Efficient memory management

2023 ESC at Sesame

Andrea Bocci

CERN

disclaimer

disclaimer

this is my first time preparing and teaching this course , so your feedback is very welcome !

if we have extra time after covering all the material, I'll be happy to answer more questions, or propose more exercises

why memory ?

what is computer memory …

- *memory* refers to the storage used by a program to read and write data

- a *virtual memory* OS can map different hardware to a single address space:
  - system memory (usually DDR SDRAM), allocated with `malloc()` or `new`
  - GPU memory: *e.g.* CUDA unified memory
  - HBM memory: *e.g.* on the latest Xeon Max CPUs
  - disk (SSD or HDD) areas: *e.g.* swap space or `mmap()`'ed files

- … and why is it important ?

- from the point of view of the CPU, most memory is *slow*
  - this is the single most important factor to consider to write efficient software

memory speed vs CPU speed

- modern CPUs (and GPUs) work at frequency of the order of the GHz
  - datacentre CPUs: 2 GHz – 4 GHz
  - datacentre GPUs: 1 GHz – 2 GHz
- system memory is significantly slower
  - with a latency of 200 ns, a CPU can perform 400 operations while waiting for data to arrive !

lower latency
higher bandwidth

memory	latency	bandwidth	capacity	cost
L1 cache	2 ns	100 TB/s	64 kB / core	
L2 cache	6 ns	50 TB/s	512 kB / core	
L3 cache	20 ns	(?) 10 TB/s	4 MB / core	1-2 $/MB
HBM RAM	200 ns	2 TB/s	up to 80 GB / device	20-100 $/GB
DDR RAM	200 ns	20-200 GB/s	up to 64 GB / core	3-4 $/GB
SSD	50-100 us	5 GB/s	30 TB / drive	100-200 $/TB
HDD	2 ms	300 MB/s	30 TB / drive	10-20 $/TB

lower cost
higher capacity

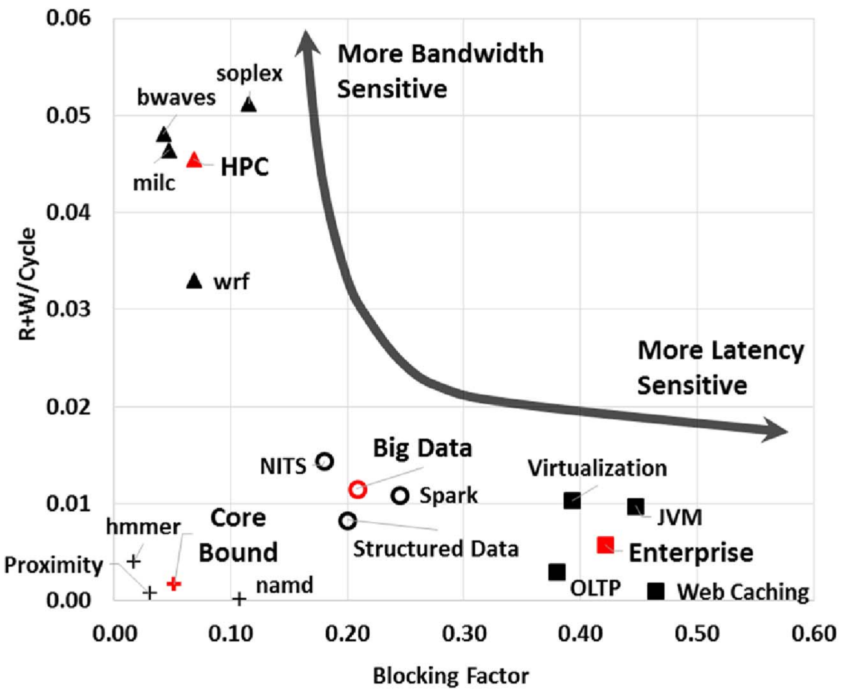
based on the performance of an AMD Rome EPYC CPU, NVIDIA A100 GPU, and datacentre-grade SSDs and HDDs

impact of memory latency

- can we write software that runs efficiently despite the relatively high memory latency ?

- yes !
  - from a paper by Intel engineers that analyses the performance of different software packages
  - **Enterprise** software is strongly affected by memory latency
  - **HPC** software is mildly affected by latency, being instead limited by memory bandwidth

- how ?



Quantifying the performance impact of memory latency and bandwidth for big data workloads, *2015 IEEE International Symposium on Workload Characterization*, Russel M. Clapp *et al*, Intel Corporation
DOI:10.13140/RG.2.1.2677.2562

data-oriented design

- Data-oriented design
  - exploit temporal data locality
    - work as much as possible on data that has just been read or written to memory
    - benefit from the data that is “hot” in the processor cache
  - exploit spatial data locality
    - work as much as possible on adjacent data, to benefit from reading whole cache lines
    - avoid pointers to pointers to pointers to …
    - *e.g.* Array of Structures vs Structures of Arrays
  - hide memory latency
    - prefetch data in advance before it needs to be used, and work on previous data in the meanti
    - keep the processor busy while more data is being fetched from memory
    - a common approach on GPUs, more complicated on CPUs
  - avoid dynamic memory allocations
    - when possible – definitely in your hot inner loops
  - avoid costly high level abstraction
- early adopters: video game development

c++ types and memory

size of data types

- size
  - the size of a type is the number of bytes required to store an object of that type
  - the size of a `char`, `std::byte` and `char8_t` is always 1
  - the size of a class type includes any additional padding and alignment requirements
  - the size of a type can be queried with the `sizeof()` operator

type	32-bit mode	64-bit mode
sizeof(char)	1 bytes	1 bytes
sizeof(short)	2 bytes	2 bytes
sizeof(int)	4 bytes	4 bytes
sizeof(long)	4 bytes	8 bytes
sizeof(long long)	8 bytes	8 bytes
sizeof(__int128)	*n/a*	16 bytes
sizeof(float)	4 bytes	4 bytes
sizeof(double)	8 bytes	8 bytes
sizeof(long double)	12 bytes	16 bytes
sizeof(void *)	4 bytes	8 bytes
sizeof(std::vector<int>)	12 bytes	24 bytes

data types: alignment

- alignment
  - the alignment of a type is the number of bytes between successive addresses at which objects of this type can be allocated
    - *e.g.* if a type has an alignment of 4, it can be allocated only every 4 bytes: `0x…00`, `0x…04`, `0x…08`, `0x…0c`, `0x…10`, …
  - the alignment of a class type is the largest of the alignment of its members
    - this guarantees that all data members are properly aligned
  - the alignment of a type can be queried with the `alignof()` operator
  - stricter alignment can be requested with the `alignas()` specifier
  - alignment is always a **power of 2**: 1, 2, 4, 8, 16, …

- `std::max_align_t`
  - a type with an alignment requirement as large as any scalar type
  - `alignof(std::max_align_t)` returns the maximum alignment of any scalar type
  - `alignas(std::max_align_t)` aligns a variable or type to the largest alignment of any scalar type

exercise

- write a simple program that prints the *size* and *alignment* of various
  - integer types: `bool`, `char`, `short`, `int`, `long`, …
  - floating point types: `float`, `double`, `long double`
  - pointers
    - does the size and alignment of a pointer depend on the type it points to ?
  - `std::max_align_t`
  - arrays
    - does the size and alignment of the array depend on the array element type ?
    - does the size of the array include all of its content ?
  - STL containers
    - `std::string`, `std::vector`, *etc.*
    - does the size of the container include all of its content ?
  - user defined structures or classes
    - try mixing types with different sizes and alignments
    - try using the `alignas()` specifier

https://godbolt.org/z/MWGrWqr5h

layout of class data members

- non-static data members are allocated so that the members declared later have higher addresses within a class object
  - up to C++20, the compiler can arrange the `public` and `private` data members in two separate groups
  - this is no longer the case starting from C++23

- additional padding may be necessary to properly align each data member

- **my advice**: group data members based on their size and alignment
  - avoid padding, and reduce the overall object size

- **my advice**: group data member based on their usage
  - if possible, fit data that is used together within a single cache line (usually 64 bytes)

exercise

- how would you declare a class or struct for a Particle with these data member
  - 1 `const std::string` to hold the particle's name;
  - 3 `double`s for the x, y, z velocities
  - 3 `bool`s to mark if there has been a collision along the x, y z directions
  - 1 `float` for the mass
  - 1 `float` for the energy
  - 3 `double`s for the x, y, z coordinates
  - 1 `const int` for the particle's id
  - 1 `static int` to keep track of the total number of objects

  ?

- can you fit all non-`const` data in a single cache line ?

exercise

```cpp
struct BadParticle {
    static int counter_;        // static: not part of the class layout

    double x_, px_;
    bool hit_x_;

    double y_, py_;
    bool hit_y_;

    double z_, pz_;
    bool hit_z_;

    float mass_;
    float energy_;

    const std::string name_;    // const: keep out of the hot data
    const int id_;
};
```

```cpp
struct GoodParticle {
    static int counter_;        // static: not part of the class layout

    double x_, y_, z_;          // non-const data modified together
    double px_, py_, pz_;
    bool hit_x_, hit_y_, hit_z_;

    float mass_;
    float energy_;

    const int id_;
    const std::string name_;    // const: keep out of the hot data
};
```

https://godbolt.org/z/zTP47zbdK

memory primitives

basic memory operations

- allocating and freeing a memory block
  - dealing with alignment
  - C++: memory allocation *vs* object construction
  - C++: constructing an object in place
- filling or clearing a memory block
- copying the content of a memory block
  - C++: trivial, standard-layout, and implicit-lifetime types

memory allocations

- `void* std::malloc(std::size_t size);`
  - allocate a block of memory of at least `size` bytes, with an alignment valid for all scalar types
  - return a pointer without any type information
  - return a null pointer is the allocation failed
  - the memory is **not initialised**, and **no object** is constructed in this memory (...)
  - the memory is not freed automatically
  - useful to get a buffer that will be immediately overwritten, or as a **primitive** for other operations

- `void* std::calloc(std::size_t num, std::size_t size);`
  - similar to `malloc()`
  - allocate a block of memory for at least `num` elements of `size` bytes
  - the memory is **initialised to zeros**
  - this may be more efficient that calling `malloc()` and explicitly zeroing the memory

dealing with alignment

- `malloc()` returns a pointer to a memory block suitably aligned for any scalar types
  - usually, this means the alignment is 8 or 16 bytes
  - can we get memory with a wider alignment ?
  - for example, we may want memory aligned to a cache line size of 64 bytes

- `void* std::aligned_alloc(std::size_t alignment, std::size_t size);`
  - similar to `malloc()`, allocate a block of memory of at least `size` bytes
  - the memory buffer is aligned to at least `alignment` bytes

freeing allocations

- to avoid memory leaks, the memory allocated by `malloc()`, `calloc()` or `aligned_alloc()` must be deallocated with `free()`

- `void std::free(void* ptr);`
  - frees a memory block obtained by `malloc()`, `calloc()` or `aligned_alloc()`
  - the contents of the memory is not erased
  - any objects in the memory are **not destroyed**

memory allocation *vs* object construction

- in C++ creating an object involves two operations
  - allocating some memory
  - constructing an object in this memory

- in some cases we may want to separate these operations, for example…
  - to allocate an object inside some special-purpose memory
  - to dynamically create multiple objects or arrays of objects inside a single memory buffer

- in a similar way, we can separate the destruction and deallocation of an objects:
  - destroying an object in memory
  - deallocating the memory

constructing objects

- `malloc()` and similar functions return raw, uninitialised memory
  - they do not construct any objects
  - how can we construct C++ objects in this raw memory ?

- `T* new(ptr) T{args…};`
  - use memory already allocated at address `ptr`
  - construct an object of type `T` using the constructor `T::T(args…)`

- `T* new(ptr) T[N]{…};`
  - use memory already allocated at address `ptr`
  - construct `N` objects of type `T` using the default constructor or the provided values

- `T* std::construct_at(T* ptr, args…);`
  - same as `T* new(ptr) T{args…};`
  - requires C++20

destroying objects

- before deallocating (or reusing) some memory, we must destroy the objects that we have created there

- `std::destroy_at(T* ptr);`
  - calls the destructor of the object of type `T` at the memory address `ptr`
  - equivalent to `ptr->~T()`

- `std::destroy_n(T* ptr, std::size_t n);`
  - calls the destructor of the n objects of type `T` starting at the memory address `ptr`
  - equivalent to `for (; n > 0; ++ptr, --n) ptr->~T()`
  - (actually, this function takes an Iterator, not a pointer)

- `std::destroy(T* first, T* last);`
  - calls the destructor of the objects of type `T` in the range `[first, last)`

filling and copying memory

- `void* std::memset(void* dest, int ch, std::size_t count);`
  - writes the *byte value* of `ch` to count bytes starting at the address `dest`
  - take care not to overflow the buffer!

- `void* std::memcpy(void* dest, const void* src, std::size_t count);`
  - copies `count` bytes from `src` to `dest`
  - the two buffers *must not* overlap !

- `void* std::memmove(void* dest, const void* src, std::size_t count);`
  - copies `count` bytes from `src` to `dest`
  - the two buffers *may* overlap

“efficient” C++ types

- creating, copying, moving, and destroying C++ objects calls special member functions
  - constructors
  - copy and move constructors
  - copy and move assignments
  - destructor

- exercise
  - what happens if you use `std::memcpy` to make a copy of an `std::string` ?
  - did it really make a copy of the object ?
  - if you modify either of the old or new objects, what happens to the other one ?
  - what is happening ?

“efficient” C++ types

- creating, copying, moving, and destroying C++ objects calls special member functions
  - constructors
  - copy and move constructors
  - copy and move assignments
  - destructor

- this is extremely useful to guarantee the correctness of the application

- but …
  - sometimes we may want to avoid to achieve higher efficiency
  - sometimes it may not be possible to call these special functions (*e.g.* copy an object to GPU memory)

- *trivial* and *implicit-lifetime* types

trivial types

- trivially copyable types
  - have compiler-defined or defaulted copy and move constructors, and destructor
  - have no virtual member functions and no virtual base classes
  - may have different access specifier (`public`, `private`, *etc.*)

- can be copied and destructed without calling any special member functions
  - can be copied with `std::memcpy()` or `std::memmove()`
  - can be implicitly destructed when deallocating memory

- trivial types, in addition
  - have compiler-defined or defaulted default constructor, and no default initialisers

implicit-lifetime types

- *implicit-lifetime* types
  - are scalars or aggregates (arrays or simple class/structs)
  - have a trivial default constructor and destructor
  - have no private or protected (non-static) data members and base classes
  - have no virtual member functions

- can be implicitly constructed when allocating memory
  - without the need to call any constructor

exercise

- can you declare a class type that is
  - trivially copyable
  - has an implicit lifetime

- suggestion
  - use the `std::is_trivially_copyable_v<T>` and `std::is_implicit_lifetime_v<T>` type traits to check!

optimising memory access

optimising memory access

- efficient data processing depends on
  - data structures
  - data access patterns

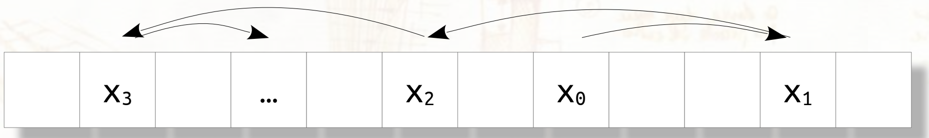
- should be designed together to minimise memory latency and maximise throughput
  - maximise locality
  - minimise wasted memory access

- memory access patterns
  - sequential access ← good on CPUs, because of the serial execution, not so good on GPUs
  - strided access ← good on GPUs, because of the implicit parallelism, not so good on CPUs
  - other special cases for 2D, 3D, *etc.* loops
  - random access ← bad everywhere

memory access patterns

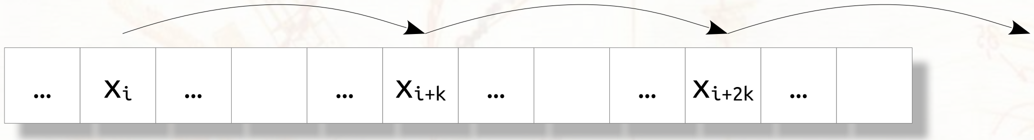
- 

    - **sequential access**: elements are accessed consecutively
    - good use of the prefetcher
    - good cache locality, good utilisation of the memory bandwidth
    - consecutive memory areas can be read for each cycle: **coalesced memory access**

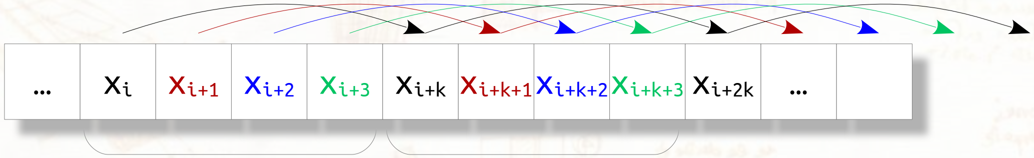
- 

    - **random access**: elements are access in arbitrary order
    - impossible to prefetch next access
    - bad cache locality, bad utilisation of the memory bandwidth
    - do not do this !

memory access patterns

- 

  - **strided access**: elements are accessed at fixed intervals
  - good use of the prefetcher
  - on CPUs: cache locality and memory bandwidth utilisation depend on the stride
    - stride << cache line size: partial usage (1 / stride)
    - stride ≳ cache line size: bad utilisation

- 

  - on GPUs: good use of cache locality and memory bandwidth if the stride equal the grid size
  - consecutive memory areas can be read for each cycle: **coalesced memory access**

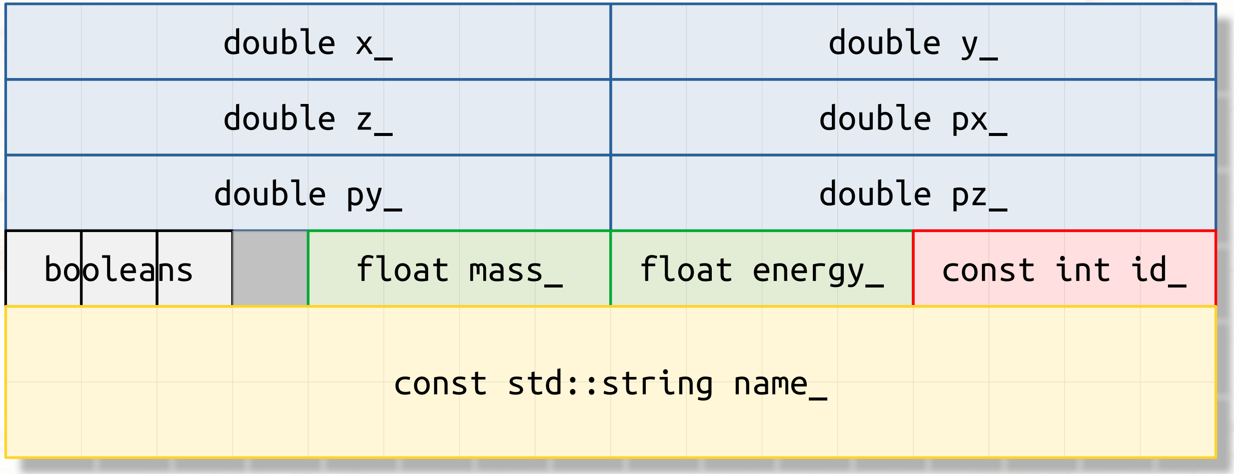
GoodParticle

```
struct GoodParticle {
    static int counter_;

    double x_, y_, z_;
    double px_, py_, pz_;
    bool hit_x_, hit_y_, hit_z_;

    float mass_;
    float energy_;

    const int id_;
    const std::string name_;
};
```



exercise

- write a function that takes as arguments
  - a collection of `GoodParticle` objects (by pointers, iterators, or reference)
  - a boundary: `double x_max`
  - a time interval: `double t`
- and
  - iterates over the collection of `GoodParticle` objects
  - for each object
    - update the position `x = x + px / mass * t`
    - if the updated `x` is less than `0` or greater than `x_max`
      - set `hit_x` to `true` and change the sign of `px`
    - else
      - set `hit_x` to `false`
- what memory access pattern are you using ?

strided access

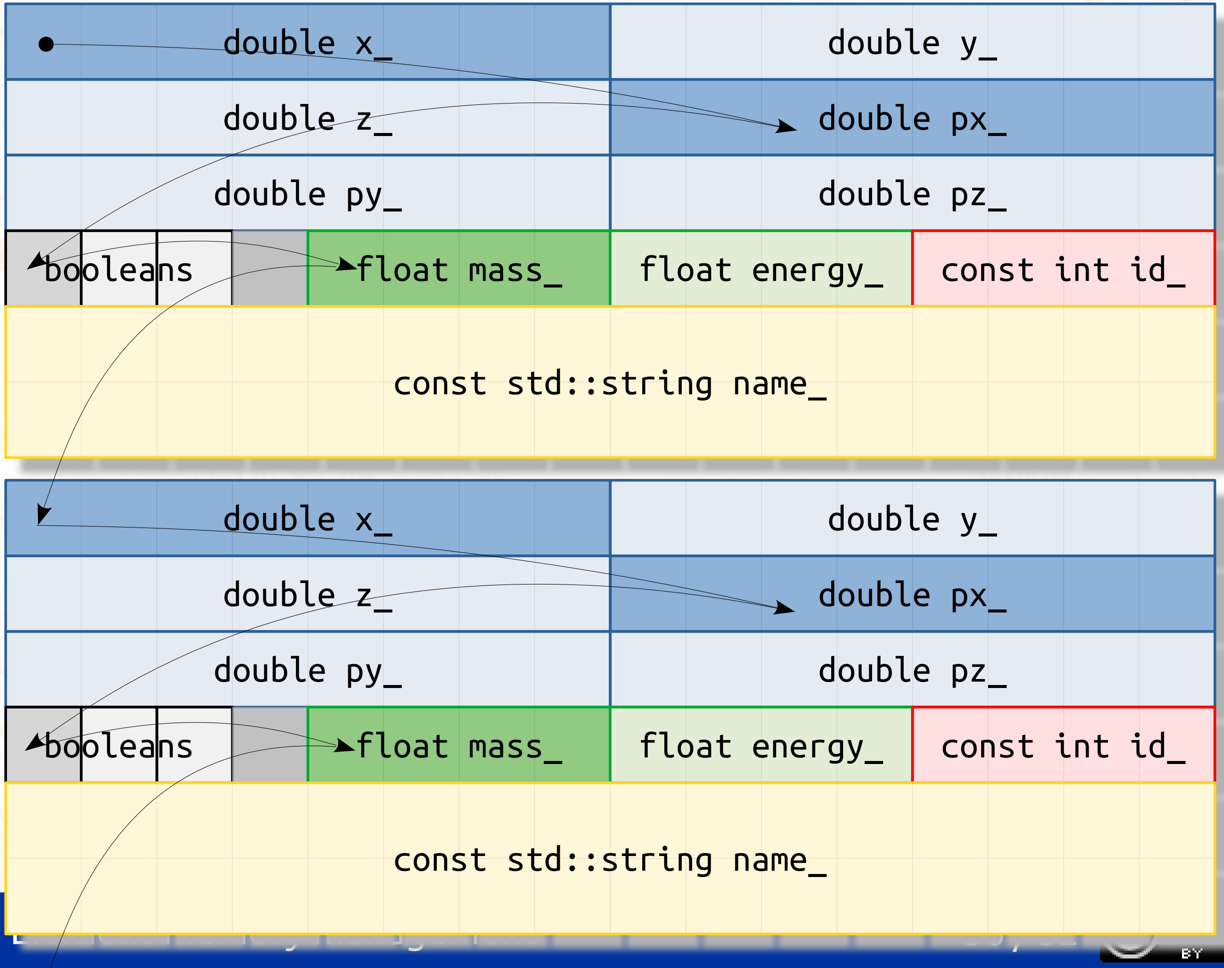
```
struct GoodParticle {
    static int counter_;

    double x_, y_, z_;
    double px_, py_, pz_;
    bool hit_x_, hit_y_, hit_z_;

    float mass_;
    float energy_;

    const int id_;
    const std::string name_;
};

std::vector<GoodParticle> particles;
```



how efficient is it ?

- this is a **strided access**
  - a `GoodParticle` has a size of 96 bytes
  - our example accesses only a few members from each `GoodParticle` object
- while reading consecutive `GoodParticle` objects
  - we read all 96 bytes into the cache
  - we access only 21 bytes
    - `x` (`double`, 8 bytes), `px` (`double`, 8 bytes), `mass` (`float`, 4 bytes) and `x_hit` (`bool`, 1 byte) = 21 bytes
- we actually use less than 25% of the memory that we read !
- how can we change the data structure to improve the locality and bandwidth utilisation ?

Array of Structures

- our current approach uses what is called an Array of Structures (AoS)
- this is a typical pattern used in OO programming
  - define individual, self contained objects
  - allocate as many as needed in an array or vector

- an operation that accesses only a small part of the object is likely to exhibit poor locality
  - leading to a poor use of the cache and of the memory bandwidth

- can we rearrange the data member to improve the locality ?
  - even if we put all the x-related members together, the best use we could achieve is 21 / 64

- different operation likely access different combination of data members
  - unlikely to find a layout that is optimal for all of them

Structure of Arrays

- the problem is inherent to the Array of Structure approach
  - it is due to the encapsulation of the data members for a *single object*
  - we want to process efficiently a *collection of objects*

- we need to design a data structure that is efficient for the whole collection

- Structure of Arrays (SoA)
  - use an array for each data member
  - store the first data member for the whole collection …
  - … the the second data member for the whole collection …
  - … and so on

std::vector<GoodParticle>

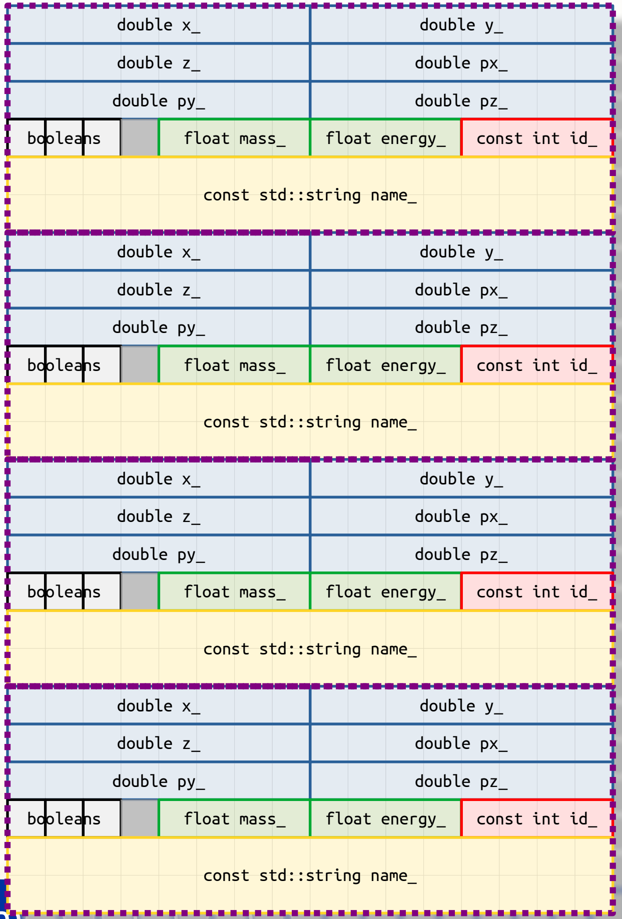
```
struct GoodParticle {
    static int counter_;

    double x_, y_, z_;
    double px_, py_, pz_;
    bool hit_x_, hit_y_, hit_z_;

    float mass_;
    float energy_;

    const int id_;
    const std::string name_;
};

std::vector<GoodParticle> particles;
```



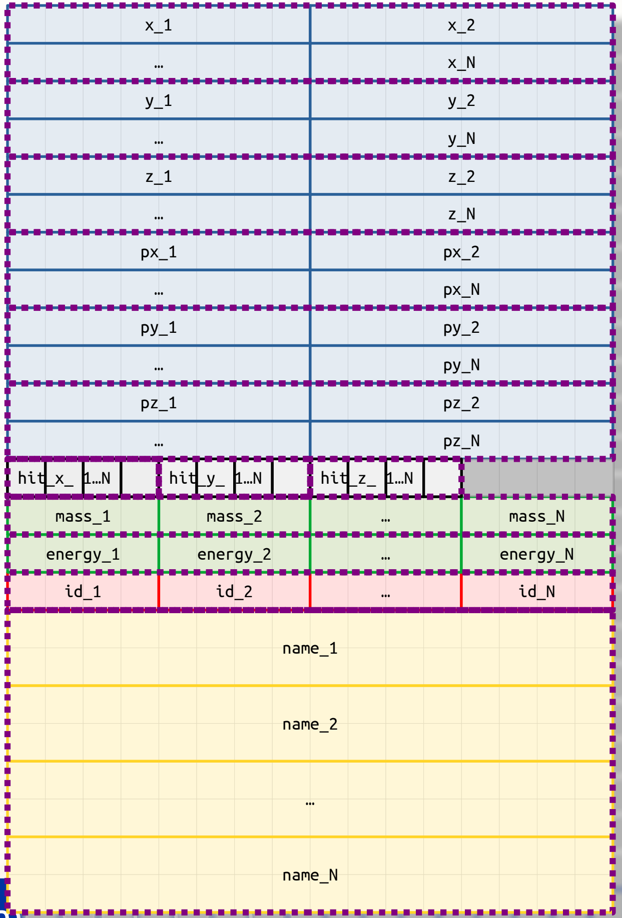
ParticleSoA

```cpp
struct ParticleSoA {
    int counter_;

    std::vector<double> x_, y_, z_;
    std::vector<double> px_, py_, pz_;
    std::vector<bool> hit_x_, hit_y_, hit_z_;

    std::vector<float> mass_;
    std::vector<float> energy_;

    std::vector<int> id_;
    std::vector<std::string> name_;
};
```

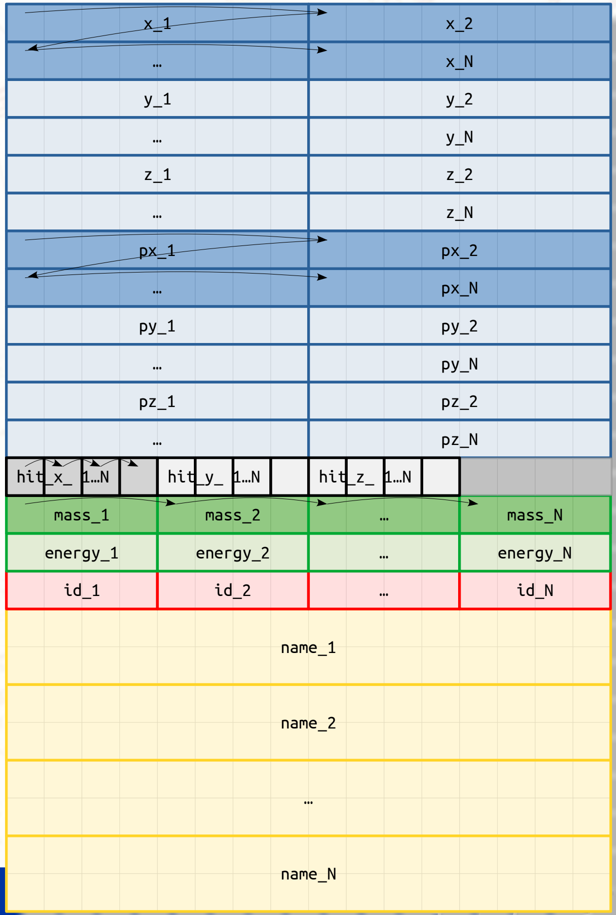


exercise

- write a function that takes as arguments
  - a `ParticleSoA` object (by pointer, or reference)
  - a boundary: `double x_max`
  - a time interval: `double t`

- and
  - update all positions $x_i = x_i + px_i / mass_i * t$
  - if the updated x_i is less than `0` or greater than `x_max`
    - set hit_x_i to `true` and change the sign of px_i
  - else
    - set hit_x_i to `false`

- what memory access pattern are you using ?



looking further

- our `ParticleSoA` uses a separate `std::vector` for each "column"
- this is useful if we later decide to resize the whole SoA to hold a different number of elements
  - resizing requires allocating new memory, making copies of the contents of the vectors, *etc.*
- we can achieve better efficiency if we know the size in advance
  - construct the vectors with the final size
  - replace N vectors with a single memory buffer
- design a data structure that
  - contains a **single memory buffer** and a single **size**
  - contains **N pointers**, one to the beginning of each column
  - has an explicit constructor that takes the size as its only argument, allocates enough memory to hold all columns, and sets each pointer to the start of its column
  - do not forget about the alignment of each column !

memory allocators

alternative allocators

- by default, rely on the system allocator
  - on Linux, this is the glibc memory allocator

- alternative allocators can provide
  - different profiling and debugging tools
  - depending on the workflow: faster execution, reduced memory usage, more stable performance

- TCMalloc
  - Google's fast, multi-threaded, customized implementation of C's malloc() and C++'s operator new

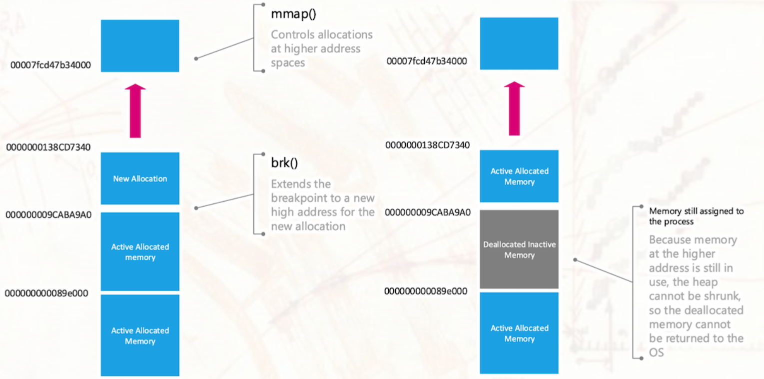
- jemalloc
  - a general purpose malloc() implementation that emphasizes fragmentation avoidance and scalable concurrency support; used by FreeBSD, Facebook, Mozilla Firefox, *etc.*

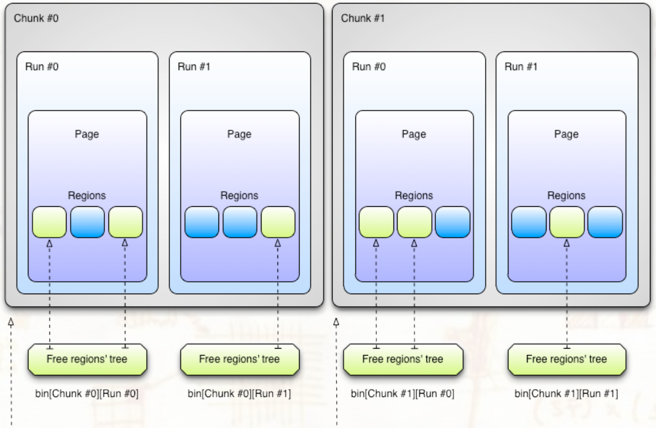
a real world use case

- Taming memory fragmentation in Venice with Jemalloc
  - by Zac Policzer, from the LinkedIn Engineering Blog

- the Linux glibc system allocator exhibits a stack-like pattern
  - memory allocated sits “on top” of the earlier allocations
  - if your program allocates and frees many objects with different lifetimes, the allocator may not be able to return the memory back to the OS

- jemalloc tries very hard to reduce memory fragmentation and return memory back to the operating system
  - reduces “memory hoarding”





Exploiting the jemalloc Memory Allocator: Owning Firefox’s Heap
by Patroklos Argyroudis, Chariton Karamitas, at CENSUS Labs

a simple exercise on memory hoarding

- we can find a simple reproducer of the problem, from Zac's blog post
  - https://gist.github.com/ZacAttack/8c67b998c90afdb19c715dfe327112d2#file-heap-fragmentor-cpp
  - can you get it to compile and run ?
- how to test with jemalloc
  - link with jemalloc at compile time:

```
g++ -std=c++17 -O2 -g heap-fragmentor.cc -L PATH_TO_JEMALLOC -ljemalloc -o heap-fragmentor
./heap-fragmentor
```

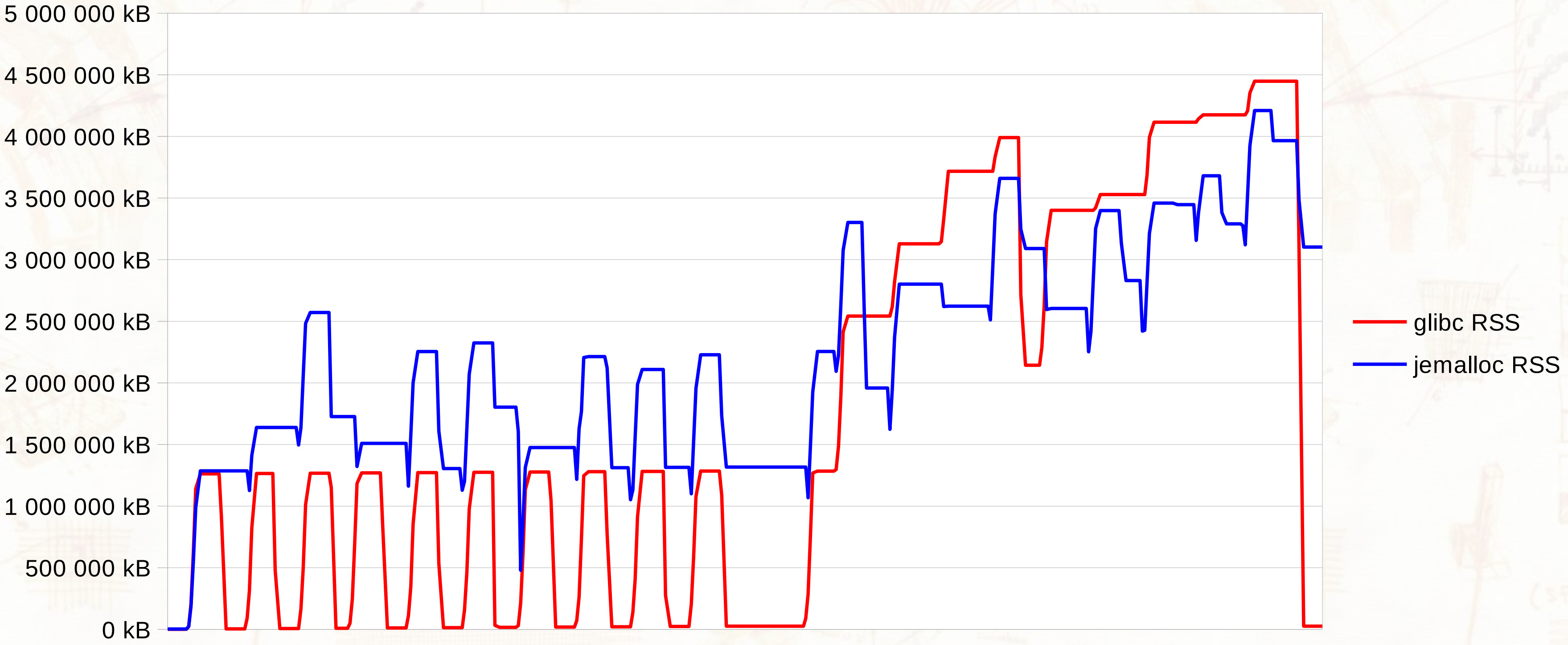
  - use glibc allocator by default, and preload libjemalloc.so at runtime:

```
g++ -std=c++17 -O2 -g heap-fragmentor.cc -o heap-fragmentor
LD_PRELOAD=PATH_TO_JEMALLOC/libjemalloc.so ./heap-fragmentor
```

- what happens ?
- can we make this more realistic ?
  - allocate and free many blocks
  - randomise the allocation sizes

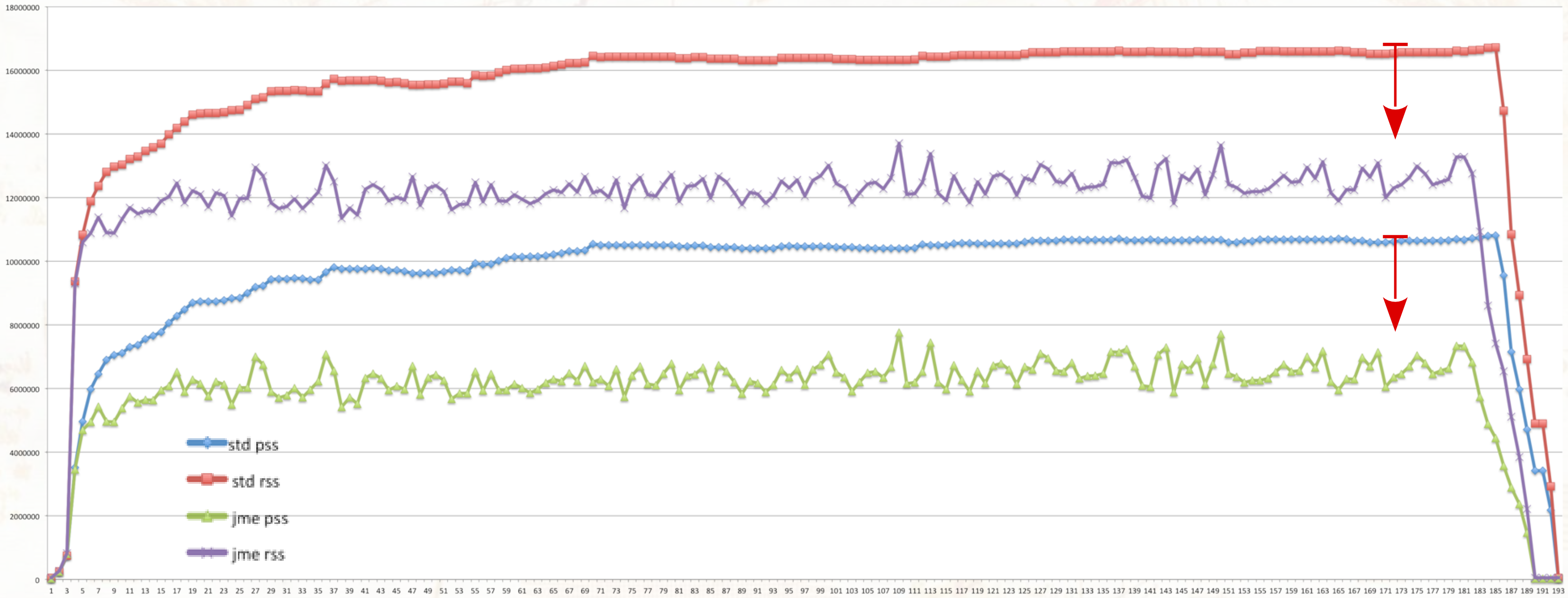
a more complex exercise

- link to heap-fragmentor.cc on GitHub



a scientific use case

- CMSSW, the reconstruction software of the CMS Experiment, introduced multithreading in 2012
- this aggravated the effect of memory allocation patterns for which the glibc system allocator is not optimal, leading to a high utilisation of system memory



reduce peak memory usage by > 20%

added benefit: the program runs faster !

bibliography on allocators

- Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap
  - by Patroklos Argyroudis, Chariton Karamitas, at CENSUS Labs
- Taming memory fragmentation in Venice with Jemalloc
  - by Zac Policzer, from the LinkedIn Engineering Blog
- The effect of switching to TCMalloc on RocksDB memory use
  - by Dmitry Vorobev, from the Cloudflare Blog
- Reducing memory footprint using jemalloc
  - by Vincenzo Innocente, at CERN

(more) questions ?