Software Performance and Class Layout

May 28, 2023Memory Subsystem Performance4 Replies

Up until this point, most of the memory optimizations we did were related to memory access pattern (e.g For Software Performance, the Way Data is Accessed Matters!) or decreasing total memory access count (e.g. Decreasing the Number of Memory Accesses 1/2). These techniques were good because, although they are not simple, the changes are localized in a single place. However, there is a limit to how much speed improvement you can obtain by using them alone. In other words, “it’s time to pull out the big guns”.

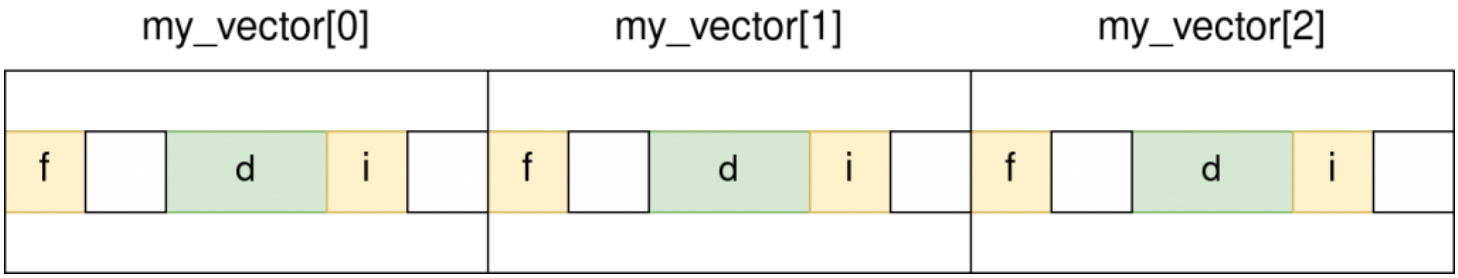
This is where data layout modification comes into the picture. When we say class data layout, we are interested in how the compiler organizes our data in memory. Data layout is known at compile time, and determined by the compiler. In contrast, memory layout is known at runtime, and determined by the memory allocator. We will cover the impact of memory allocators in one of the upcoming posts.

What is a class or a struct from the developer’s point of view, from the computer’s point of view is just some data grouped together in memory in a certain way. Consider the following code:

```
class my_class {
   float f;
   double d;
   int i;
};
std::vector<my_class> my_vector;
```

The code consists of class `my_class` and vector of `my_class` called `my_vector`. The offset of the member `f` from the beginning of the class is 0. The offset of the member `d` is not 4, as one might expect, but 8. The reason is that doubles must be 8-byte aligned on most architectures, therefore the compiler will insert a padding of 4 bytes between `f` and `d`. The offset of the member `i` is 16. The total class size is not 20 however, but 24. The class needs to be 8-byte aligned as well, because of the member `d` which is 8-byte aligned. Therefore, the compiler inserts an additional 4 bytes of padding after `i`.

This class, when stored in a vector, looks like this:



The CPU accesses floats at offsets 0, 24 and 48; doubles at offsets 8, 32 and 56; and integers at offsets 16, 40 and 64. If the size of the cache line is 64 bytes, then a single cache line can hold 2 full instances of `my_class` and 2/3 of the third instance.

In the context of our investigation, data layout modifications mean changing how the compiler stores data in the memory. For example:

- Removing members from a class.
- Adding new members to a class.
- Adding or removing padding.
- Changing the order of members inside a class.

When developers do their job, they often change the data layout. However, the goal is different. Most of the time they are implementing new or removing unused features. In our case, we change the data layout to make our software faster.

Increasing Performance Through Changing the Data Layout

There are two guidelines when optimizing the data layout that essentially govern the process:

- *What is accessed together should be close to one another in memory*
- *What is not accessed together should be moved away from one another*, to make place for things that are accessed together.

In this post, we will see how these principles apply when working with code that processes classes. This approach is not however without its downsides. There are two potential problems with changing the class data layout if we want to improve software performance:

- A small change to the data layout (e.g. extracting a rarely accessed member in an auxiliary class) might require many changes in various places in your code base.
- A small change that improves the performance of one hotspot might introduce performance regression in another hotspot. For example, if hotspot A doesn't use member X of class C, but hotspot B does use it, then moving X outside of C will make hotspot A faster but hotspot B slower.

Techniques

Declare Members Used Together Close to One Another

This is a very simple recommendation aimed at improving locality of reference. If you have two or more class members that you access together in your hot loop, you should declare them one after another in the class definition. Example:

```
class my_class {
   int a;
   int b;
   ...
   int z;
};
int sum_all(my_class* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].a + m[i].z;
    }
    return sum;
}
```

In the hot loop in function `sum_all`, the members `a` and `c` are accessed together. Therefore, they should be declared one after another in the definition of `my_class`.

```
class my_class {
   int a;
   int z;
   int b;
   ...
};
```

When declared like this, the compiler places `a` and `c` one after another in memory. This increases the chances that the two members will be in the same cache line and can increase performance. The bigger the gap between the two members accessed together, the bigger the benefit from this rule.

Although this rule at first point sounds quite reasonable, sometimes it won't work. For example:

- If the class is smaller than the cache line size, the layout of its members is often not important. This recommendation mostly applies to large classes (larger than 128 bytes).
- If `a` and `z` are not directly declared one after another, but are "close enough" in the sense that they nevertheless belong to the same cache line.
- If `a` is at the beginning of the class and `z` is at the end of the class, then if the class is stored in a vector and accessed sequentially, the `z` in position X and `a` in position X+1 will be direct neighbors.

Overall, this is a good recommendation generally, both for maintainability and performance, but may fail in some specific details.

Move Away Unused Members

If your hot loop is not accessing some data members of the class, you can move them to auxiliary classes to improve the loop's performance. For example:

```
class my_class {
   int m1;
   int m2;
   int m3;
};
int sum_all(my_class* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].m2;
    }
    return sum;
}
```

In this example, the hot loop accesses members `m1` and `m2` of our class `my_class`, but not the member `m3`. Since the data is brought from the memory to the data caches in cache line sizes (typically 64 bytes), this means that every time we access `m1` and `m2`, the member is `m3` is also brought from the memory to the data caches. This wastes memory bandwidth and moving this member to an auxiliary class improves performance. The transformation looks like this:

```
class my_class_base {
   int m1;
   int m2;
```

```
};
class my_class_aux {
   int m3;
};
int sum_all(my_class_base* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].m2;
    }
    return sum;
}
```

Bring in Used Members from Other Classes

If a hot loop is accessing two or more classes, performance can be improved by bringing in part of class X to class Y. For example:

```
class my_class1 {
    int m1;
    int m2;
};
class my_class2 {
    int a1;
    int a2;
}
int sum_all(my_class1* m1, my_class2* m2, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m1[i].m1 + m1[i].m2 + m2[i].a1;
    }
    return sum;
}
```

In this hotloop, we access both members of the class `my_class1`, but only one member of the class `my_class2`. This makes for inefficient use of the memory subsystem, since `a2` is brought to the data caches together with `a1`. One possible solution would be to move the member `a1` to class `my_class1`. Here is the resulting code:

```
class my_class1 {
    int m1;
    int m2;
    int a1;
};
class my_class2 {
    int a2;
};
int sum_all(my_class1* m1, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m1[i].m1 + m1[i].m2 + m1[i].a1;
    }
    return sum;
}
```

In this case, the class `my_class2` is not accessed at all. The members `m1`, `m2` and `a2` are stored in the same place, and access to `m1` will be on the same cache line as `m2` and `a1` and therefore very cheap.

Bring in Members Accessed Through Pointers

If a class accesses a piece of data by dereferencing a pointer, then getting rid of the pointer dereference by making the pointed data part of the same class can help performance. For example:

```
class my_class {
   int m1;
   int* p_a1;
};
int sum_all(my_class* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + *m[i].p_a1;
    }
    return sum;
}
```

The member `a1` is accessed through a pointer `p_a1`. If it is possible to get rid of the pointer and make the data itself part of the class, this would help performance. The resulting transformation looks like this:

```
class my_class {
   int m1;
   int a1;
};
int sum_all(my_class* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].a1;
    }
    return sum;
}
```

Although it is not always possible to bring in the pointers, it can be very beneficial for several reasons:

- The members `m1` and `a1` share the same cache line, so access to the first one automatically makes access to the second one very cheap.
- Decreases memory fragmentation, since there are fewer calls to the memory allocator.
- Decreases instruction count, since access to `a1` doesn't require pointer dereferencing.
- Dereferencing a pointer can often result in a data cache miss, since a pointer can point to any place in memory. Bringing in the pointed data fixes the problem of data cache misses.

Denormalization

The term *denormalization* comes from the world of databases and means having identical copy of the same data in more than one place to increase the locality of reference and improve performance. To illustrate it, consider the following example:

```
class shared {
    int a1;
};
class my_class_1 {
   int m1;
   shared* s;
```

```
};
class my_class_2 {
   int m1;
   shared* s;
};
int sum_all_1(my_class_1* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].s->a1;
    }
    return sum;
}
int sum_all_2(my_class_2* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].s->a1;
    }
    return sum;
}
```

The data in class `shared` is accessed through pointers by both `my_class_1` and `my_class_2`. So, if `shared.a1` is modified, this is reflected in both classes. The problem with this approach is the pointer dereference, as explained in the previous section.

A possible solution is to create a copy of `shared` in both `my_class_1` and `my_class_2`. The advantages of this approach are explained in the previous section. Here is how the modified data layout would look like:

```
class shared {
    int a1;
};
class my_class_1 {
   int m1;
   shared s;
};
class my_class_2 {
   int m1;
   shared s;
};
int sum_all_1(my_class_1* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].s.a1;
    }
    return sum;
}
int sum_all_2(my_class_2* m, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += m[i].m1 + m[i].s.a1;
    }
    return sum;
}
```

The disadvantage is clear: updating the value of `shared s` and propagating those updates to both classes now becomes much more difficult.

Structure Of Arrays (SOA)

The last technique that changes the data layout is called *structure of arrays (SOA)*. It is very famous among people striving for peek performance and vectorization[1]. The idea is to take a vector of classes, in this framework also known as *array of structs*, and convert it to a single class, where each member is an array of original data members. To illustrate, have a look at bellow code:

```
class my_class {
   int a;
   int b;
   int c;
};
my_class example_array[20];
```

To convert it to struct of arrays, we get rid of the original array and convert each member to an array. The resulting code looks like this:

```
class my_classes {
   int a[20];
   int b[20];
   int c[20];
};
my_classes example_array;
```

This creates three arrays named `a`, `b` and `c` of simple type `int`. For sequential memory accesses, this approach works very well most of the time. Only the data that is consumed by the hot loop is actually fetched from the memory. It also enables compiler autovectorization, because vectorization works best with arrays of simple data types.

Of course, this approach is not without its problems:

- It requires a large rewrite touching many components in unusual ways.
- It locks you out of many possibilities. For example, you cannot store class `my_classes` in a hash map.
- It works well only with sequential memory access. With other memory access types, the results might not be satisfactory.
- If the sizes of arrays are a power of two, it can easily lead to cache conflicts.
- If your class has many arrays, this can create problems on the hardware level. Inside the CPU, there are data prefetchers that can figure out the memory access pattern and prefetch the data before the instruction is even issued. When the prefetcher detects a memory access pattern, it forms a *data stream*. A data stream consists of memory addresses that the program accessed in the past according to some predictable pattern. But the number of streams is limited and architecture dependent, and accessing more than that number can result in performance degradation.

Additional Considerations

Although the above techniques are good rules of thumb, there are a few additional considerations when changing the class layout.

Data Padding

As explained earlier, the compiler inserts padding to force correct alignment for all members in the class. However, the CPU is unaware of padding data and it fetches it from the memory to the data caches just like any other data. This wastes memory bandwidth.

Sometimes, removing padding can result in speed improvements. There are two techniques to control padding:

- *Rearranging data to avoid padding*: sometimes, rearranging data can remove the padding. For example, declaring our example class as `struct my_class { float f; int i; double d; }; would completely remove data padding.
- *Using* `__attribute__((__packed__))`: it will remove padding, but the downside of this approach is possible unaligned access to some data (which most architectures however support but with a penalty if a piece of data is split between two cache lines).

You can use a tool called `pahole` to investigate class and struct padding. Example of `pahole` output for a class we will use later in our testing:

```
struct complex_t<0, 0> {
    float                      re;                   /*     0     4 */
    int                        p1[0];                /*     4     0 */
    /* XXX 4 bytes hole, try to pack */
    double                     im;                   /*     8     8 */
    int                        p2[0];                /*    16     0 */
    ***
    /* size: 16, cachelines: 1, members: 4 */
    /* sum members: 12, holes: 1, sum holes: 4 */
    /* last cacheline: 16 bytes */
};
```

As you can see, the tool prints offsets, sizes and padding for all the members of the class.

Random and Strided Memory Accesses

Up to this point, everything we were talking about applied to data stored in continuous storage (arrays and vectors) and sequential memory accesses (running through the array from one side to another without skipping any elements).

Similar reasoning applies to random and strided memory accesses, but with a caveat. Let's explain it with an example.

Let's assume our class is 48 bytes in size, and that we are accessing all the members in the class. Let's also assume that the cache line size is 64 bytes. If instances of a class are stored in an array, then this is how class instances are distributed among cache line sizes:

Class Instance	Position in Cache Lines
Instance 0	Bytes 0 – 47: Cache Line 0 [0 – 47]
Instance 1	Bytes 0 – 15: Cache Line 0 [48 – 63] Bytes 16 – 47: Cache Line 1 [0 – 31]

Class Instance	Position in Cache Lines
Instance 2	Bytes 0 – 31: Cache Line 1 [32 – 63] Bytes 32 – 47: Cache Line 2 [0 – 15]
Instance 3	Bytes 0 – 47: Cache Line 2 [16 – 63]

In the above table, we see that four class instances fit three cache lines. Out of those four instances, two instances fit one cache line, and two instances are split between two cache lines.

In the case of sequential data access, the fact that some instances are split between two cache lines doesn't cause problems. Although an instance X can be split between cache line Y and Y + 1, the next instance X + 1 will use the data from the cache line Y + 1. Everything that gets transferred from the memory to the data caches actually gets consumed.

But in the case of random or strided accesses, there is a problem. With those types of accesses, the instances that are split between two cache lines are slower to access, because they require two memory transfers, one for each cache line.

In case of random accesses or strided accesses, one could try to **pad** the class so the size is a multiple of cache line size. For example, if the cache line size is 64 bytes, then it is guaranteed that classes with sizes 8, 16, 32 or 64 bytes will never be split among two cache lines.

Changing Data Layout on the Fly

If you will be running through your dataset many times, instead of changing the data layout, an alternative is to create a temporary, smaller data structure, with only the needed data. Smaller data structures are faster to work with.

Take as an example sorting. Let's say we have an array of really large classes, but we want them sorted according to their index. Here is the original class:

```
class big_class {
   int index;
   ...
};
void my_sort(std::vector<big_class>& v) {
    std::sort(v.begin(), v.end(), [](const big_class& l, const big_class& r) { return l.index <
r.index; });
}
```

The algorithmic complexity of function `std::sort` is `O(n log(n))`, which means that for the large size of vector `v`, one could expect that the vector is passed through several times.

We could convert this vector to a smaller temporary representation on the fly, sort the smaller vector, and then unpack the large vector. This also saves the time needed to copy `big_class` during sorting. Here is the implementation:

```
class small_class {
    int index;
    int pointer;
};
void my_sort(std::vector<big_class>& v) {
    std::vector<small_class> tmp;
    tmp.reserve(v.size());
    for (int i = 0; i < v.size(); i++) {
        tmp.push_back({v[i].index, i});
    }
    std::sort(tmp.begin(), tmp.end(), [](const small_class& l, const small_class& r) { return
l.index < r.index; });
    std::vector<big_class> result;
    result.reserve(tmp.size());
    for (int i = 0; i < tmp.size(); i++) {
        result.push_back(v[tmp[i].index]);
    }
    v = std::move(result);
}
```

The original code is now split into three parts. The first part creates a a vector of `small_class` by using the `index` data from the big class (lines 10-12). Second part does sorting on the `small_class` (line 14). And the third part recreates the original `big_class` sorted array (lines 18-20).

There is certainly a large overhead of doing the modifications on the fly. The program executes more instructions. But if the original class was large and we were iterating many times over it, in that case the additional overhead might have paid off. We investigate this in the experiments section.

Experiments

All the experiments were performed on Intel(R) Core(TM) i5-10210U, with TurboBoost disabled. The compiler is CLANG 15. Each measurement was performed five times, and we report an average number. The source code is available here.

Class Size

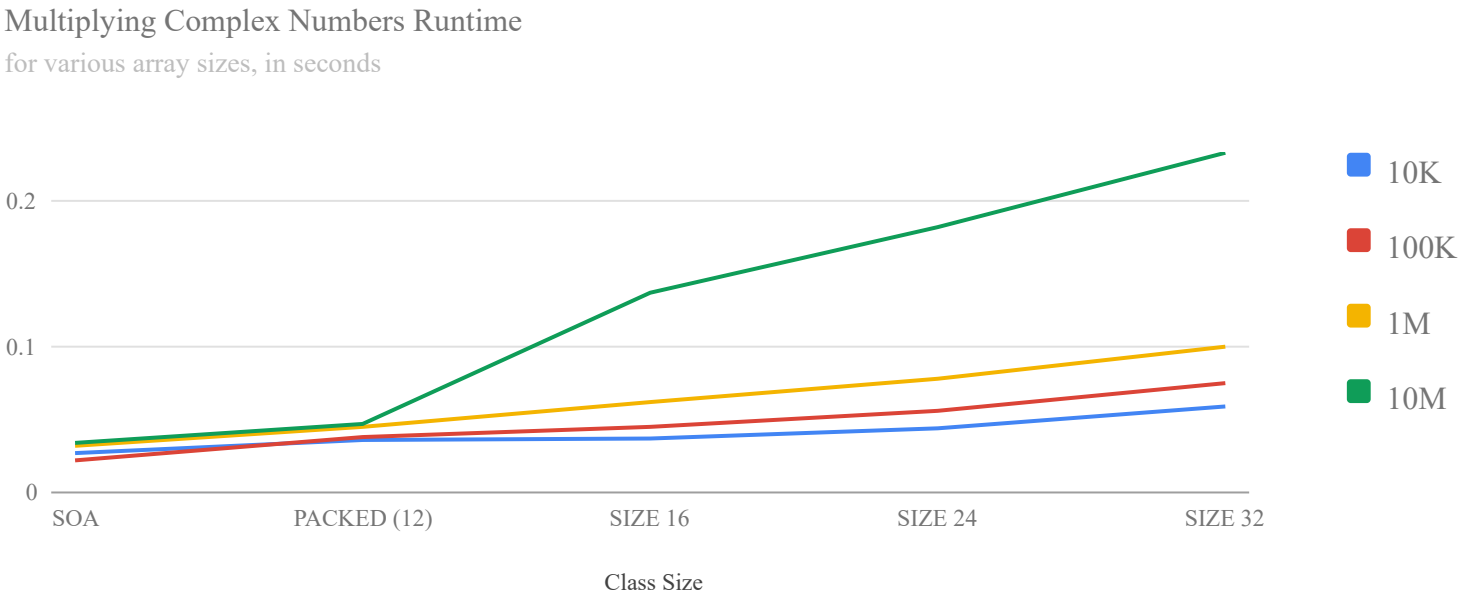
For this experiment, we use a class called `complex_t` which is used to model complex numbers. Here is the source code:

```
template <int padding1, int padding2>
struct complex_t {
    float re;
    int p1[padding1];
    double im;
    int p2[padding2];
};
```

The class has two template parameters called `padding1` and `padding1`, used to control the class size and how close members `re` and `im` are in memory. For the experiment with class size, the member `p1` is always zero. We test class sizes 16, 24 and 32.

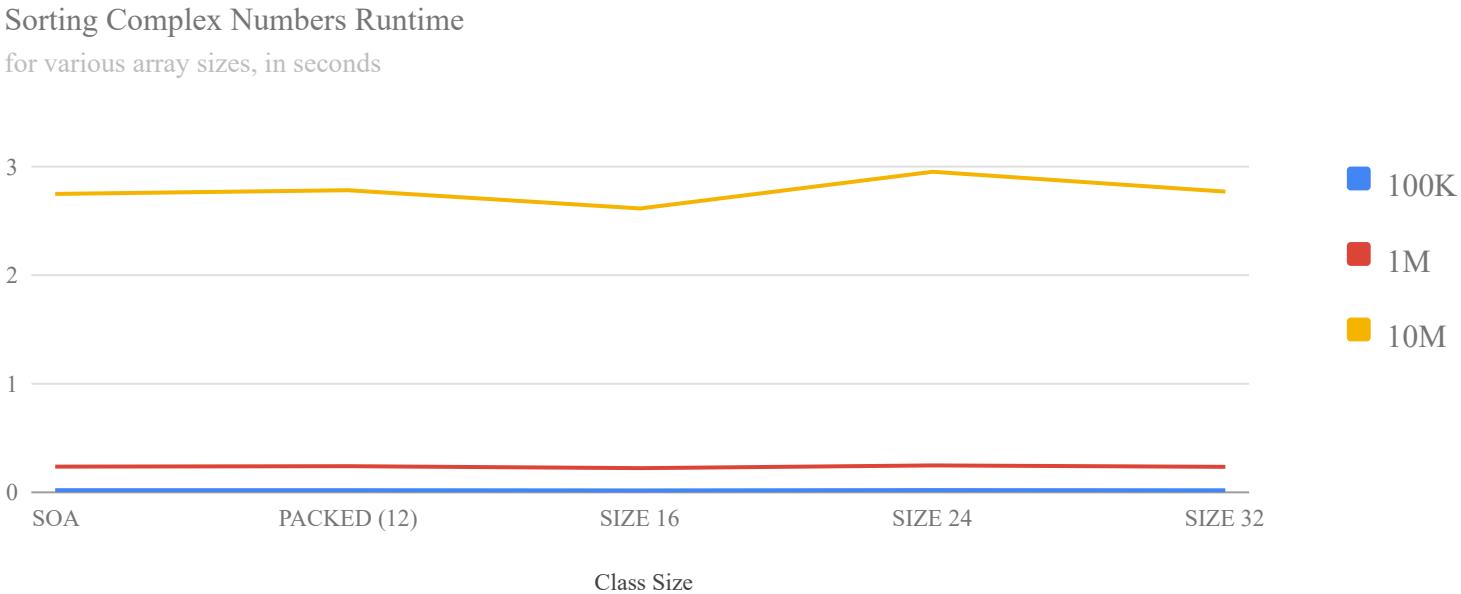
In addition, we provide an implementation using a packed struct (which removes the implicit padding between members `re` and `im` and has a size 12) and SOA implementation (which keeps members `re` and `im` in separate arrays).

The first test is taking two arrays of complex numbers and multiplying them. Here are the runtimes for various array sizes:



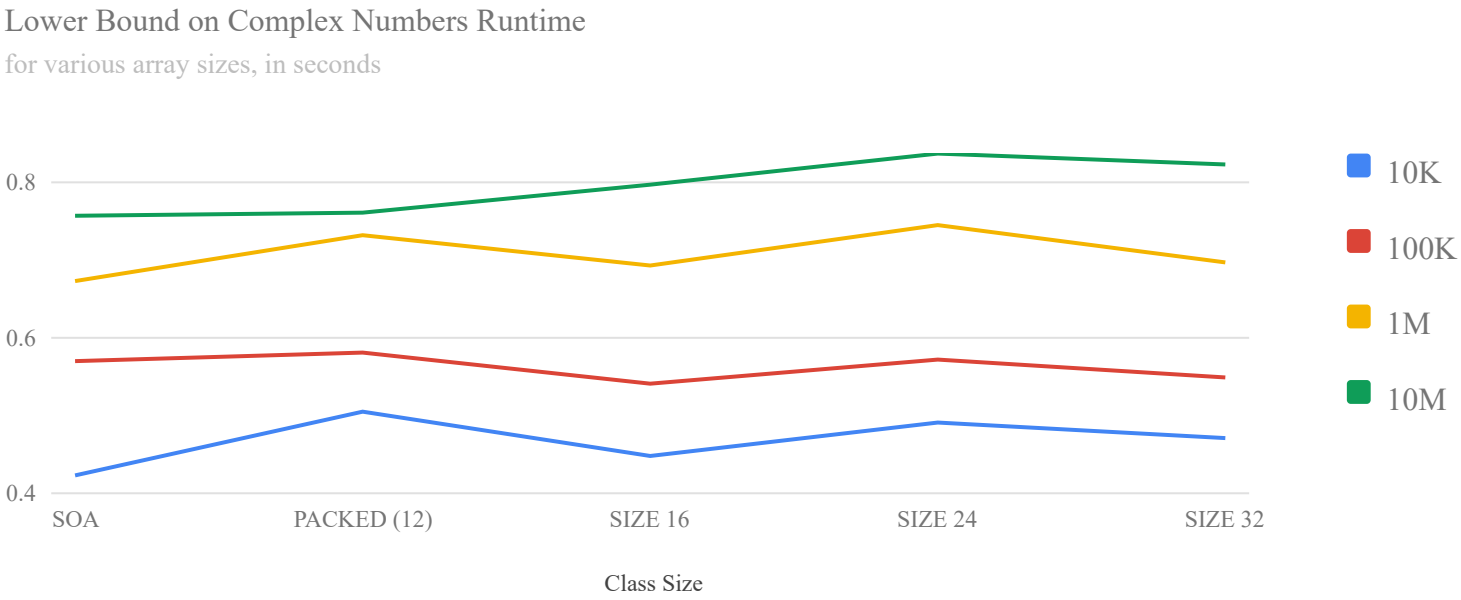
From this graph, it is obvious that the bigger the class the longer the runtime. This is especially noticeable for the largest array size (10 M elements, or 128 MB). SOA is fastest, but the compiler managed to vectorize SOA implementation, which can be one of the reasons why SOA is the fastest compared to e.g. packed implementation.

The second test is about sorting: how much time does it need to sort an array of complex number, depending on the class size and array size. This time the graph looks differently:



As you can see, this time the sorting runtime doesn't depend on the class size. The reason is that sorting is a speculation-limited, not memory-limited problem. Most of the hardware inefficiencies come from the hardware wrongly guessing the outcome of a branch and having to rerun instructions. For this reason we don't see the performance of sorting depending drastically on the class size.

And, the third operation is an implementation of `std::lower_bound` on a sorted array (which is essentially a binary search on a sorted array realized through random memory accesses). Here are the runtimes.



The numbers for this algorithm look different. Although the runtime depends on the class size, the dependence is not linear. We notice that the runtime is shorter if the class size is a multiple of cache line size (in our case, the cache line size is 64 bytes, so the runtimes for class size 16 and 32 are faster).

Changing Data Layout on the Fly

For this experiment, again we take our large class `complex_t` and sort it in two ways. The first way, we just call regular `std::sort`. The second way, we create a temporary struct called `proxy` which contains only fields `re` and `im` used for sorting as well as field `pos` which holds the position in the sorted array. We sort the `proxy` array, and then we recreate the sorted vector of `complex_t` based on the sorted `proxy` vector. The code looks like this:

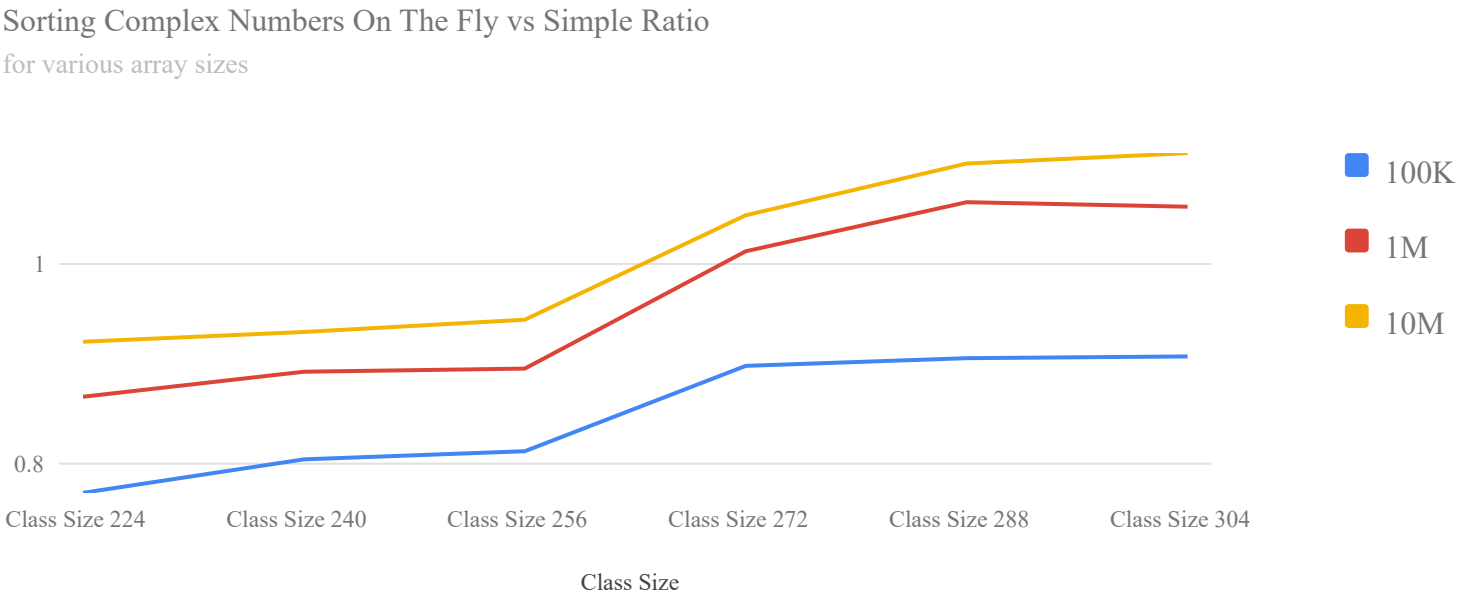
```
struct proxy {
    complex_simple n;
    int pos;
    bool operator<(const proxy& other) {
        return n < other.n;
    }
};
std::vector<Complex> vec_copy = vec_original1;
std::vector<proxy> vec_small(size);
for (int i = 0; i < size; ++i) {
    vec_small[i].n.re = vec_copy[i].re;
    vec_small[i].n.im = vec_copy[i].im;
    vec_small[i].pos = i;
}
std::sort(vec_small.begin(), vec_small.end());
for (int i = 0; i < size; ++i) {
    vec_original1[i] = vec_copy[vec_small[i].pos];
}
```

The class `complex_simple` within the `proxy` struct (line 2) is the small class holding only the members that are used for sorting (`re` and `im`). In addition, we need the `pos` to store the position in the sorted array.

For this type of sorting, we create a copy of the original vector in `vec_copy` on line 10. The loop on lines 13-17 creates a vector of `proxy` objects which are sorted on line 19. Finally, we recreate the original vector `vec_original1` on line 22.

Compared to the original simple sort, this type of sorting (1) creates a copy of the original array to be sorted (2) constructs a vector of proxies and (3) recreates `vec_original` using `vec_copy` and vector of proxies. Besides needing additional memory, it performs more steps, albeit on a vector of smaller objects.

We performed measurements on three vector lengths: 100K, 1M and 10M complex numbers. We also sorted class sizes 224, 240, 256, 272, 288, 304. Here are the ratios between thee on the fly version and simple call to `std::sort`.



The ratio above 1 means that the on the fly version is faster; conversely, the value under 1 means that the simple version is faster.

Sorting has an algorithmic complexity of `O(n log n)`, which means we will run a few times over the dataset, but not too many times. With this in mind, we see that sorting only starts to pay off when the class size is 272 bytes or larger. This method indeed has a huge overhead and therefore a limited applicability.

Final Words

In this post we explored how class data layout influences software performance. It is indeed possible to improve software performance by keeping all the accessed data in one place and moving away everything else. Lean classes result in less wasted memory transfers and better performance. The downside of these techniques is that they are quite intrusive and can require large code rewrites.

This post would make a very good introduction to a book about data-oriented design and entity-component-system paradigms. Those paradigms are used in game development where performance is

important. One of the key premise of DOD and ECS is that an object (called *entity*) is broken down into components and components are stored independently of one another in a fashion similar to storing data in relational databases. For readers interested in this topic, I would recommend a book by Richard Fabian called *Data-Oriented Design*.