Efficient Memory Management

Wahid Redjeb[1,2]
wahid.redjeb@cern.ch

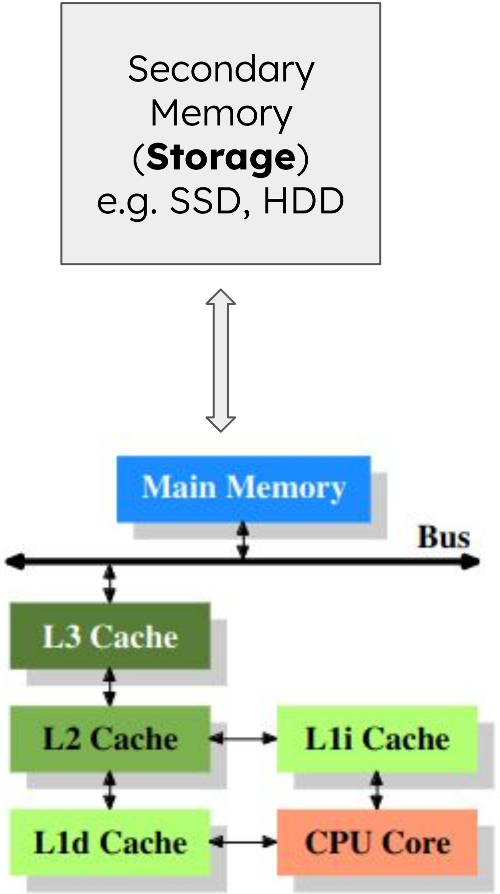
[1]CERN, European Organization for Nuclear Research, Meyrin, Switzerland
[2]RWTH Aachen University, III. Physikalisches Institut A, Aachen, Germany,

What is memory?

- In general, memory refers to the storage a program uses to write and read data
  - RAM
  - GPU memory
  - HBM
  - Disk space (HDD, SDD)
  - Caches

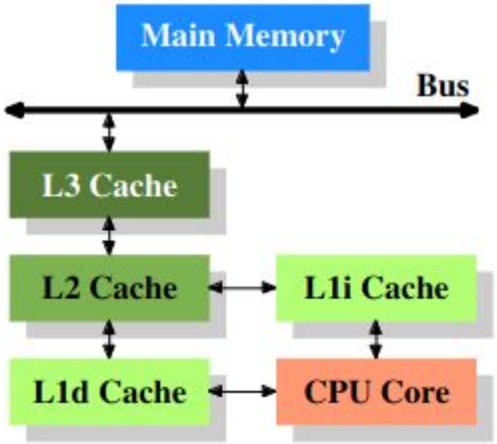
Different types of memory

- Secondary Memory (SSD, HDD) [variable storage]
- Main Memory (RAM) [usually tens of GBs]
- 3 levels of cache
  - Small [32/64kB] separate L1 (I+D) caches for each core.
  - Medium [256kB - 6MB] combined L2 cache, perhaps shared among some cores.
  - Large [4 - 20MB] combined L3 cache shared between all cores



Caches

- CPU looks for data in L1 -> L2 -> L3 -> RAM
- Data area loaded in cache in unit of **cache lines**
  - **Usually 64bytes, but depends on architecture**
- Decision in which hierarchy level some data will stay depends on hardware
  - Memory controllers looks at **memory access patterns**
  - **Cache locality**
  - Cache lines might be promoted or demoted depending on these patterns
- Cache eviction policies
  - LRU (Last-recently-used)
  - FIFO (First-in-First-Out)
  - Random



Different types of memory - Latency

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

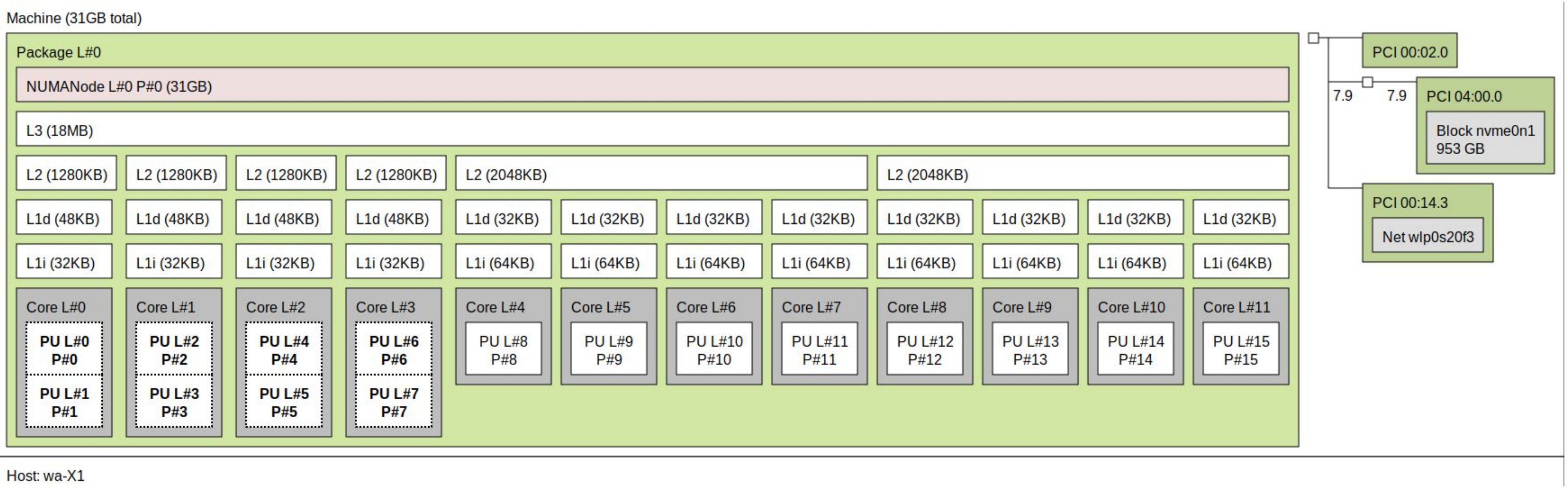
based on the performance of an AMD Rome EPYC CPU, NVIDIA A100 GPU, and datacentre-grade SSDs and HDDs

A.Bocci, CERN

Different types of memory

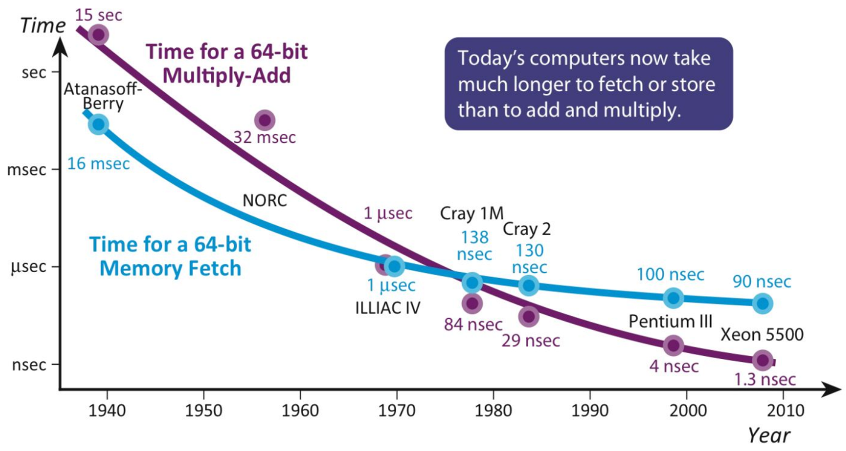
Have a look at your system

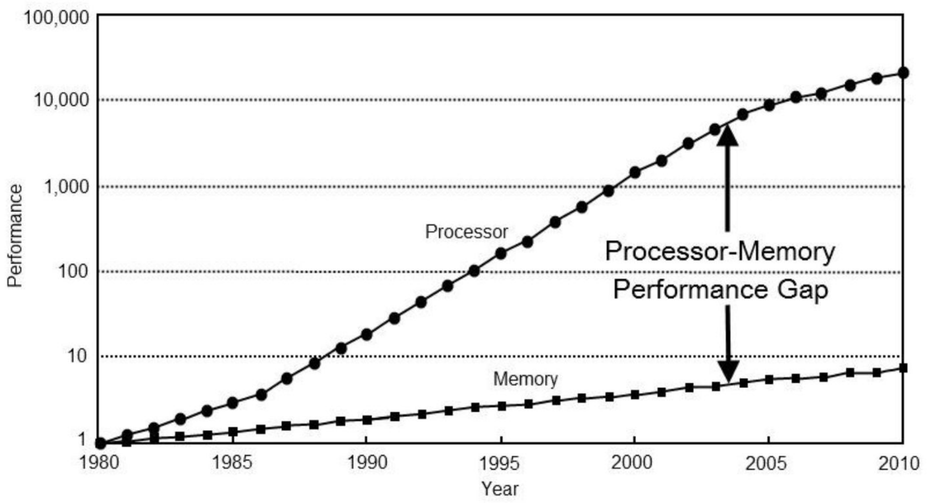
- `lscpu`
- `lstopo`



Why are we interested in memory?

- Most of the memory is *very* **slow** compared to CPU operations





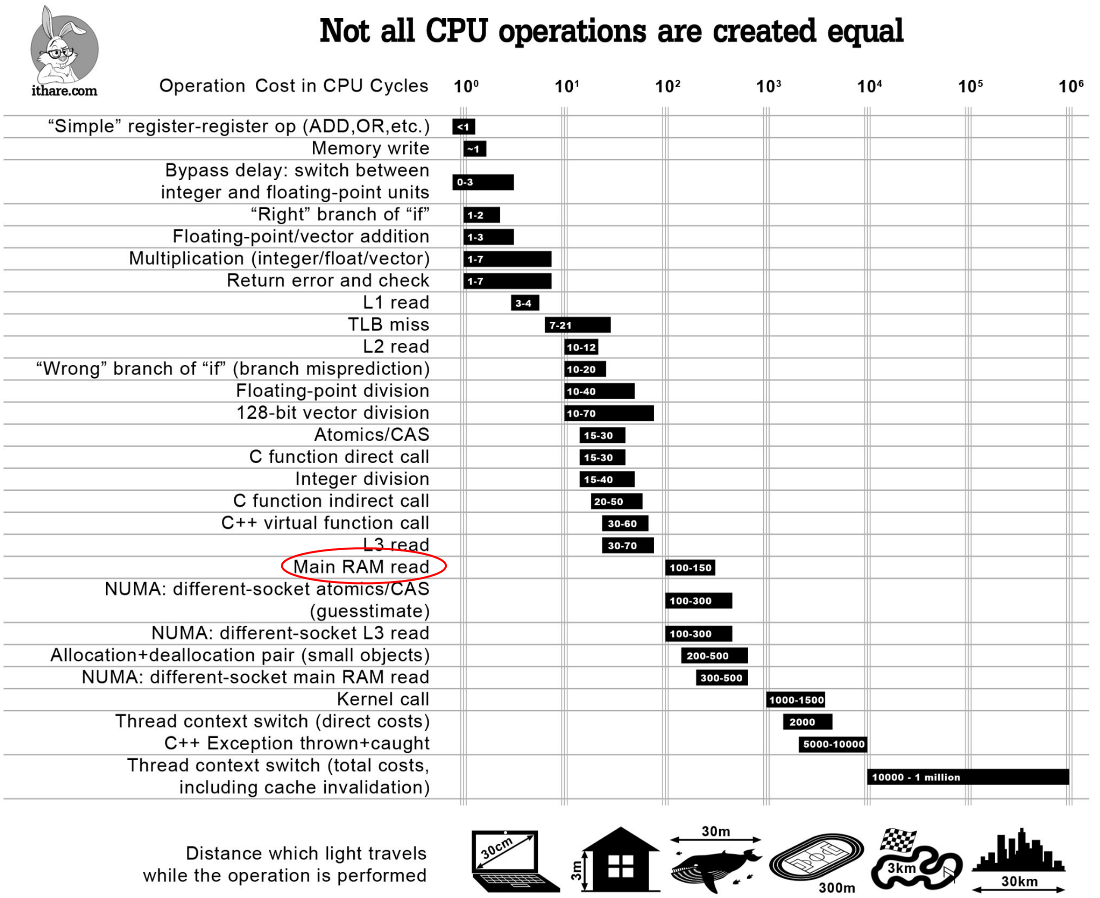
Why are we interested in memory?

Everything here is better than reading from main memory

When writing efficient code, the most important thing to address is memory

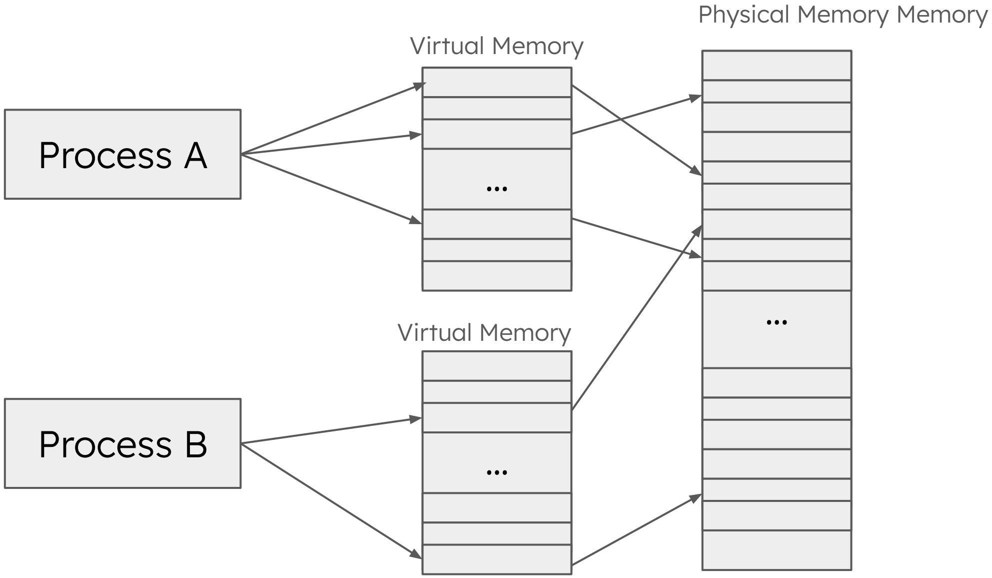
But there's no general rule, the best solution to adopt depends on your **data**

- **Know your data**



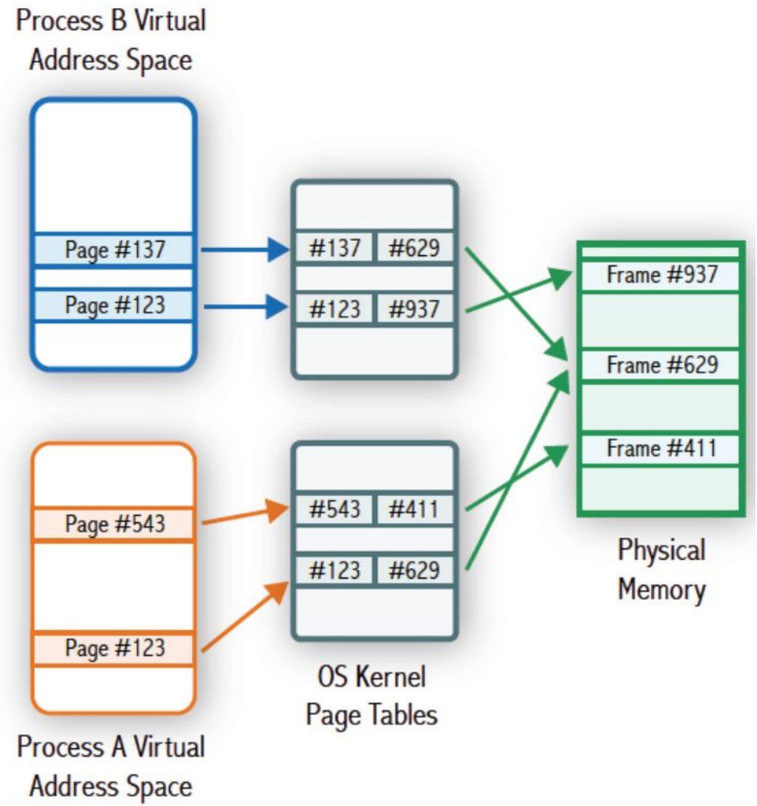
Virtual Memory

- Memory is managed through **virtual memory OS**
- Technique to let each process “think” it is alone in the system
  - The process sees contiguous memory
- Virtual memory maps virtual address to physical addresses in the memory
  - RAM
  - Disk
  - GPU memory



Virtual Memory

- Virtual and real physical memory is divided in **pages**, usually 4kB
- The OS provides the CPU per-process **page tables** to map a virtual address to contiguous physical page frame
  - The Memory Management Unit (hardware) looks at the page table and performs the address translation
- The **Translation Lookaside Buffer (TLB)** helps in performing the translation
  - Cache recent Translation
  - Avoid walking through the entire page tables



Data oriented design

- Data temporal locality
  - Exploit data that has just been read or written to memory
  - Exploit data that is “hot” in the processor cache
- Data spatial locality
  - Fully exploit cache line: work on adjacent data!
  - Avoid pointers chasing if possible
    - Pointers to pointers to pointers …
  - AoS → SoA
- Hide memory latency
  - Prefetch data in advance while working on previous data
  - Keep the processor busy while more data is fetched
  - Common strategy on GPU
- If possible avoid dynamic allocations
  - Remember: understand your data
  - Custom allocators
- Avoid high level abstraction

BASICS

Size of Data Types

Size of a type corresponds to the number of bytes needed to store an object of that type

- Use **sizeof()** operator to get the size of your type
  - Try it yourself with some common types
  - char, int, float, double, int *, std::vector<double>, std::vector<int>
- Define your own Class / Struct with different members and get the size of your class
  - Try to change the order of the members
  - Try to add a bool to your members

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

13 bytes

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

13 bytes →

```
sizeof(MyStruct) -> 24
```

Size of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

13 bytes →

Alignment of data types

- To have a more efficient memory access from the CPU data types are ***aligned***
- *Alignment is an integer value representing the number of bytes between successive addresses at which objects of this type can be allocated.*
    - E.g.: type with alignment of 4 can be allocated only every 4 bytes
- The valid alignment values are **non-negative integral powers of two.**
- The operator **`alignof`**`()` gives you the alignment of a type
- You can request stricter alignment using **`alignas`**`()` specifier
- The alignment of any class object is given by the largest of the alignment of its members

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a																				

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a					b	b	b	b	b	b	b	b								

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a					b	b	b	b	b	b	b	b	c							

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(int) -> 4
alignof(double) -> 8
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
a	a	a	a	:(	:(	:(	:(	b	b	b	b	b	b	b	b	c	:(	:(	:(	:(	:(	:(	:(

Padding required for double alignment

Padding required to ensure correct alignment for arrays

Alignment of Data Types

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

```
alignof(double) -> 8
alignof(int) -> 4
```

```
alignof(bool) -> 1
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
b	b	b	b	b	b	b	b	a	a	a	a	c	:(	:(	:(

Padding required to ensure correct alignment for arrays

`sizeof(MyStruct) = 16`

Alignment of Data Types - Optimize memory design

```
struct MyStruct {
    int a; //4 bytes
    double b; //8 bytes
    bool c; // 1 byte
};
```

Put data members in decreasing size order
Group data members based on their size and alignment

- Dedicate some time to understand if you are introducing padding and if you can avoid it

Group data members based on their usage

- Better to have data members that are used together within a single cache line!
  - Cache line usually are 64bytes.

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
b	b	b	b	b	b	b	b	a	a	a	a	c	:(	:(	:(

Padding required to ensure correct alignment for arrays

```
sizeof(MyStruct) = 16
```

Exercise

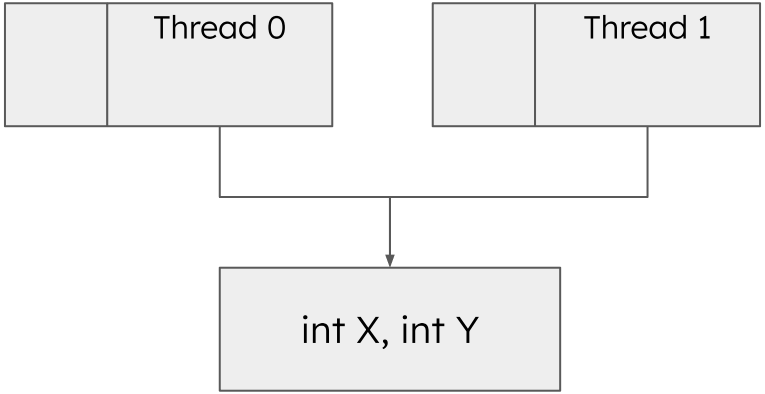
- Create a Class or struct for a Particle with the following members
  - 1 `const std::string` to hold the particle's name;
  - 3 `double`s for the x, y, z velocities
  - 3 `bool`s to mark if there has been a collision along the x, y z directions
  - 1 `float` for the mass
  - 1 `float` for the energy
  - 3 `double`s for the px, py, pz coordinates
  - 1 `const int` for the particle's id
- What is the best order for your members?

(False Sharing)

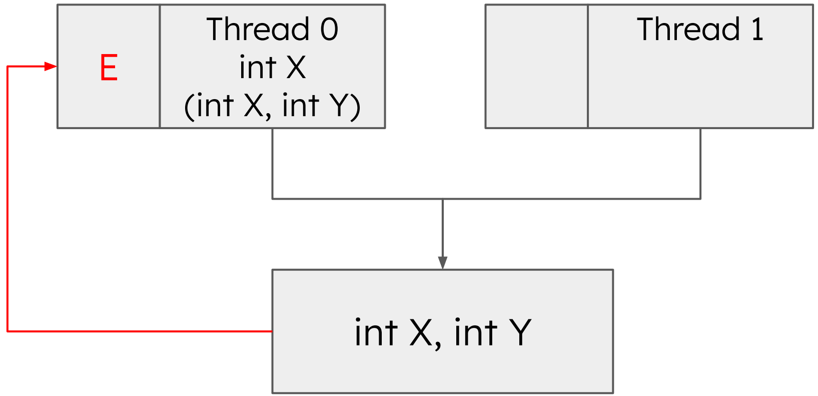
- False sharing is a performance-degrading usage pattern that happens in multi-threaded application
- If two cores are accessing different elements that are in the same cache line
  - Each core has it’s own copy of the cache line
- Core0 reads the value X from the cache line
  - It marks the cache line as **exclusive**
- Core1 reads the value Y from the its copy of the same cache line
  - Both core mark the cache line as **shared**
- Core0 decides to write in address space of X
  - Marks its cache line as **updated**
    - It has to send a message to Core1 saying it has updated the cache line
- Core1 marks its cache line as **invalid**
  - Has to re-read the cache line from main memory
- Core0 has to immediately return the result back to main memory

This process for keeping caches in coherence can be extremely expensive!

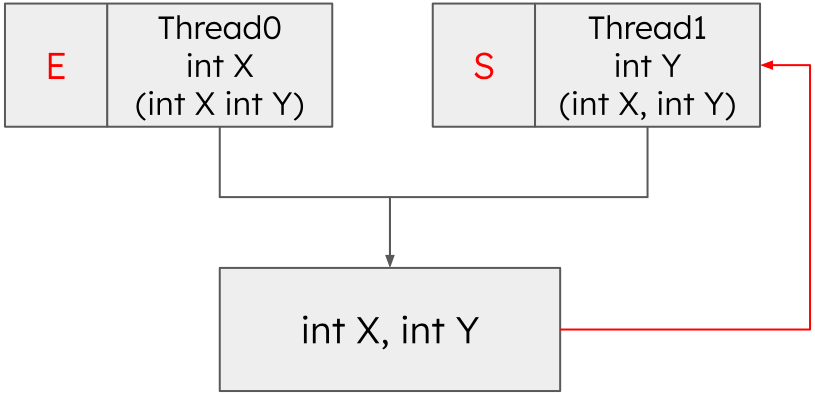
Initial State



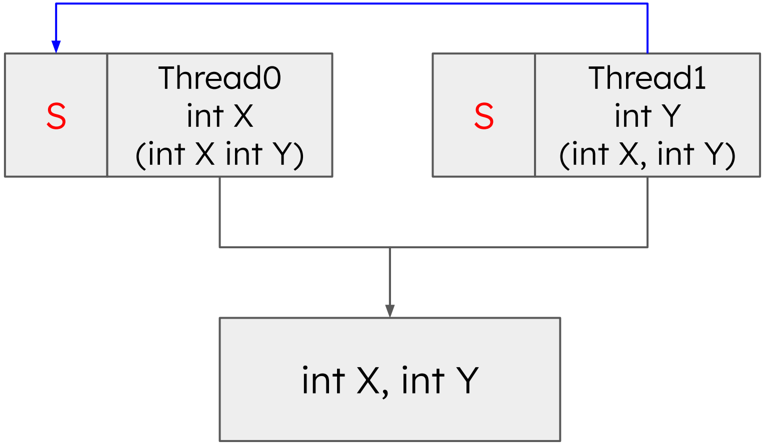
Thread 0 wants to read X
→ Loads cache line (X, Y)
→ Marks cache as Exclusive



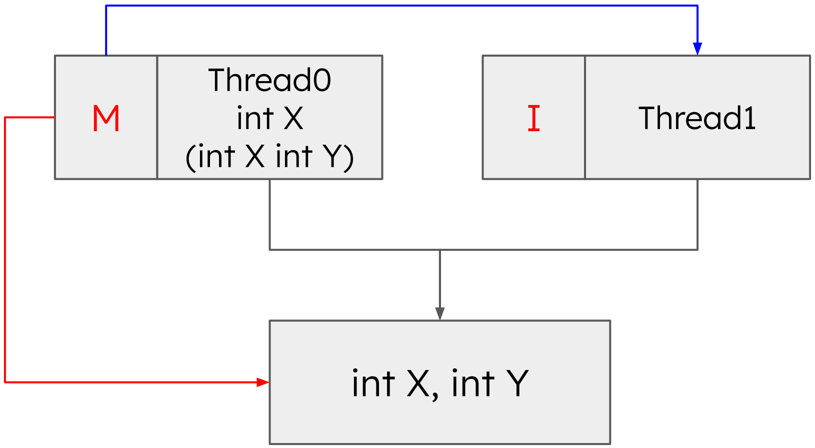
Thread 1 wants to read Y
→ Loads cache line (X, Y)



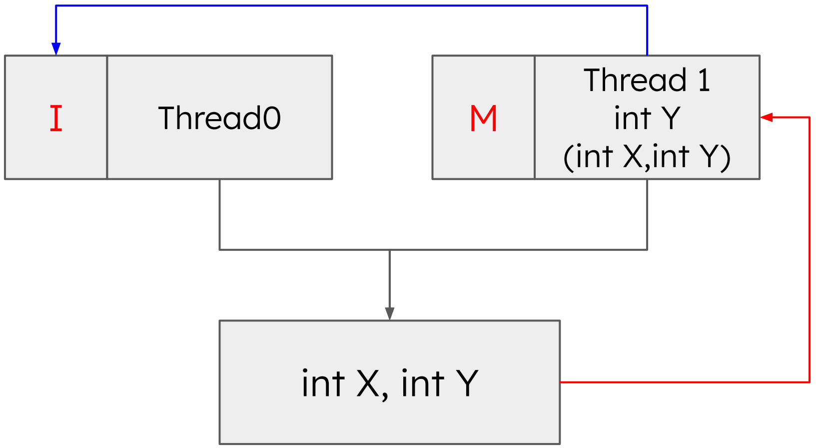
Thread 1 wants to read Y
→ Loads cache line (X, Y)
→ Marks cache as shared and tells thread 0 to mark cache as shared as well



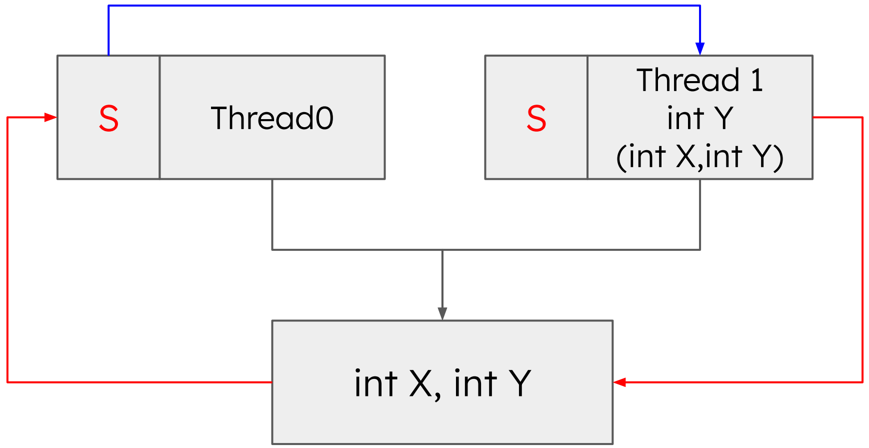
Thread 0 does some operations and then wants to update X
→ Flags the cache line as modified and notify thread 1
→ Thread 0 update the cache line → writes back (coherence write-back)
-> Thread 1 now has to invalidate its cache and throw it away



Thread 1 now wants to update Y
→ Has to reload the cache
→ Flags the cache line as modified and notify thread 0
→ Thread 0 now has to invalidate its cache and throw it away



Thread 0 now wants to update again Y
→ Has to reload the cache
→ But thread 1 needs to update the cache line (coherence write-back)
→ Flag cache as shared



False Sharing

- False sharing is a performance-degrading usage pattern that happens in multi-threaded application
  - Triggers mechanism of cache coherency
    - Caches are continuously getting evicted → low cache usage
      - High cache misses

Small exercise: https://github.com/infn-esc/esc24/tree/main/hands-on/memory/false_sharing

```
g++  false_sharing.cc -pthread
```

How to avoid false sharing?

- If you can, use "local" data
  - Local to each thread, if you need to store some results, store it in a local variable and then gather the results of the multiple threads at the end
  - E.g.: sum of vector entries
    - You split the vector in different blocks and then you assign each block to a different thread
      - Each thread will perform the summation on its own block
      - Don't update an entry of the block with the partial sum, store the partial sum in a local variable —> don't touch the cached data!
    - This will get much more clear with the lectures on parallelism
- Align your data to the cache line size
  - Such that each thread will load a different cache line to avoid interference between the two threads
  - Add padding to your data structure or use `alignas(CACHE_SIZE)`

Memory operations - Allocation

- `void* std::malloc(std::size_t size);`
  - Allocates size bytes of uninitialized storage (on the heap).
  - If successful returns pointer to the beginning of newly allocated memory
  - On failure returns a null pointer
  - Suitable alignment for any scalar type
  - **Nothing is initialized**, just raw memory
  - Requires manual freeing of the memory
- `void* std::calloc(std::size_t num, std::size_t size);`
  - Allocate memory for an array of `num` objects of size `size`
  - Initialized it to all bits zero
- `void* std::aligned_alloc(std::size_t alignment, std::size_t size);`
  - Allocate a block of memory of at least size bytes
  - The memory buffer is aligned `alignment` bytes
    - Useful in SIMD to avoid Cache False Sharing
      - Require memory aligned to a cache line (64bytes usually)

You can ask for the cache size by using

```
#include <new>

auto cls_constructive = std::hardware_constructive_interference_size;
```

There's also.

```
Auto cls_destructive = std::hardware_destructive_interference_size;
```

They are the same on x86 architecture, but different on ARM.

Memory operations - Freeing memory

- `std::free(void* ptr);`
  - Frees allocated memory block by malloc(), calloc() aligned_alloc()
  - The content of the memory is not erased!
    - Any object in the memory is not destroyed!
  - Careful in moving the pointer given by malloc → don't do it!
  - The free operation returns the memory to the system

Memory operations - Constructing objects

- Remember, `std::malloc()`, `std::calloc()`, `std::aligned_alloc()` return raw, uninitialized memory
- `T* new T(args…);`
  - Allocates and creates object `T`
- `T* new(ptr) T{args…};`
  - `ptr` is some memory previously allocated
  - Constructs an object of type `T` using its constructor `T::T(args…)`
    - The object is created in the allocated memory at `ptr`
- `T* new(ptr) T[N]{args…};`
  - `ptr` is some memory previously allocated
  - Constructs `N` object of type `T` using its constructor `T::T(args…)`
    - The object is created in the allocated memory at `ptr`
- `T* new(std::align_val_t(alignment) T{args…}`
  - Constructs an object of type `T` using its constructor `T::T(args…)`
    - Memory is aligned to alignment bytes
    - The object is created in the allocated memory at `ptr`

Memory operations - Destroy objects

- Before freeing the memory (`std::free()`), you have to destroy the created objects

- `std::destroy_at(T* ptr);`
  - Calls destructor of object of type `T` at the memory address `ptr`
  - Equivalent to `ptr->~T();`
- `std::destroy_n(T* ptr, std::size_t n);`
  - Calls destructor of `n` objects of type `T` starting at the memory address `ptr`
- `std::destroy(T* first, T* last);`
  - Calls destructor of the objects of type `T` in the range `[first, last]`
- If you allocated with `T* new(std::align_val_t(alignment)) T{args…}`
  - `delete(T* ptr, std::align_val_t(alignment))`

Memory Leak

As we noticed, both malloc and new requires the programmer to free/destroy the object manually to avoid memory leaks.

Memory Leak: *Failure to release unreachable memory, which can no longer be allocated again by any process during execution of the allocating process.* A memory leak occurs when a program allocates memory on the heap but it fails to deallocate the memory when not needed, losing the reference to the allocated memory (unreachable). Results in an increasing memory usage that slows down the program.

Memory Leak - Example - ASan

```cpp
#include <iostream>

void function() {

        int* ptr = (int*)std::malloc(sizeof(int)*10);

        //forgot to free the memory

}

int main (){

        for(int i = 0; i < 10; i++) {

                function(); // call function

                //don't have any way to reach ptr

                //my reference to allocated memory got lost

        }

        return 0 ;

}
```

```
g++ memory_leak.cc -fsanitize=address

./a.out
```

```
=================================================================
==19662==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 400 byte(s) in 10 object(s) allocated from:
        #0 0x7389410b4887 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
        #1 0x5cd7bc97325e in function() (/home/wa/Documents/Wahid/Bertinoro/a.out+0x125e)
        #2 0x5cd7bc97327f in main (/home/wa/Documents/Wahid/Bertinoro/a.out+0x127f)
        #3 0x738940829d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: 400 byte(s) leaked in 10 allocation(s).
```

Memory Leak - Example - Valgrind

```
#include <iostream>

void function() {

	int* ptr = (int*)std::malloc(sizeof(int)*10);

	//forgot to free the memory

}

int main (){

	for(int i = 0; i < 10; i++) {

		function(); // call function

		//don't have any way to reach ptr

		//my reference to allocated memory got lost

	}

	return 0 ;

}
```

```
g++ memory_leak.cc

valgrind --leak-check=full ./a.out
```

```
==17482== Memcheck, a memory error detector
==17482== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17482== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==17482== Command: ./a.out
==17482==
==17482==
==17482== HEAP SUMMARY:
==17482==     in use at exit: 400 bytes in 10 blocks
==17482==   total heap usage: 11 allocs, 1 frees, 73,104 bytes allocated
==17482==
==17482== 400 bytes in 10 blocks are definitely lost in loss record 1 of 1
==17482==    at 0x4848899: malloc (in usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==17482==    by 0x10919E: function() (in /home/wa/Documents/Wahid/Bertinoro/a.out)
==17482==    by 0x1091BF: main (in /home/wa/Documents/Wahid/Bertinoro/a.out)
==17482==
==17482== LEAK SUMMARY:
==17482==    definitely lost: 400 bytes in 10 blocks
==17482==    indirectly lost: 0 bytes in 0 blocks
==17482==      possibly lost: 0 bytes in 0 blocks
==17482==    still reachable: 0 bytes in 0 blocks
==17482==         suppressed: 0 bytes in 0 blocks
==17482==
```

Smart Pointers

A smart pointer is an object that works like a pointer, but it also manages the lifetime of the object it is pointing to.

Exploits the RAII (Resources acquisition is initialization) idiom:

- Resources are acquired in the constructor
- Resources are released in the destructor

In short: remove the need of manually freeing the allocated memory

In depth explanation this afternoon by Francesco :)

Unique Pointer: Implementation example

```
#include <iostream>
template<typename Pointee>
class UniquePtr {
Pointee* m_p;
public:
explicit UniquePtr(Pointee* p): m_p{p} {} //aquire the resource
~UniquePtr() { delete m_p; } //delete the resource
UniquePtr(UniquePtr const&) = delete; //more in Francesco's lectures
UniquePtr& operator=(UniquePtr const&) = delete; //more in Francesco's lectures
Pointee* operator->() { return m_p; }
Pointee& operator*() { return *m_p; }
};

int main (){
    UniquePtr<int> p{new int{42}};
    std::cout << *p << "\n";
    return 0;
}

>> 42
```

Optimize Memory Access

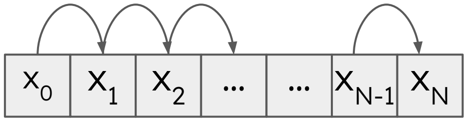
Two main principles:

- Exploit **time locality**
  - If a program accesses one memory address, there is a good chance that it will access the same address again after a short amount of time.
    - E.g loops (variable `sum` continuously updated)
- Exploit **spatial locality**
  - If a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Note: Data Structure and Memory Access are two faces of the same coin. You should design them together!

Sequential Memory Access

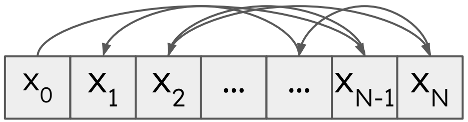
- Consecutive element access
- Good cache locality
- Good memory bandwidth
- Each cycle can read consecutive memory area
  - Cached Memory Access
- Good use of prefetcher



Perfect memory access pattern for CPUs!

Random Memory Access

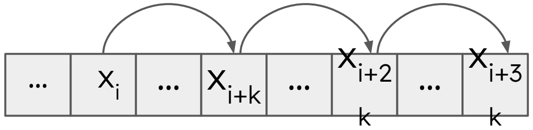
- Elements are accessed in random order
- Cache locality not ensured anymore
- Bad memory bandwidth
- Impossible to prefetch data
- Prefetcher not used



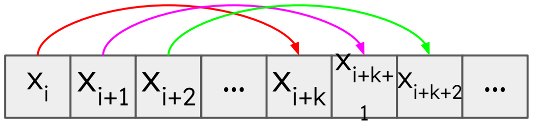
Never use this!

Strided Memory Access

- Elements are accessed at fixed intervals
- Good use of prefetcher
  - Pattern easy to predict



- Very common pattern on GPU
  - Stride size = Grid Size
  - Coalesced memory access
    - Good cache locality and bandwidth

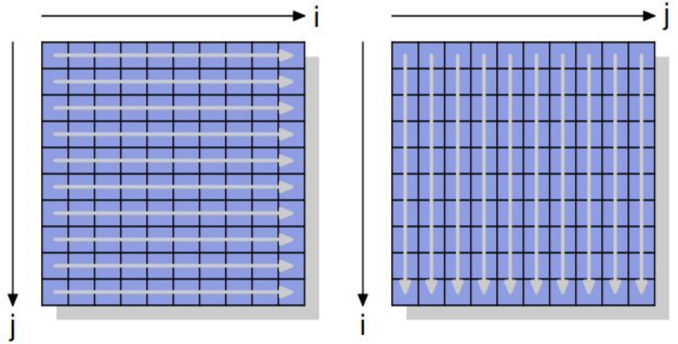


Optimize Memory Access - Example

Matrix multiplication: Given two matrices two matrices A and B with elements a_{ij} and b_{ij} with $0 \le i, j < N$ the product is

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{i(N-1)} b_{(N-1)j}$$

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += A[i][k] * B[k][j]
```



A is accessed sequentially → good!
B is not → everytime I jump to another row. For each iteration in the k-loop a get a cache-hit-miss. → Bad spatial locality!

What Every Programmer Should Know About Memory, Ulrich Drepper

Optimize Memory Access - Example

Let's transpose B, then the matrix multiplication would look like this:

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{jk}^{\mathrm{T}} = a_{i1} b_{j1}^{\mathrm{T}} + a_{i2} b_{j2}^{\mathrm{T}} + \cdots + a_{i(N-1)} b_{j(N-1)}^{\mathrm{T}}$$

```
double Bt[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    Bt[i][j] = B[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += A[i][k] * Bt[j][k]
```

A is accessed sequentially → good!
Bt is accessed sequentially → good!

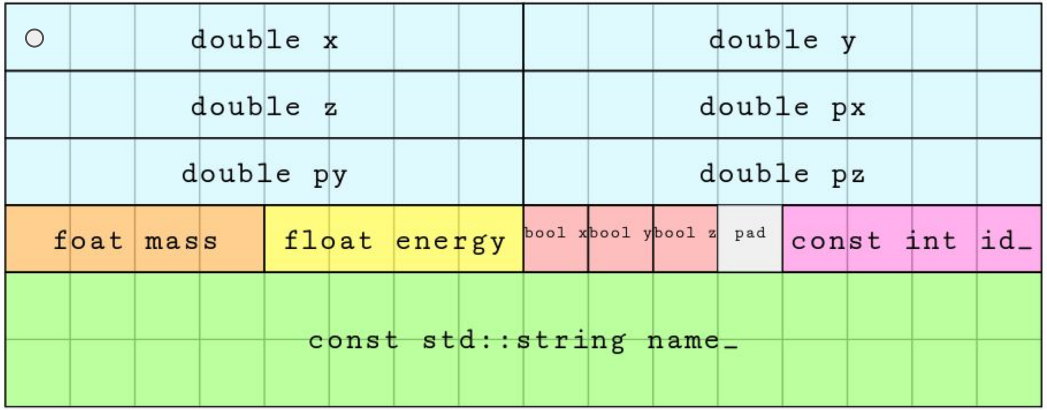
	Original	Transposed
Cycles	16,765,297,870	3,922,373,010
Relative	100%	23.4%

What Every Programmer Should Know About Memory, Ulrich Drepper

Memory Access - Data Structures

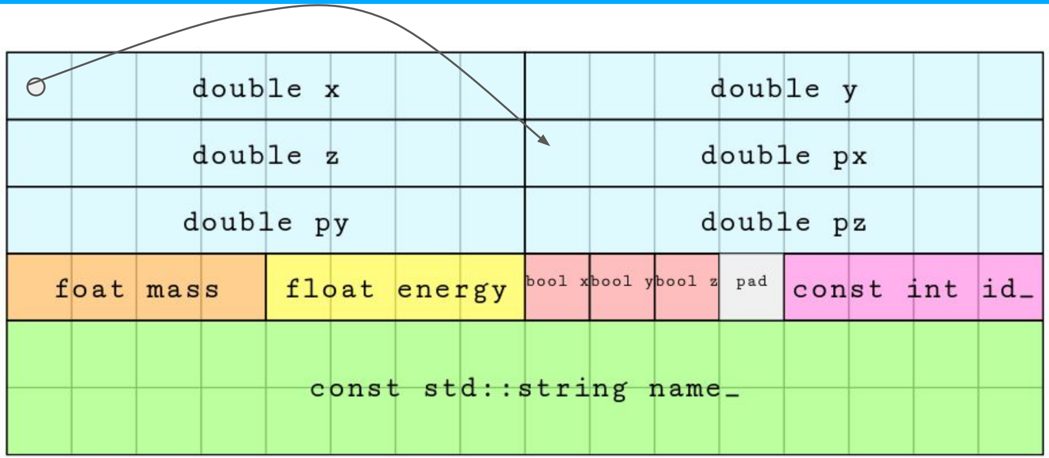
- The way you access memory is not only driven by the algorithm, but it strongly depends on how you designed your datastructure
- Let's investigate our GoodParticle datastructure
  - https://github.com/infn-esc/esc24/tree/main/hands-on/memory/datastructures
- Write a function to initialize a collection of N GoodParticles
  - Assign some value to each member of GoodParticle
  - Pick a x_max value
  - And a time value t
- Write another function that takes as input the collection, and x_max
- Iterate over the elements of this collection and for each element:
  - Update the position x → x = x + px / mass * t
  - If x < 0 or x > x_max → set hit_x to true
    - Else, set it to false and change the sign of px

GoodParticle memory access



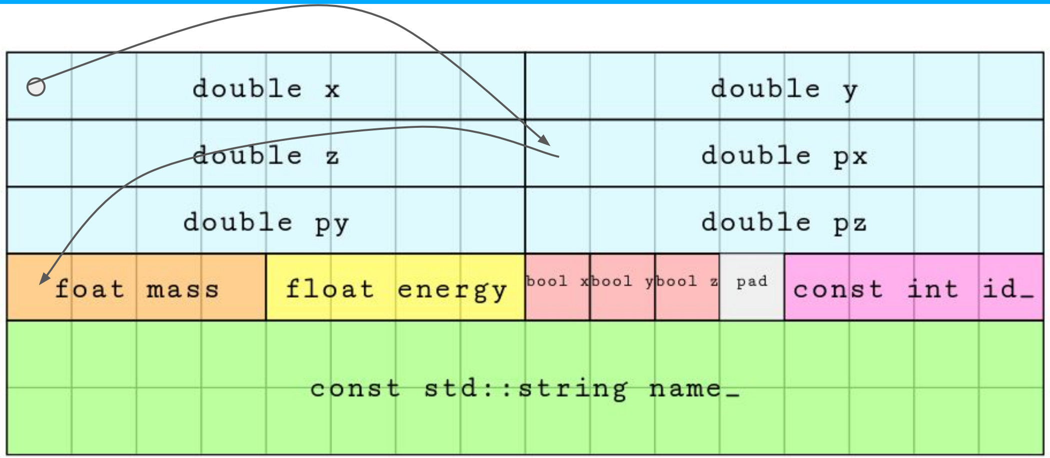
x +=

GoodParticle memory access



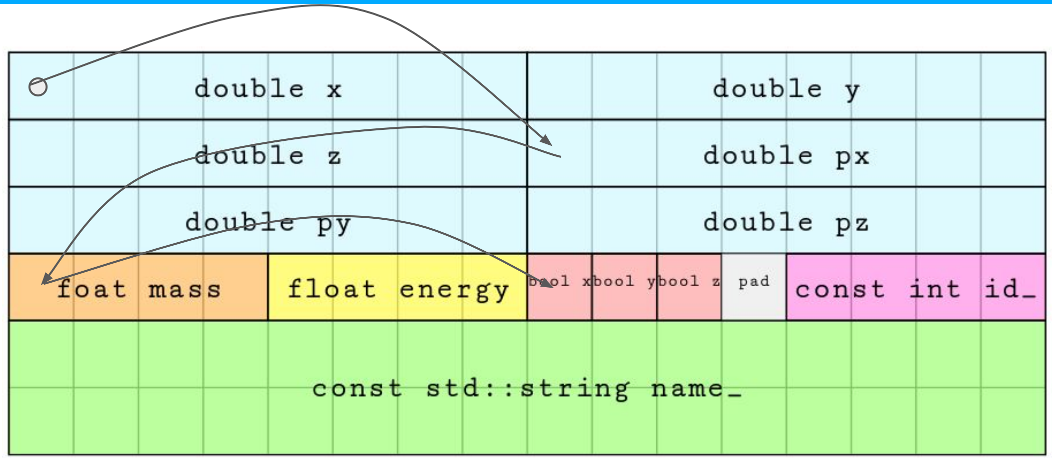
x += px

GoodParticle memory access



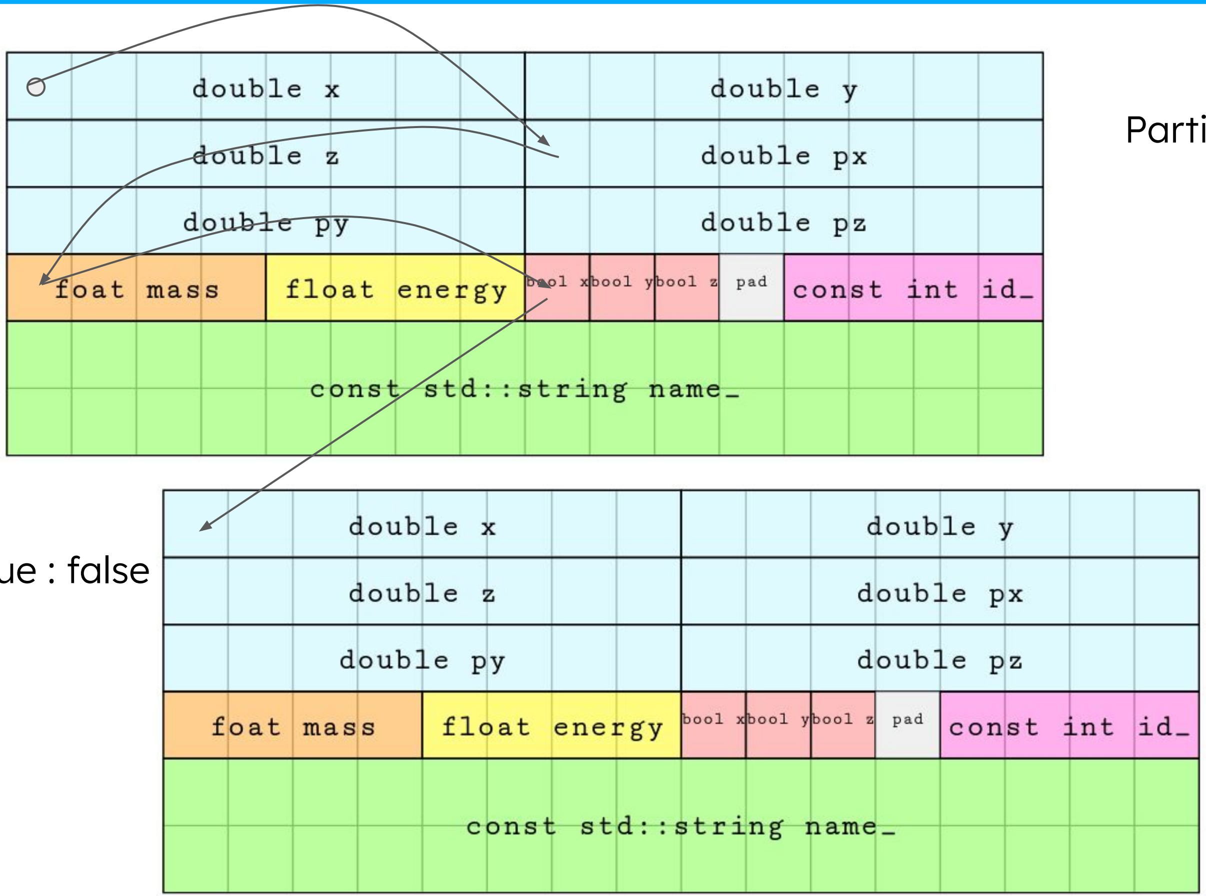
x += px/m * t

GoodParticle memory access



p.x += p.px/p.m * t
p.hit_x = statement? true : false

GoodParticle memory access



Particle 1

Particle 2

p.x += p.px/p.m * t
p.hit_x = statement? true : false

Next iteration

GoodParticle memory access

- Our problem needs only some members of our class GoodParticle
  - We are paying the price of loading the full object for accessing its members
  - `sizeof(GoodParticle) = 96bytes`
  - `sizeof(double`$_x$`) + sizeof(double`$_{px}$`) + sizeof(double`$_{hit\_x}$`) + sizeof(float`$_{mass}$`) = 21bytes`
    - We are using only 22% of what we are reading!
- Our `std::vector<GoodParticle>` is commonly called Array of Struct
  - Very common dastracture coming from Object Oriented Programming (OOP)
    - Self contained objects
      - Bad cache locality and bad memory bandwidth
    - Commonly used because it easy to represent the reality
      - Not so good for manipulating data in some scenario
- In principe we would like to have a data structure that allow us to use only what we need in a specific piece of code

Array of Structs vs Struct of Arrays

```cpp
struct Particle {
    double x;
    double y;
    double z;
    …
};

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

- All data fields for each element are stored together in a contiguous block of memory.
- Cache locality might be loss if not all the elements are used

```cpp
struct ParticleSoA {
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> z;
    …
};

ParticleSoA particles;
```

- Each data field of all elements is stored in separate arrays.
- This layout is beneficial when you need to perform operations on some fields for all elements concurrently

AoS vs SoA

- Take the last exercise
  - Implement an SoA version of GoodParticle
  - Add two more functions, one for initializing the SoA collection and one to perform the operation previously discussed
- Try to time it
  - Try to use compiler optimization (-O1 -O2 -O3)
  - What happens?
- What memory access pattern are we using now?
- Is your data structure interface that different?

AoS vs SoA

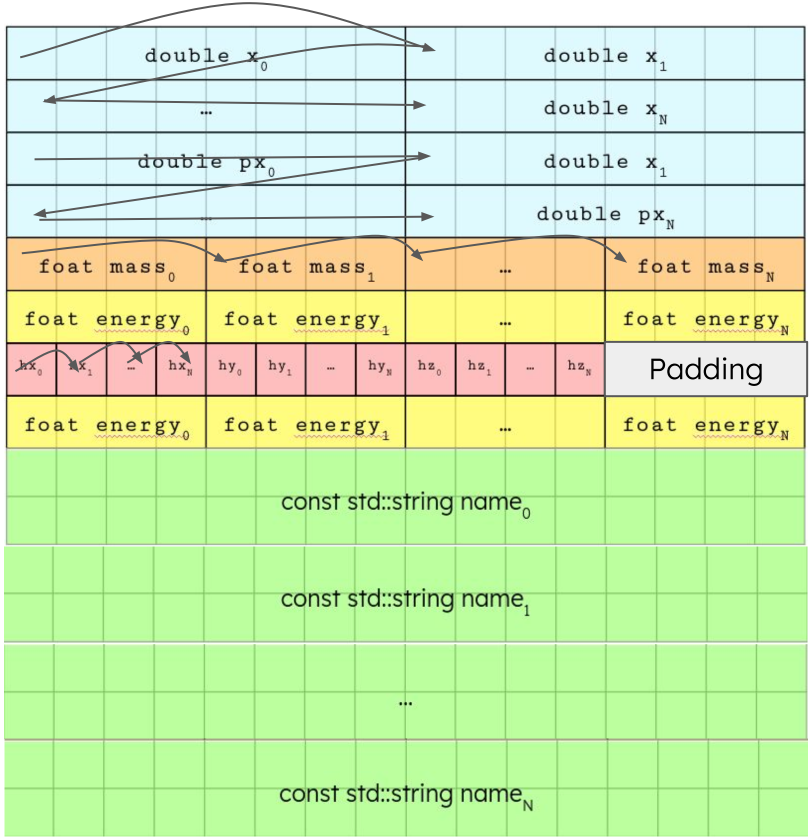
- Sequential access pattern on each member of our object!
- Use only what you need
  - You can pass to your function only the members you are going to use

```
int N = 100;
std::vector<GoodParticleAoS> particles(N);
```

96 bytes * 100 = 9600 bytes
9600 bytes / 64 bytes/cacheline = **150 cache lines**

```
ParticleSoA particles(N);
```

21 bytes * 100 = 2100 bytes
2100 / 64 bytes/cacheline = **33 cache lines!**



More on SoA

- So far our SoA uses std::vector, which is useful to be able to resize our datastructure
- However, resizing is quite expensive
- Better to have fixed sized SoA
    - If you don’t know your exact size, better to put a Max Value
        - Knowing the size (and alignment) at compile time helps the compiler to optimize your code
            - Especially true for vectorization!
- Moreover, you can use single memory buffers to allocate and deallocate memory in one go, or to transfer it to accelerators
    - And you could also reuse the same memory!

Exercise

- Modify your ParticleSoA struct such that:
  - Contains a single memory buffer and a single size
  - Contains M pointers pointing to the beginning of each “column”
  - Explicit constructor that takes the number of particle you want to allocate
    - Allocates the needed memory with a single operation
    - Set each pointer to the beginning of the column
    - Remember alignment!
  - Note → if you allocate the buffer with std::malloc it will give you a void* pointer
    - You can use reinterpret_cast<T*> to cast your pointer to a different type

```
g++ -Wall -Wextra -fsanitize=alignment,address your_program.cc
```

To check if gcc is happy with your alignment!

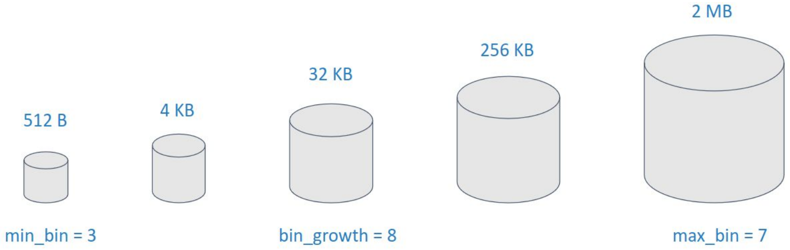
More exercises: Caching Allocator

- Allocating and deallocating can be very expensive
- We can try to reduce the impact of the allocations and deallocation by reusing some allocated memory

- Write a class representing an allocator
  - Should have an allocate(), deallocate() and free() methods
  - Let’s take inspiration from the CUB caching allocator
  - Next slide for more details

More exercises: Caching Allocator

- Idea: reuse memory already preallocated but not used
- Let's decide to only allocate memory in fixed size blocks

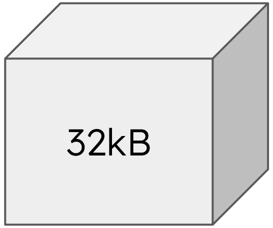


- Everytime I ask for some memory the allocator should decide the minimum block it has to allocate.
  - For example if I ask for 24kB of memory it would allocate 32kB
- Once the memory is not used anymore, we don't release the memory, but instead we keep the memory in a pool
  - If another allocation fits this 32kB of memory, the same block will be reused
  - Otherwise, we create another block

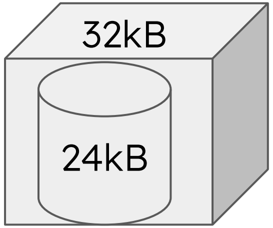
More exercises: Caching Allocator

Ask allocation

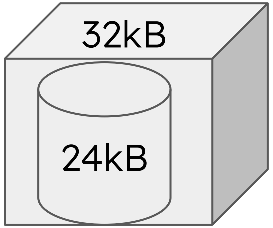
Allocate a big enough block



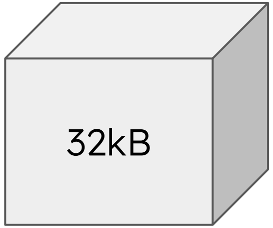
Assign block for request allocation



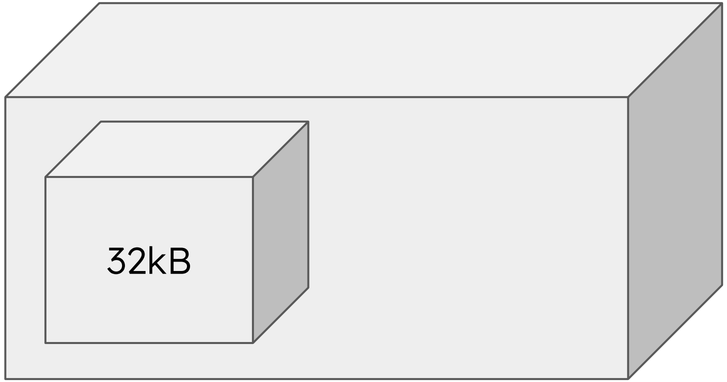
Ask for deallocation



deallocating

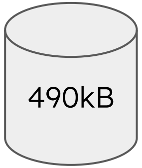


Caching allocated block

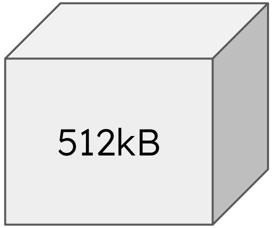


More exercises: Caching Allocator

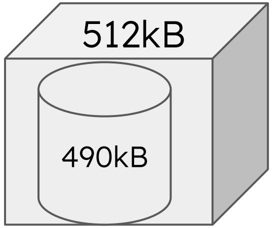
Ask for another allocation



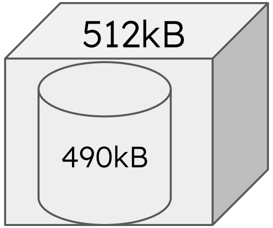
Allocate a big enough block



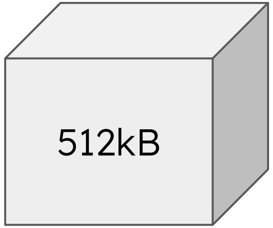
Assign block for request allocation



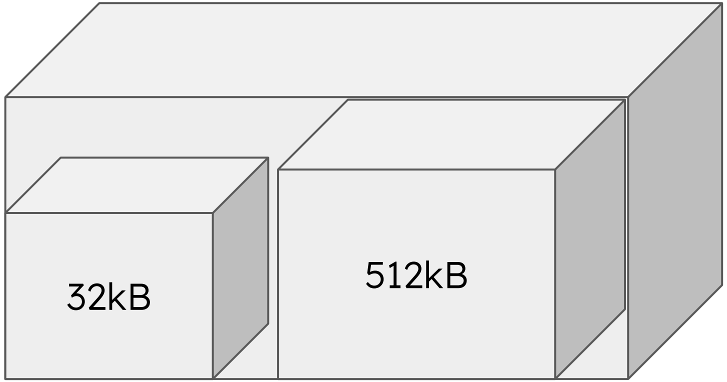
Ask for de allocation



deallocating



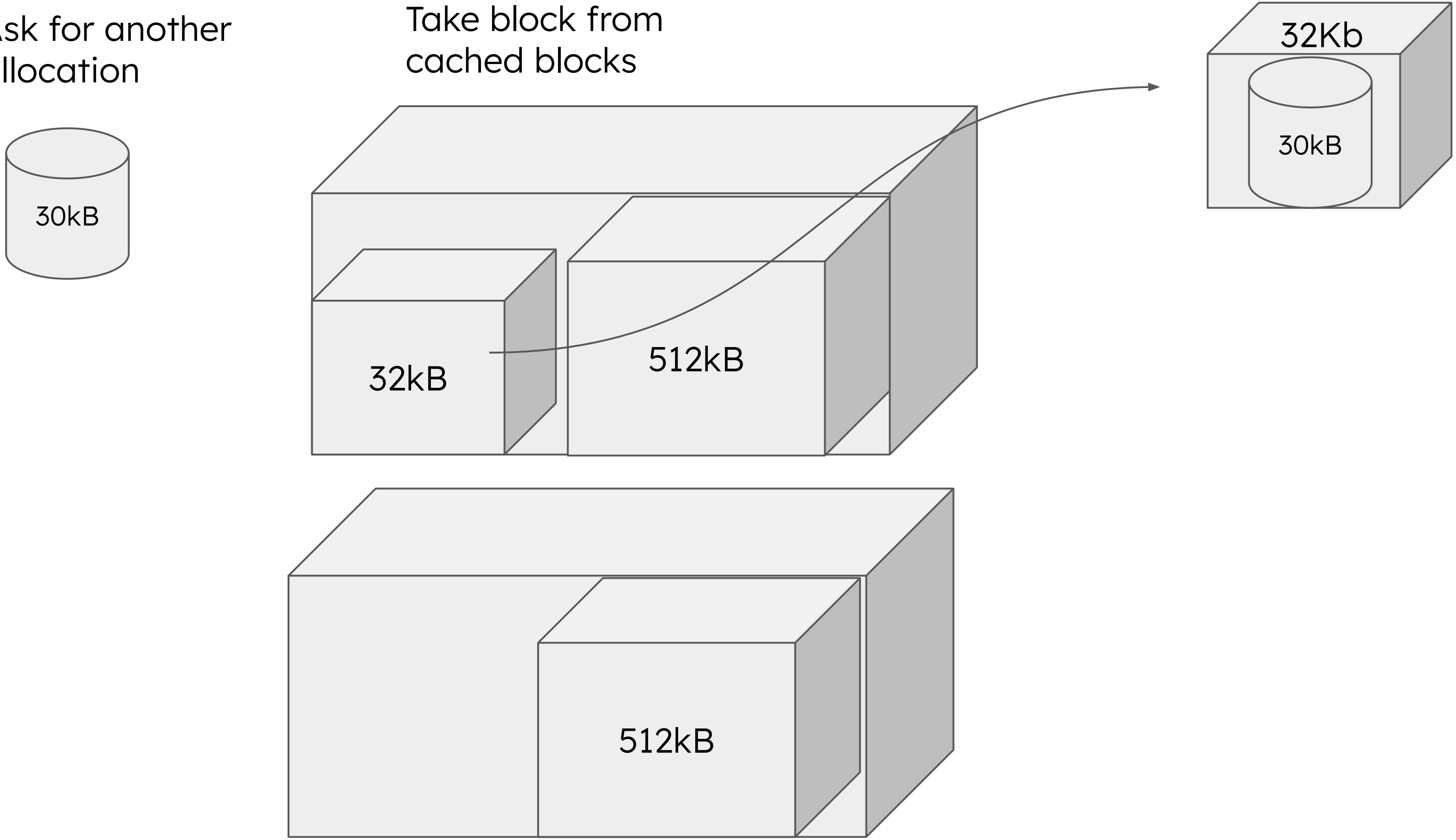
Caching allocated block



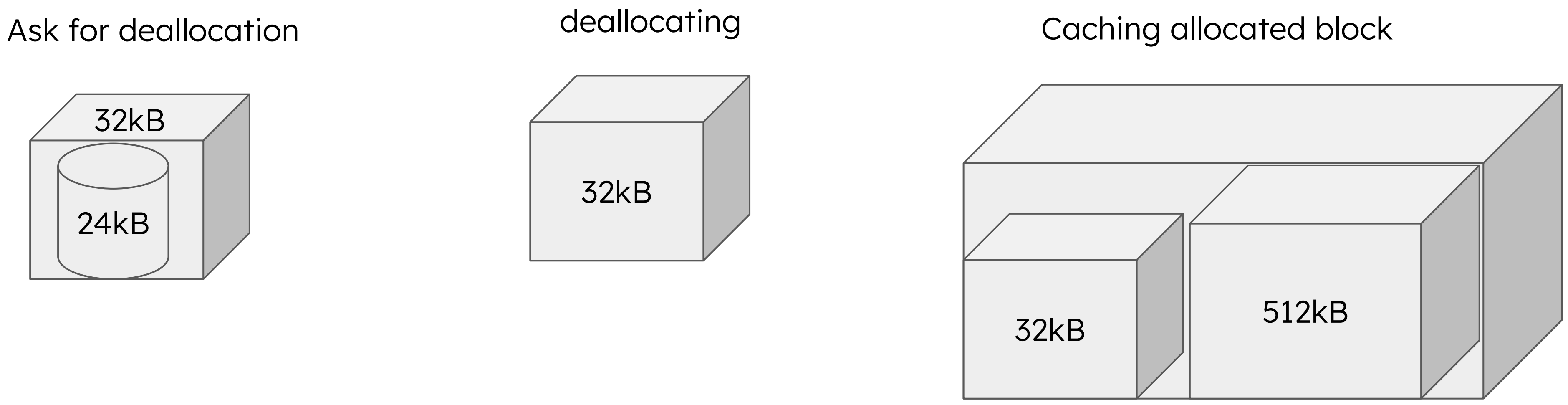
More exercises: Caching Allocator

Ask for another allocation

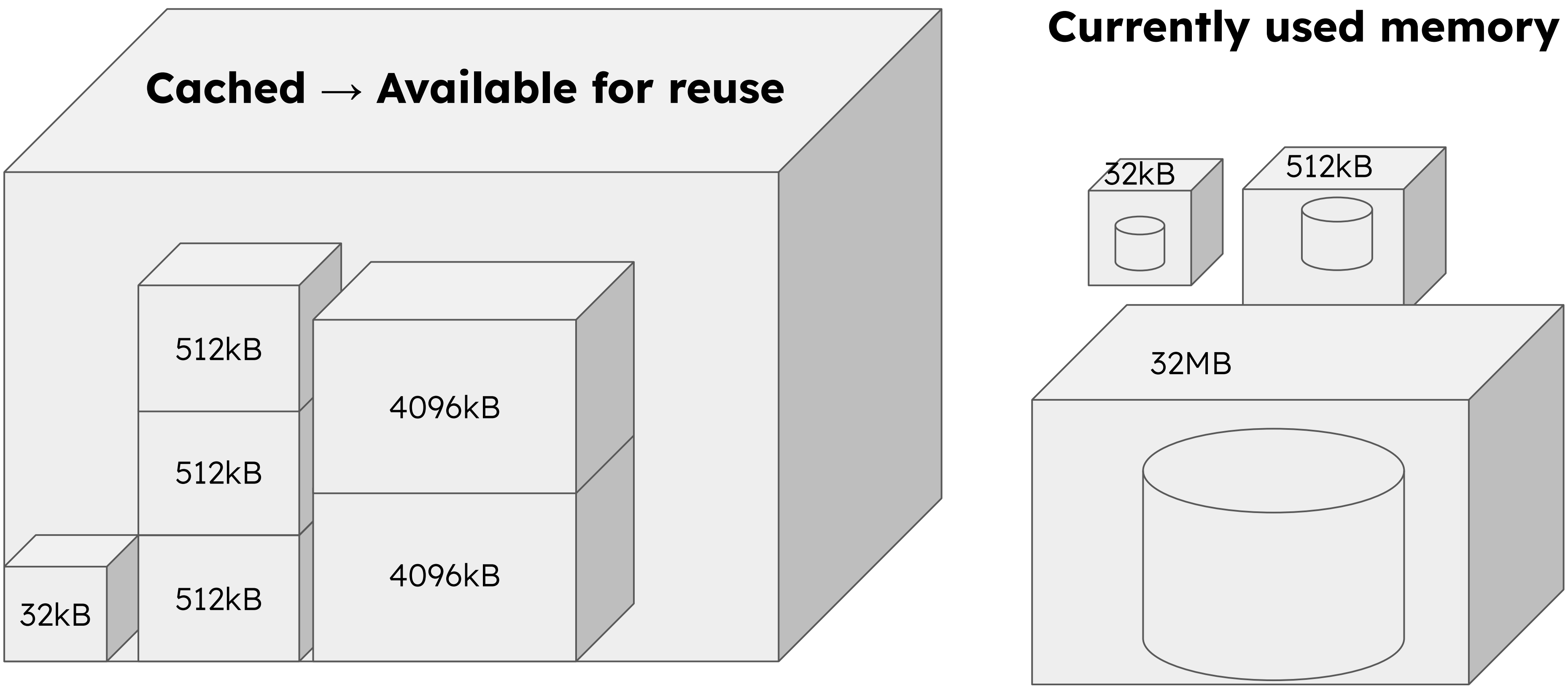
Take block from cached blocks



More exercises: Caching Allocator



More exercises: Caching Allocator



More exercises: Caching Allocator

- At this point I hope the illustration helped …

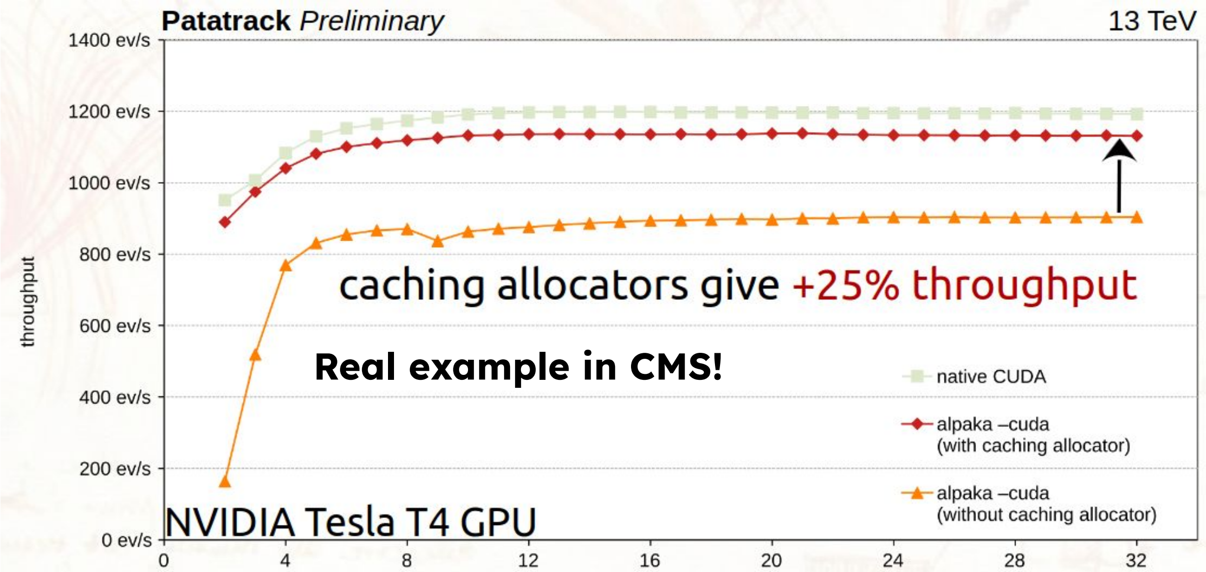
- Write an allocator that allocates blocks of memory fitting the requested size (blocks of memory of power of 2)
  - The allocator should have a min_number_bin and max_number_bin, max allocation size
  - Bin growth (8^bin_number)
  - If requested allocation is bigger than max_number_bin, allocate space normally
  - If requested size is bigger than max_allocation_size, return bad alloc
  - Remember alignment!
  - Use your allocator to allocate members in your ParticleSoA structure!
  - Put some std::cout here and there to see the allocations/deallocations

Caching Allocator - Bonus

- At some point you will know how to deal with multi-threading using TBB
  - That means you will have to deal with race conditions!
- Can you make your allocator thread-safe?
- But possibilities are even more now, for example you can also decide to have an allocator for each thread or for each group of threads!

Caching Allocator - Bonus Bonus

- Stuff becomes more and more complex … now you have a GPU and you are the guru of GPU programming
  - You can manage both CPU and GPU memory with allocators!
  - I am not going to provide a solution for this exercise, but in case you are eager to try, you can have a look at the caching allocator used by CMS



From A.Bocci - ACAT2022

Memory Fragmentation

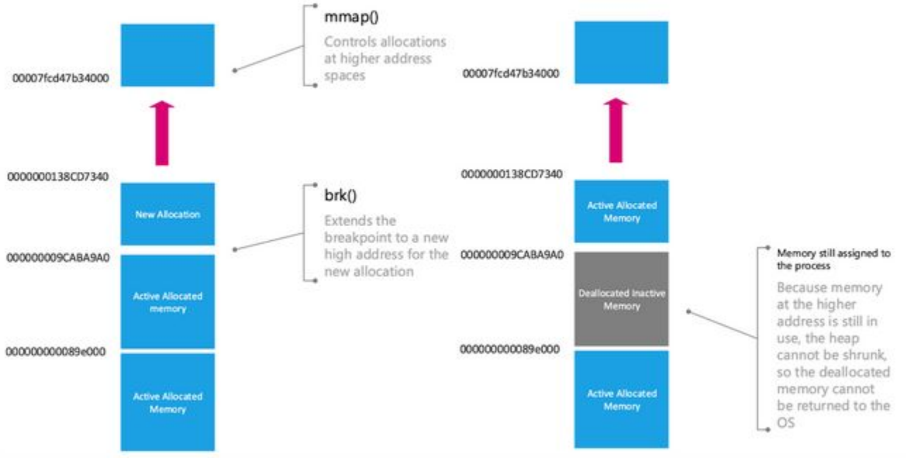
- **UNIX system uses the glibc memory allocator**

Allocate 1kB, 4kB, 2kB

Deallocate 1kB, 4kB

Allocate 2kB

Allocate 4kB → Unable



4kB are available, but not of contiguous memory
→ Memory Fragmentation

Jemalloc and TCMalloc

If your program allocates and deallocates objects with different life times, you get memory fragmentation and the process might not be able to return the memory to the OS

- Alternative allocators
  - Might give you better performance and reduce memory fragmentation
    - But detailed studies are necessary on the full application
- Jemalloc
  - Used by Mozilla Firefox, Facebook, …
  - Tries to avoid memory fragmentation
- TCMalloc
  - Developed by Google
  - Fast C implementation of malloc and new, multithreaded

AddressSanitizer

There are tools for looking at your memory management and for debugging memory problems:

- AddressSanitizer
  - Instrument the program at compile time
    - `g++ program.cc -fsanitize=address`
      - Enable AddressSanitizer → usually is enough
    - Much more options can be enabled https://github.com/google/sanitizers/wiki/addresssanitizerflags
  - Much faster and usually more precise
    - It has all the symbols available at compilation time

Valgrind

There are tools for looking at your memory management and for debugging memory problems:

- Valgrind
  - Instrument the binary → runtime
  - `valgrind -tool=memcheck ./a.out`
    - Valgrind is a tool suite, so you can enable different other tools
      - Cachegrind → performs cache analysis
      - Massif → performs heap analysis
      - Helgrind → thread debugging tools → race conditions
  - Much more slower

Which one to use?

- I would say AddressSanitizer → faster
- But you should use both, especially in case you want to look at performance improvements

Some hints

- Design your datastructures together with your algorithms
  - Don’t try to represent reality with code
    - If some data need different treatment, try to separate them from the rest
- Remember caches
  - E.g.: take some time in writing your for loops
    - Can I design my loop to have a better access pattern?
    - Can I redesign my Datastructure layout to have a better access pattern?
    - Is there anything I can bring out of my loop?
- Try to avoid re-allocation
  - E.g.: std::vector<>.reserve()
  - Custom allocators
- If you have big allocation to do
  - Try to split it!
- Keep in mind false sharing!
- Try to reduce allocation size → reduce your data types (double —> float)
- Avoid copying → pass by ref instead of value

Take Away Message

- Memory is what keeps you away from running code efficiently
- Keep memory always in mind when you are developing your software
- Remember to understand your hardware and map what you are programming on it
- Investigate your data before developing your data structure and try to understand the memory footprint and how to better access the memory
- Profile profile profile
  - perf, **ASan**, **valgrind**, intel VTune

Reference

- Thanks Andrea Bocci for all the inputs and help in preparing the lecture!
- Reducing memory footprint using jemalloc
  - https://twiki.cern.ch/twiki/bin/view/LCG/VIJemalloc
- **What Every Programmer Should Know About Memory**
  - https://akkadia.org/drepper/cpumemory.pdf
- What Programmers Should Know About Memory Allocation - S. Al Bahra, H. Sowa, P. Khuong - CppCon 2019
  - https://www.youtube.com/watch?v=gYfd25Bdmws&t
- CppCon 2014: Mike Acton "Data-Oriented Design and C++"
  - https://www.youtube.com/watch?v=rX0ItVEVjHc&t=2838s
- Computer Architecture A Quantitative Approach - Fifth Edition -J. Hennessy, D. Patterson
- jemalloc

BONUS

Jemalloc example

- Here’s a program with the aim of fragmenting the memory from Zac blog post
    - https://gist.github.com/ZacAttack/8c67b998c90afdb19c715dfe327112d2#file-heap-fragmentor-cpp
- Compile it and try to look at the