mul_ps( _mm_loadu_ps(&lhs->x[i]), _mm_loadu_ps(&rhs->x[i]) );
    __m128 py = _mm_mul_ps( _mm_loadu_ps(&lhs->y[i]), _mm_loadu_ps(&rhs->y[i]) );
    __m128 pz = _mm_mul_ps( _mm_loadu_ps(&lhs->z[i]), _mm_loadu_ps(&rhs->z[i]) );
    __m128 pw = _mm_mul_ps( _mm_loadu_ps(&lhs->w[i]), _mm_loadu_ps(&rhs->w[i]) );
    __m128 a0 = _mm_add_ps(px, py);
    __m128 a1 = _mm_add_ps(pz, pw);
    __m128 dots = _mm_add_ps(a0, a1);
    _mm_storeu_ps(dest, dots);
  }
}
```

Listing 6. SIMD Dot product calculation (4 dot products at a time). Written for the Intel SSE SIMD instruction set.

```
#include <xmmintrin.h>
#include <stddef.h>

struct quaternion { float x, y, z, w; };
struct Joint { quaternion loc_orient, glob_orient; };
struct Weights { Joint* jnt; float *px, *py, *pz; float *wv; };

void move_weights(struct Weights* arr, size_t len) {
    __m128 ONES = _mm_set1_ps(1);
    // Assume for simplicity that len is a multiple of 4 (we do not show the code
    // responsible for handling the residual weights).
    for (size_t i = 0; i < len; i += 4) {
        quaternion& orient0 = arr->jnt[i + 0].glob_orient;
        quaternion& orient1 = arr->jnt[i + 1].glob_orient;
        quaternion& orient2 = arr->jnt[i + 2].glob_orient;
        quaternion& orient3 = arr->jnt[i + 3].glob_orient;

        // Gathers have to be performed explicitly with intrinsics
        __m128 x = _mm_set_ps(orient3.x, orient2.x, orient1.x, orient0.x);
        __m128 y = _mm_set_ps(orient3.y, orient2.y, orient1.y, orient0.y);
        __m128 z = _mm_set_ps(orient3.z, orient2.z, orient1.z, orient0.z);
        __m128 w = _mm_set_ps(orient3.w, orient2.w, orient1.w, orient0.w);

        __m128 px = _mm_loadu_ps(&arr->px[i]);
        __m128 py = _mm_loadu_ps(&arr->py[i]);
        __m128 pz = _mm_loadu_ps(&arr->pz[i]);

        __m128 x2 = _mm_add_ps(x, x);
        __m128 y2 = _mm_add_ps(y, y);
        __m128 z2 = _mm_add_ps(z, z);
        __m128 xx2 = _mm_mul_ps(x, x2);
        __m128 yy2 = _mm_mul_ps(y, y2);
        __m128 zz2 = _mm_mul_ps(z, z2);

        __m128 xy2 = _mm_mul_ps(x, y2);
        __m128 wz2 = _mm_mul_ps(w, z2);
        __m128 xz2 = _mm_mul_ps(x, z2);
        __m128 wy2 = _mm_mul_ps(w, y2);
        __m128 yz2 = _mm_mul_ps(y, z2);
        __m128 wx2 = _mm_mul_ps(w, x2);

        __m128 a11 = _mm_sub_ps(_mm_sub_ps(ONES, yy2), zz2);
        __m128 a12 = _mm_add_ps(xy2, wz2);
        __m128 a13 = _mm_sub_ps(xz2, wy2);
        __m128 a21 = _mm_sub_ps(xy2, wz2);
        __m128 a22 = _mm_sub_ps(_mm_sub_ps(ONES, xx2), zz2);
        __m128 a23 = _mm_add_ps(yz2, wx2);
        __m128 a31 = _mm_add_ps(xz2, wy2);
        __m128 a32 = _mm_sub_ps(yz2, wx2);
        __m128 a33 = _mm_sub_ps(_mm_sub_ps(ONES, xx2), yy2);

        __m128 npx = _mm_mul_ps(a11, px);
        npx = _mm_add_ps(npx, _mm_mul_ps(a12, py));
        npx = _mm_add_ps(npx, _mm_mul_ps(a13, pz));

        __m128 npy = _mm_mul_ps(a21, py);
        npy = _mm_add_ps(npy, _mm_mul_ps(a22, py));
        npy = _mm_add_ps(npy, _mm_mul_ps(a23, pz));

        __m128 npz = _mm_mul_ps(a31, px);
        npz = _mm_add_ps(npz, _mm_mul_ps(a32, py));
        npz = _mm_add_ps(npz, _mm_mul_ps(a33, pz));

        _mm_storeu_ps(&arr->px[i], npx);
        _mm_storeu_ps(&arr->py[i], npy);
        _mm_storeu_ps(&arr->pz[i], npz);
    }
}
```

Listing 7. Weight displacement calculation (SSE intrinsics).