UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

C++ Performance

Liam Keegan, SSC

Course Outline

- Performance
  - How algorithms, data structures and hardware affect performance
- Best practices
  - Broadly applicable c++ strategies and idioms for performance
- Other tricks
  - Other c++ tricks, only applicable in certain situations, use with care
- Code examples
  - Optimize some code samples for performance

Performance

Motivation

- We use c++ because we need performance
  - Otherwise we could (should?) use a higher-level language, e.g. Python
- Writing fast code is simple
  - Need to use good ***algorithms*** for ***efficiency*** (do less work)
  - Need to use good ***data structures*** for ***speed*** (do work faster)
- Writing fast code is difficult
  - Often these two needs are in conflict with each other
  - These needs can also conflict with coding styles, for example Object Oriented Programming
  - Computers are complicated: hard to reason about the performance of code
  - The answer to “what is the fastest way to do this?” is nearly always “it depends”
  - Vital to profile code and have performance benchmarks

Algorithms

- Complexity
  - Count operations required to process N items
    - E.g. 2 N + 5 N log(N)
  - For large N, term with highest power of N dominates
  - “Big-O” algorithmic complexity is the highest power of N
    - E.g. O(N log(N))
- Memory
  - Required memory is also counted and expressed in Big-O notation
- Optimal algorithms
  - For many problems, can prove an algorithm is “optimal”
  - This means is has the same Big-O cost as the best possible algorithm

Algorithm goal: reduce complexity

- Generally we want the most efficient algorithm
  - I.e. the one with the smallest Big-O complexity
- Often there a multiple “optimal” algorithms
  - Different pros and cons depending on your data and your hardware
- Although sometimes a less efficient algorithm can be better
  - if memory is a constraint, and there is a less efficient algorithm that requires less memory
  - if the big-O *coefficient* is much smaller
    - E.g. unless N is very large, 1e9 * log(N) is larger than N
  - if it allows for a much faster implementation on your hardware
    - E.g. more fast operations can be better than a few slow operations
  - if it allows for parallelization over threads / cores / nodes
  - etc

Hardware

- CPU gets some data, operates on it:
- Get the data from
  - L1 cache (32kb): 1 ns
  - L2 cache (256kb): 3 ns
  - L3 cache (32mb): 20 ns
  - RAM (32gb): 100 ns
  - SSD (1tb): 10000 ns
- Do the operation
  - CPU core cycle: 1 ns
  - SIMD: apply same instruction to multiple data in the same operation
  - Multi-core: a CPU typically has several cores
- CPU spends a lot of time idle, waiting for data

Cache

- Cache misses are very expensive
  - L1 cache: 1 ns wait + 1 ns work
  - RAM: 100 ns wait + 1 ns work
- Cache lines
  - Cache data is organised in lines of 64 bytes
  - CPU gets the whole cache line when it asks for an item of data
- SIMD
  - A single CPU instruction can operate on multiple items of data
  - Eg AVX512 instructions can operate on all 64 bytes in a single operation
- Pre-fetching
  - CPU will also try to guess what data will be needed next
  - If you are iterating linearly over contiguous data, will nearly always predict correctly

Hardware goal: avoid cache misses

- Spatial locality of data
  - Store data contiguously, and in the order in which it will be accessed
  - Also known as “Data-oriented design”
    - or old-fashioned C-style arrays of data
    - or SoA (struct-of-arrays) vs AoS (arrays-of-struct)
  - Why? So that it is more likely to already be in the cache
- Temporal locality of algorithms
  - When accessing the same data again, do it sooner rather than later
  - Why? So that it is more likely to still be in the cache

Parallelism

- Increasingly important aspect of performance
- Different (complementary) levels of parallelism
  - Within a cpu thread: e.g. SIMD instructions
  - Multiple cpu threads: e.g. openmp, tbb, std::thread
  - Multiple cpus / nodes: e.g. MPI
- Higher compute intensity makes cache even more relevant
- Offloading to GPU
  - Thousands of threads
  - Memory bandwidth even more of an issue

General Performance strategy

- For large problems
  - We care about the efficiency of our ***algorithms***
  - We want to do less work
- For small problems
  - We care about cache friendly ***data structures***
  - We want to do our work fast
- Sometimes a large problem can also be many small problems
  - Use an ***efficient*** algorithm to reduce a large problem to many small problems
  - Use a ***fast*** cache-friendly method to solve each small problem
- Algorithms and data structures are interconnected
  - But helpful to consider their impact separately
  - Helps to understand the tradeoffs you are making with your choices

Best Practices

Performance best practices

- Pretty much always applicable
- Improve performance without affecting readability/maintainability
- For each I'll provide
  - A recommendation of what to do / not to do
  - Some reasons / explanation
  - A godbolt / compiler-explorer link to a code snippet
- We can then discuss
  - Look at the assembly generated by the compiler

Use appropriate data structures

- Whenever possible, use `std::array`
  - Contiguous storage on the stack
  - No heap allocation, cache friendly
- Otherwise use a `std::vector`
  - Contiguous data storage on the heap
  - Cache friendly data structure
- If you have key/value pairs, use a hash map
  - Avoid std::map, std::unordered_map is typically faster
  - Libraries like abseil, folly etc offer faster alternatives
  - Or, if you are mostly reading from a map
    - consider replacing it with a sorted vector
- Avoid `std::list`
  - Not cache friendly, many heap allocations

Consider size and alignment of struct members

- Different types have different sizes and alignment requirements
- Changing the order of members can change the size of a struct!
- https://godbolt.org/z/5M6E3cvfM
- Struct ordering tips
  - Put the largest item first
  - Put the most commonly accessed item first
- Or remove the padding
  - `__attribute__((packed))` for gcc/clang or `#pragma pack` for msvc
  - Note potential unaligned performance issues on older / non-x86 cpus
- Or use a struct of arrays (SoA) instead of an array of structs (AoS)
  - Also known as “data-oriented design”
  - Often better for performance (SIMD, cache, alignment, padding)

Use const and constexpr

- Whenever possible, use `constexpr`
  - For anything that can be evaluated at compile time
  - (note: doesn't mean it necessarily will be evaluated at compile time)
- Otherwise whenever possible, use `const`
  - For anything that will not be modified within that scope
  - (note: for member functions const is really a promise about thread safety)
    - Example with 2 allocations:
      - `std::string s;`
      - `s = "hello";`
    - Using const also encourages us to be more efficient (now 1 allocation):
      - `const std::string s{"hello"};`
    - Examples with and without SSO (Small String Optimization):
      - https://godbolt.org/z/jYMs5nW3v

… except for data members

- Don’t make class data members const
  - `struct Data { const std::string name; };`
  - Implicitly disables moving Data!
  - Resizing a vector of Data objects now involves copies instead of moves
  - https://godbolt.org/z/Ee91eo7WP
- But what if they should be const, i.e. invariant?
  - Make them private and write a public const getter member function
- See C++ Weekly 322 for more related examples

Use initializer lists

- Use initializer lists in constructors
  - `struct Foo { std::string m_name; };`
  - This has two allocations (default construct a string, assign name to it):
    - `Foo::Foo(const std::string& name) {m_name = name;}`
  - This has one allocation (string is directly constructed from name):
    - `Foo::Foo(const std::string& name) : m_name{name} {}`
  - What the compiler sees (but still c++):
    - https://cppinsights.io/s/8282aabe
  - What the compiler generates (assembly):
    - https://godbolt.org/z/4f3PzGc74

Obey the rule of 0

- Ideally don’t declare any constructors, allow compiler to generate defaults
  - If you don’t write any, compiler will generate them all for you.
  - https://cppinsights.io/s/834db41a
- Take care not to accidentally disable some if you do define your own!
  - Once you write a copy/move constructor/assignment op, compiler stops generating the others
  - E.g.1 only define a move constructor, then try to use a copy constructor
    - Compile error: you didn’t define a copy constructor
    - At least it is obvious what went wrong here!
    - https://cppinsights.io/s/2c6a8aee
  - E.g.2 only define a copy constructor, try to use a move constructor
    - Compiles without errors and runs without errors
    - But your “move” constructor is just calling the copy constructor!
    - https://cppinsights.io/s/4b8480ca

Use \n instead of std::endl

- `std::endl` is a newline + a flush
  - This is typically a lot slower than just a newline (‘`\n`’)
  - Only use endl when you actually want to flush
- What is a flush?
  - When you write to a stream (e.g. std::cout) data goes into a buffer
  - A flush is when the buffer is written to the destination
  - Typically this operation has a lot of overhead
  - E.g. for a terminal, writing 1 char or a thousand chars takes about the same time
- https://godbolt.org/z/d6nTv8jKd

Make constructors explicit

- Single argument constructors should be explicit
  - Otherwise there may be unexpected implicit conversions
- The function calculates something from a Foo: f(const Foo& foo)
- What does this do?: f(1)
- If there is a `Foo(int x)` constructor
  - f(1) -> f(Foo(1))
  - Compiles (and creates a temporary, maybe expensive Foo)
- If the constructor is `explicit Foo(int x)`
  - f(1)
  - Compilation error: no known conversion from 'int' to 'const Foo'

Avoid unnecessary copies and moves

- Don’t pass (large) objects by value, pass by reference or pointer
  - “Large” is typically anything bigger than a pointer, e.g. 8 bytes
- But do return objects by value: RVO (guaranteed with c++17)
  - Declare object to be returned at start of function
  - Return it by value at end of function
  - Compiler will construct it in-place at function call site (RVO)
  - Note: a previous “optimization” was to move-return the object
    - This was a performance optimization for older compilers
    - Now this is actually a pessimization
      - forces compiler to do a move instead of constructing in-place
  - https://godbolt.org/z/5oabG4za5

Smart pointers: unique_ptr

- Zero/minimal cost vs raw pointer
  - often compiler can fully optimize it away
- Always preferable to manual memory allocation
- To transfer ownership, pass as an r-value (move it)
- Otherwise, pass by const ref or the pointer returned by `.get()`
- https://godbolt.org/z/vWbKqK3MG

Smart pointers: shared_ptr

- Non-negligible cost vs raw pointer
  - needs thread-safe reference count
- When passing to a non-owning function
  - Pass by const-ref or pass a const ref to the `.get()` pointer
  - This avoids having to increment the reference count
  - But only if you are sure the object will not be destroyed during the function call!
- When the callee should also own the data
  - Pass by value: increments reference count
- Prefer `make_shared` when creating a shared pointer
  - Combines reference count & data into a single memory allocation
  - Otherwise need two allocations, one for ref count, one for data
- https://godbolt.org/z/9j9hrv9PY

Avoid memory (re)-allocations with std::vector

- Allocating memory is expensive
- Re-allocating memory is worse: allocation plus copying/moving all elements
- https://godbolt.org/z/d9EKdvK3e
- If you know an upper bound on the number of elements
  - `reserve` this size upfront: single memory allocation
- If you know the upper bound **and** it is not very large
  - consider using a `std::array` instead of a `std::vector`
- Modify (or clear and re-use) an existing vector rather than create a new one
  - clear doesn't change the capacity, avoids deleting & allocating new memory

Avoid virtual functions

- Run-time polymorphism using virtual functions
  - Is a nice design feature, but comes with a cost at run-time
  - Cost can be significant for small functions
  - Often involves a branch due to having to look up function in v-table
  - Sometimes compiler can optimize away the v-table though
  - Tip: always put `final` on a virtual func that won't be inherited from, helps compiler
- One alternative is compile-time or static polymorphism
  - E.g. Curiously Recurring Template Parameter (CRTP) idiom
  - Allows polymorphism without run-time cost
  - But code is more complicated to understand and maintain
- https://godbolt.org/z/8dfEKxqso

Enable compiler optimizations

- Enable optimizations
  - `-O2, -O3, CMAKE_BUILD_TYPE=Release`
- Enable link-time optimization (LTO)
  - `-flto, -flto=thin`
  - Normal optimizations are done independently for each translation unit (cpp file)
  - LTO sees the whole program, can e.g. inline a function defined in a different cpp file
    - But since LTO sees everything, can be too slow to be useful in a large codebase
  - Thin LTO sees a summary of each unit (just the bits likely to be useful)
    - Much faster than full LTO for a large codebase
    - May miss some optimization opportunities

Alignment of memory

- Can be advantages to aligning a block of memory
  - with the width of SIMD instructions
    - to use SIMD instructions that assume alignment
  - with a cache line or page boundary
    - avoid/reduce splitting the memory over cachelines / pages
- Can use `std::aligned_alloc` to allocate aligned memory
  - Need to manually manage memory, i.e. call `std::free`
- Surprisingly difficult to make a `std::vector` with aligned memory
  - For up to 512-bit can use SIMD intrinsic types, e.g. `std::vector<__m256d>`
    - But scalar operations on underlying types then require casts
  - Can use/write a custom allocator
    - e.g. `boost::alignment::aligned_allocator`

Over-alignment of objects

- Sometimes it can also be helpful to over-align objects
  - e.g. avoid splitting a 48-byte object over 64-byte cache-lines
- To do this use `alignas(64)` to force a struct to be 64-byte aligned
  - Previously when we were aligning the start of a contiguous block of memory
  - Now we are aligning each element, i.e. adding padding between items
- Whether this is an improvement depends on your use case
- If you are iterating over an array of items
  - This probably makes performance worse
  - Padding means less items fit in a cache line
- If you are looking up a single item (likely with a cache miss)
  - Probably makes performance better
  - Avoids a possible second cache miss if the object is split over two cache lines

Use SIMD instructions

- From easier & portable to harder & less portable
  - Let compiler auto-vectorize
    - Simplest option, but doesn't always do a good job
    - E.g. if it can't assume alignment, or prove that two arrays don't alias
    - You can add compiler-specific annotations to help it
      - E.g. gcc: `__restrict__, __builtin_assume_aligned`
  - Use a SIMD library (e.g. Eigen for matrix operations)
    - Often the best option when a suitable library exists
  - Use `omp simd` pragmas
    - Use OpenMP 4.0 pragmas to get the compiler to vectorize
    - Easier to use and more portable than compiler-specific pragmas, but needs a recent compiler
  - Write SIMD intrinsics
    - Hard to write and maintain
    - No guarantee that performance is better than above alternatives
    - Not future-proof: new cpu instructions / updated compiler / library
  - Write inline assembly
    - As above but even harder to write and maintain
- Note: running compiled SIMD code on another CPU can crash with an illegal instruction

Enable SIMD instructions

- Option 1: enable all SIMD instructions (not portable!)
  - `-march=native`
  - But **only** if you are sure you will only run the code on the **same** machine you compile it on!
    - Compiled binary may crash if ran on a different cpu!
- Option 2: enable some SIMD instructions
  - But **only** if you know all CPUs the code will run on support these instructions
  - See next slide for some specific suggestions
- Option 3: runtime dispatch
  - Separately compile different versions of a function with different SIMD instructions
  - At runtime, detect CPU capabilities and choose best available function
  - Complicates code & build, possible runtime overhead from function pointer
  - Ideally use a library that does this rather than doing it yourself

Somewhat portable SIMD (x86 CPUs)

- If CPUs are at least Westmere (less than about 20 years old)
  - `-msse4.2 -maes -mpclmul -mcx16 -mpopcnt`
- At least Haswell (less than about 8 years old)
  - `-march=haswell`
- At least Icelake (less than about 4 years old)
  - `-mavx2 -mfma -mbmi2 -mavx512f`
- If running on a HPC cluster
  - Look up which CPUs the nodes you run on may contain
  - Choose arch/instructions that will work on them all
  - Test with/without AVX512, sometimes it is slower than AVX2!
    - CPU clock speed is reduced when using certain AVX512 instructions
- Don't use `-march=native`

Aliasing

- SIMD and pipelining can be inhibited by aliasing
- Aliasing is if two pointers reference the same data
- Compiler is not allowed to assume that two pointers do not alias
  - This limits what it can do to optimize: e.g. no re-ordering and no simd
  - Compiler will try to analyze the code & prove there is no aliasing, but this is hard to do
- In C (not C++) there is the `restrict` keyword for non-aliasing pointer
  - For gcc/clang you can use `__restrict__` compiler extension
- https://godbolt.org/z/KWYz8hqbh
- Best solution as usual is to use a library that takes care of this for you
- Trivia: In Fortran arrays are assumed to **not** alias
  - This allows the compiler to do better optimizations: simd, re-ordering
  - One reason for its continued use in scientific computing, typically beats C++ for array math

Summary

run-time	→	compile-time
non-contiguous	→	contiguous
heap	→	stack
copy	→	move
move	→	in-place
list	→	vector
vector	→	array
shared_ptr	→	unique_ptr

Other Tricks

Other tricks (use with caution)

- Not always applicable
- Can potentially even reduce performance
- Typically reduce code readability and maintainability
- But can be worthwhile in specific situations
- Recommendations for use
  - Only use if they give a significant performance improvement
  - Encapsulate and limit their use
  - Add a comment if semantics are unclear or assumptions are being made
  - Future changes in compilers/cpus/etc may turn them from perf gain to loss

Inline functions

- Typically we declare functions in a header file:
  - foo.hpp: `void foo();`
- And define the function in a cpp file:
  - foo.cpp: `foo() { // foo implementation };`
- Benefits of doing this
  - Separate interface from implementation
  - Obey the One Definition Rule (ODR)
- Downside
  - Makes it hard for the compiler to inline foo
    - Although it may be able to if Link Time Optimization (LTO) is enabled

Inline functions

- Alternative: declaration and definition in the header file
- Need `inline` keyword (to avoid violating ODR)
  - foo.hpp: `inline foo() { // foo implementation };`
  - Note: member functions & templates are already implicitly inline
- Potential Benefits
  - Makes it easier for the compiler to inline the function call body
- Potential Costs
  - Slower compilation
  - Larger executable: each translation unit gets its own copy of the function
- Typically only beneficial for very small functions
- Typically not a good idea to do this for all the code
  - Although if using templates you have no choice!

Branchless conditions in numerics

- Replace a boolean branch with a cast to int
  - This code involves a branch condition:
    - `if(bool_condition[i]) sum += value[i];`
  - Casting the boolean to an int gives 1 for true, 0 for false
  - Sometimes the logic can be rewritten using this trick without the branch, e.g.
    - `sum += static_cast<int>(bool_condition[i])*value[i];`
  - Removing the branch allows the compiler to better optimize this
    - Doesn't need the branch predictor
    - Can more easily change the order of execution
    - Possibly even use SIMD vectorization
- See Branchless Programming in C++ talk by Fedor Pikus for more

Likely/unlikely annotations

- Use likely/unlikely
  - Hint for branch predictor
  - Hard to beat the compiler though, and easy to do worse!
  - Wouldn’t recommend this

Fast Math

- `-ffast-math` (or `-Ofast` which includes `fast-math`)
  - Flush denormals to zero (they are slower than normal numbers)
  - Allow non-IEEE compliant re-ordering
  - Assume some errors e.g. overflows won't occur
  - Ignore NaN (note this breaks `std::isnan`)
- Can cause a variety of subtle bugs, e.g. if
  - You or a library you use assumes IEEE
  - You or a library you use calls `isnan` to test for NaN
  - You or a library you use requires subnormals for precision
- Safe-ish subset that allows more SIMD
  - `-fno-trapping-math`
  - `-fno-matherrno`

Disable exceptions / RTTI

- Reduces the binary size
  - Often used in embedded systems with limited RAM
- -fno-rtti
  - Disables runtime type information
  - Important: this (silently) breaks `dynamic_cast` and `typeid`
  - Virtual functions will still work
- -fno-exceptions
  - Disables exception handling
  - Important: libraries you link to can still throw exceptions, but you can’t catch them
- Note: stripping the binary is another safe way to reduce its size
  - `strip` command or `-s` linker flag

Code Samples

Code samples

```
git clone --recursive https://github.com/ssciwr/high-performance-cpp.git
cd high-performance-cpp
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
ctest
./bench/bench
```

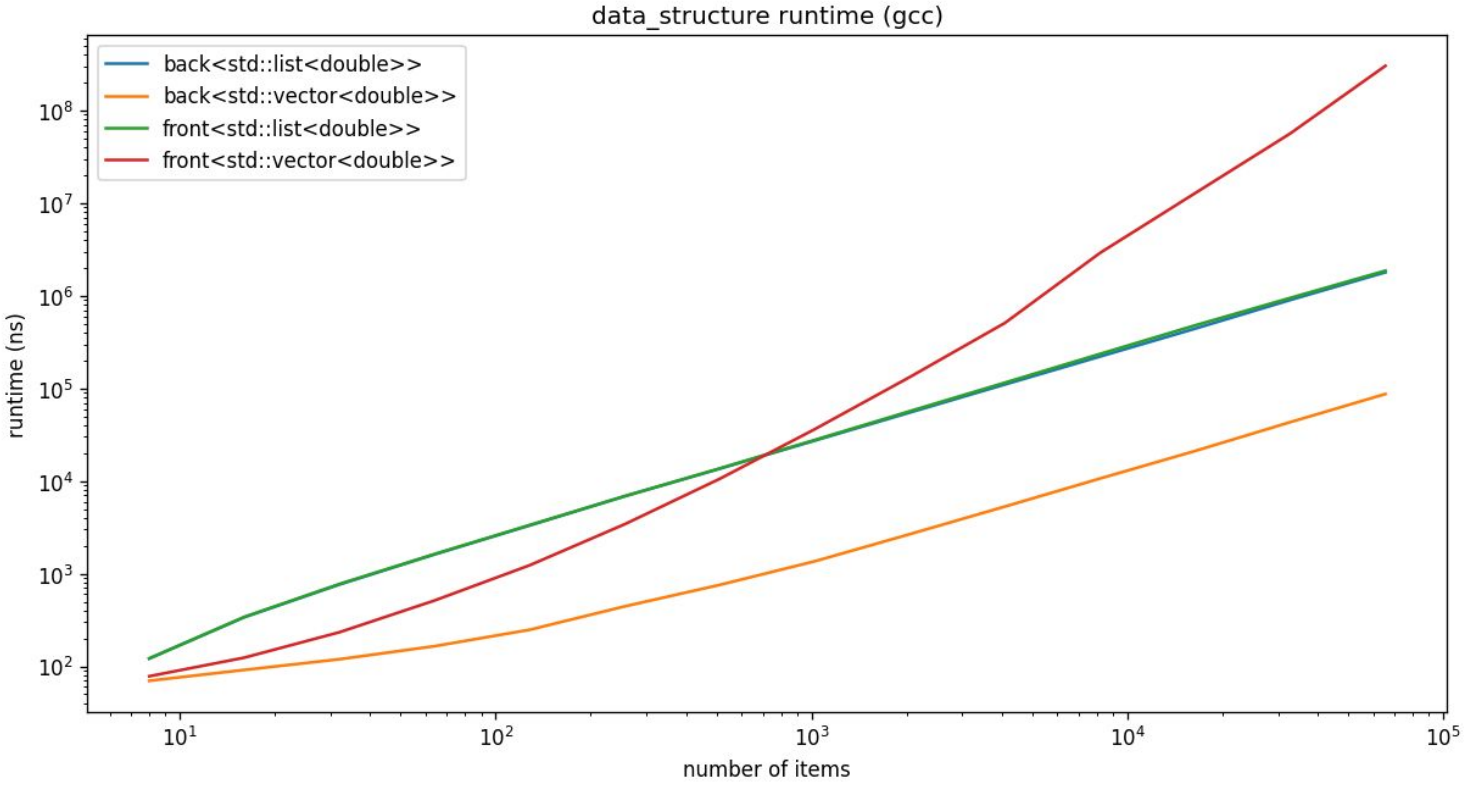
Ex 0 - `std::vector` vs `std::list`

- I claimed vector was almost always better than list
- What is a scenario where list would perform better?
- Repeatedly inserting elements at the front of the container
  - But: consider refactoring to inserting at the back with a vector
- If you need stable element addresses, i.e. address won't change
  - But: consider using a vector of `unique_ptr` instead
- Let's look at the first case to get a feel for the performance
  - Insert n values into the front of the container
  - See how long this takes for different values of n
  - Obviously an artificial use case
  - But an example of writing a benchmark to compare containers
  - `/bench/bench_ex0.cpp`

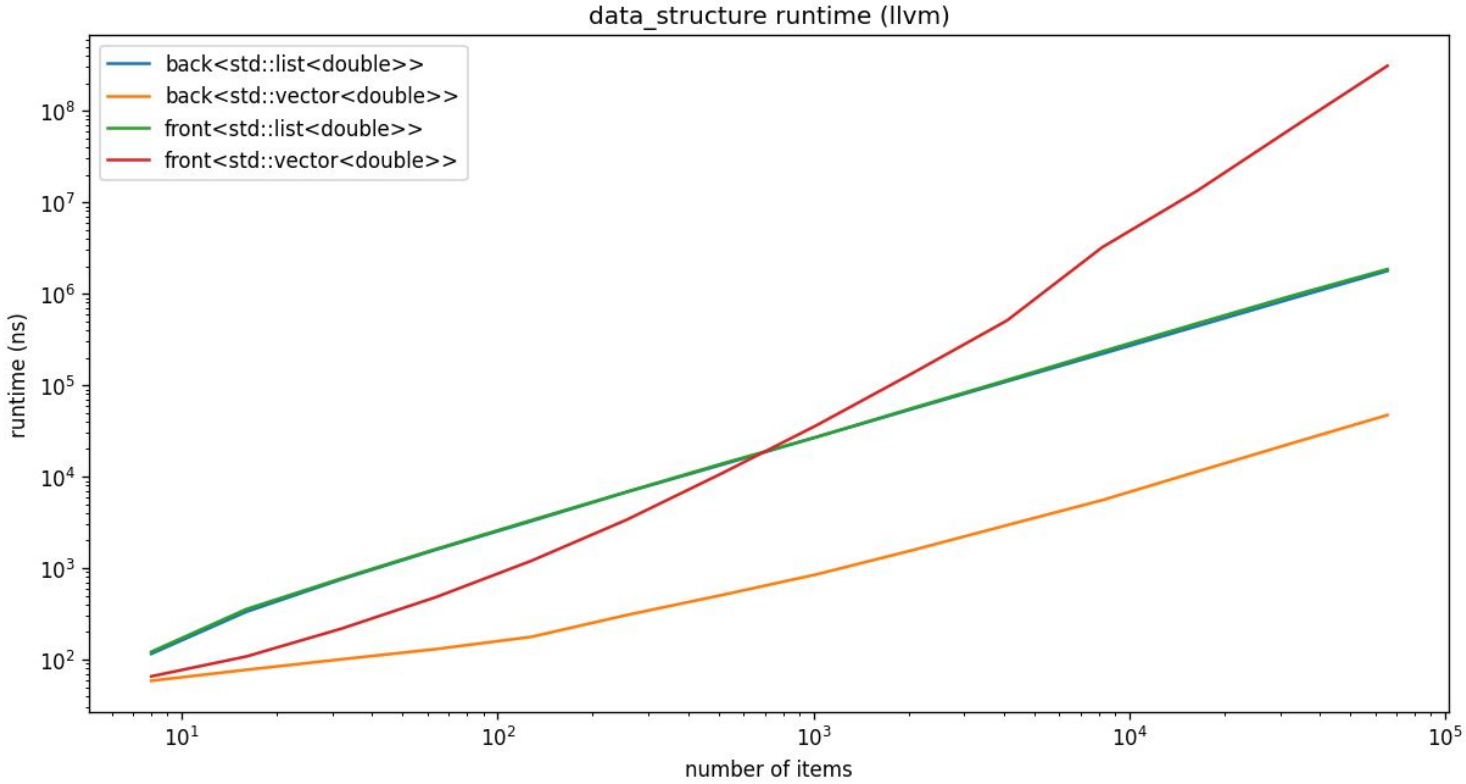
Ex 0 - `std::vector` vs `std::list`

- Insert n `double` values into the container
  - For a list, location point doesn't affect performance
    - Constant cost of insertion
  - For a vector, inserting at the back is the best case (amortized constant cost)
    - Rarely we may need to grow the vector, which means copying/moving all elements
  - Inserting at the front is the worst case expensive (linear in the size of the vector)
    - Need to copy/move all elements to make space at the front
- 65k values
  - list: 1.8ms
  - vector push_back: 0.3ms
  - vector push_front: 328ms
- 8 values
  - list: 116ns
  - vector push_back: 57ns
  - vector push_front: 66ns
- For 512 doubles, vector worst case still beats list!

Ex 0 - `std::vector` vs `std::list`



Ex 0 - `std::vector` vs `std::list`



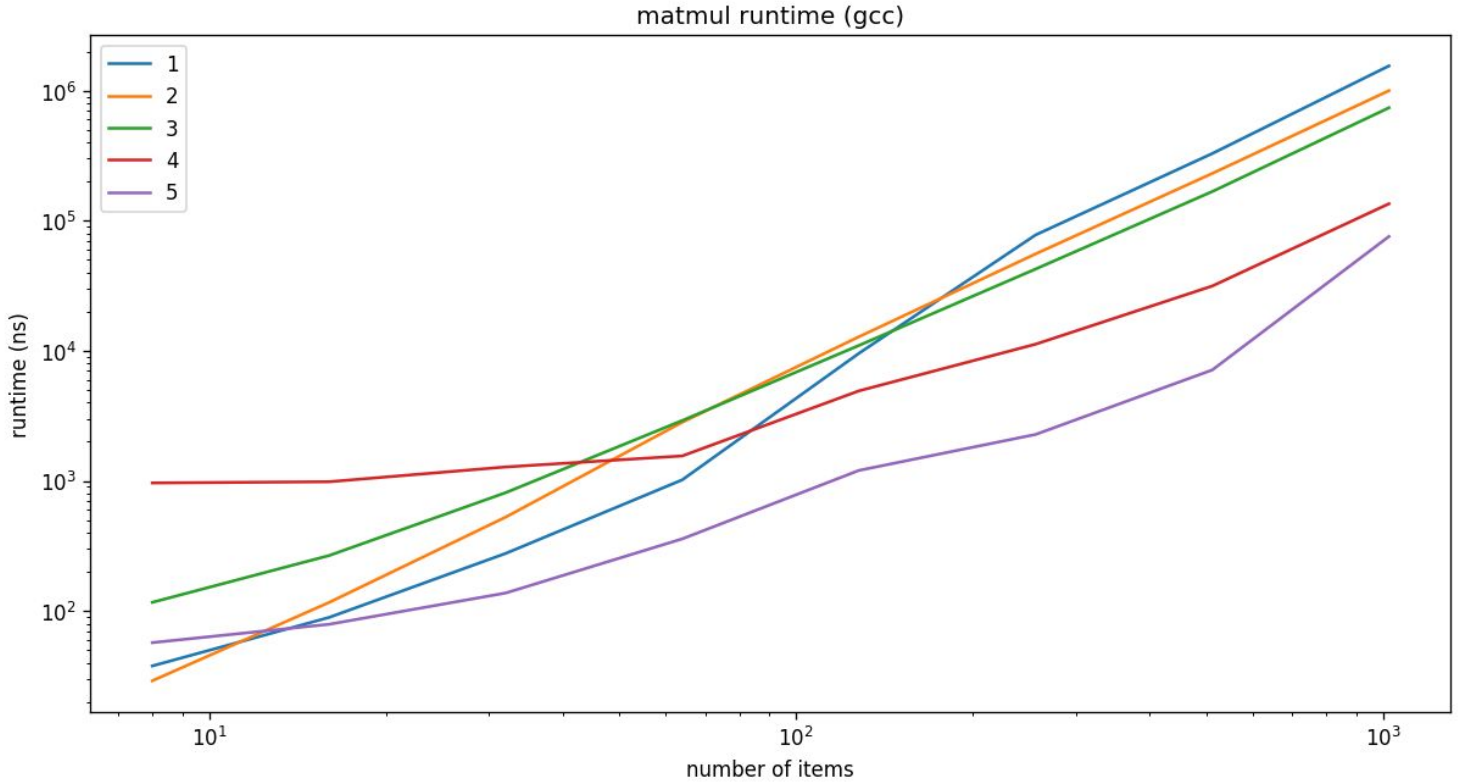
Ex 1 - matrix-vector multiplication

- Starting point
  - Matrix & vector both stored as contiguous data
  - Loop iterates over indices, performs matrix-vector multiplication
- (possible worse starting point)
  - Matrix being a non-contiguous `vector<vector<double>>`
- Things to improve
  - Cache locality
  - Vectorization
  - Parallelization

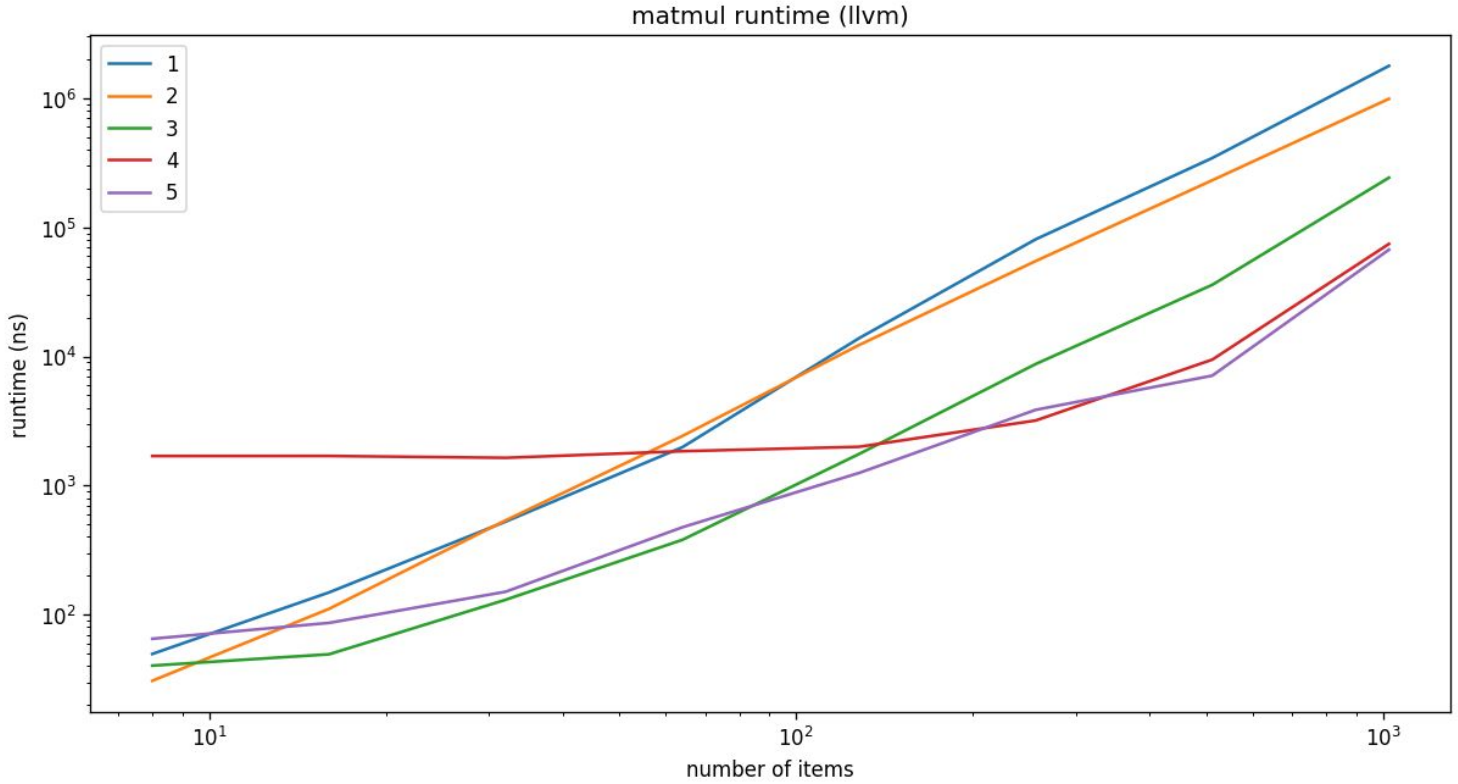
Ex 1 - matrix-vector multiplication

- matmul1
  - non-contiguous iteration order: not cache friendly
  - 1k elements runtime: 2000us
- matmul2
  - same operations, cache-friendly loop order
  - 1k elements runtime: 1000us
- matmul3
  - cache-friendly loop order + simd instructions (`#pragma omp simd reduction(+ : r)`)
  - 1k elements runtime: 280us
- matmul4
  - cache-friendly loop order + simd + use multiple threads (6 cores)
  - 1k elements runtime: 95us
- matmul5
  - use a matrix library (Eigen + openBLAS)
  - 1k elements runtime: 92us

Ex 1 - matrix-vector multiplication



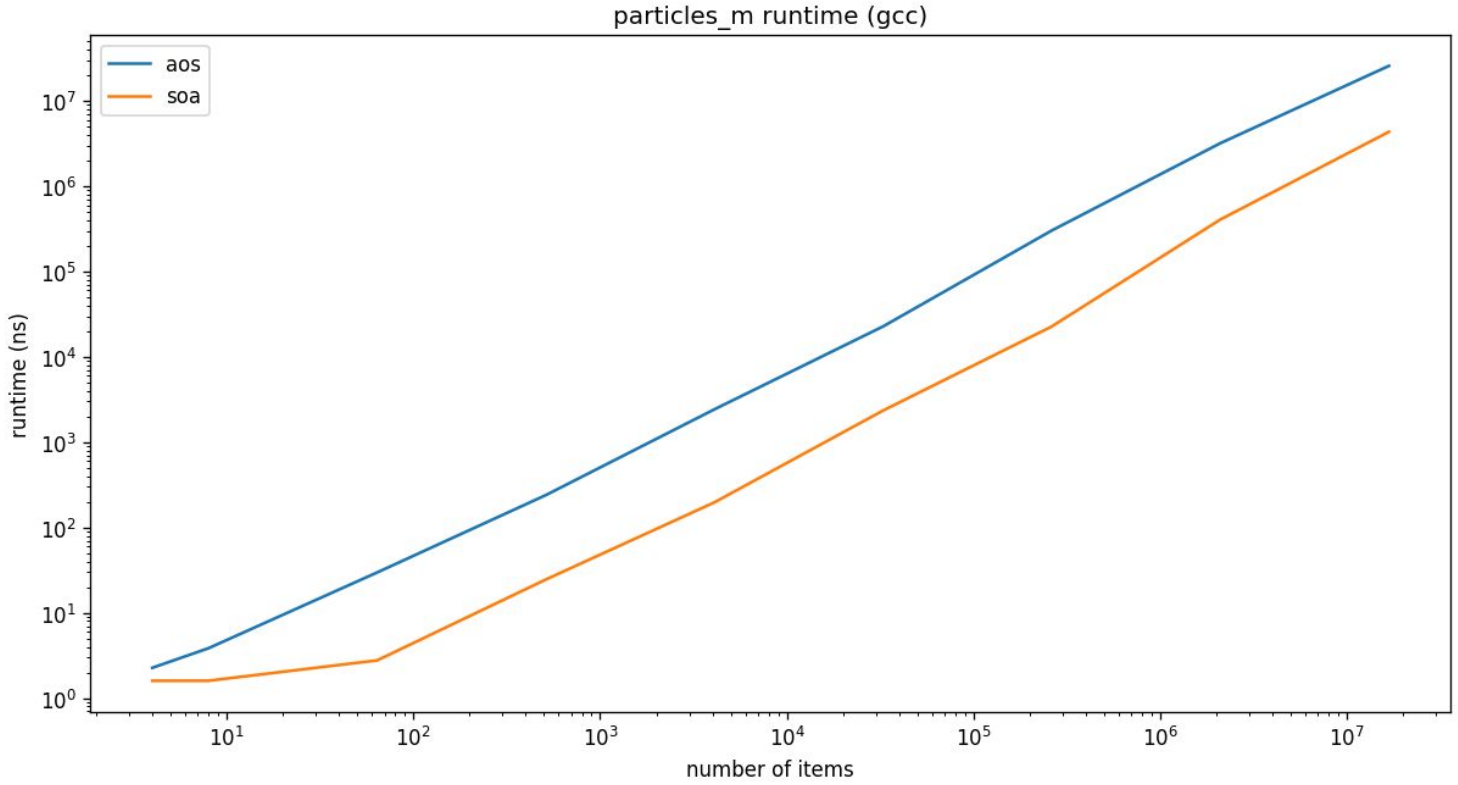
Ex 1 - matrix-vector multiplication



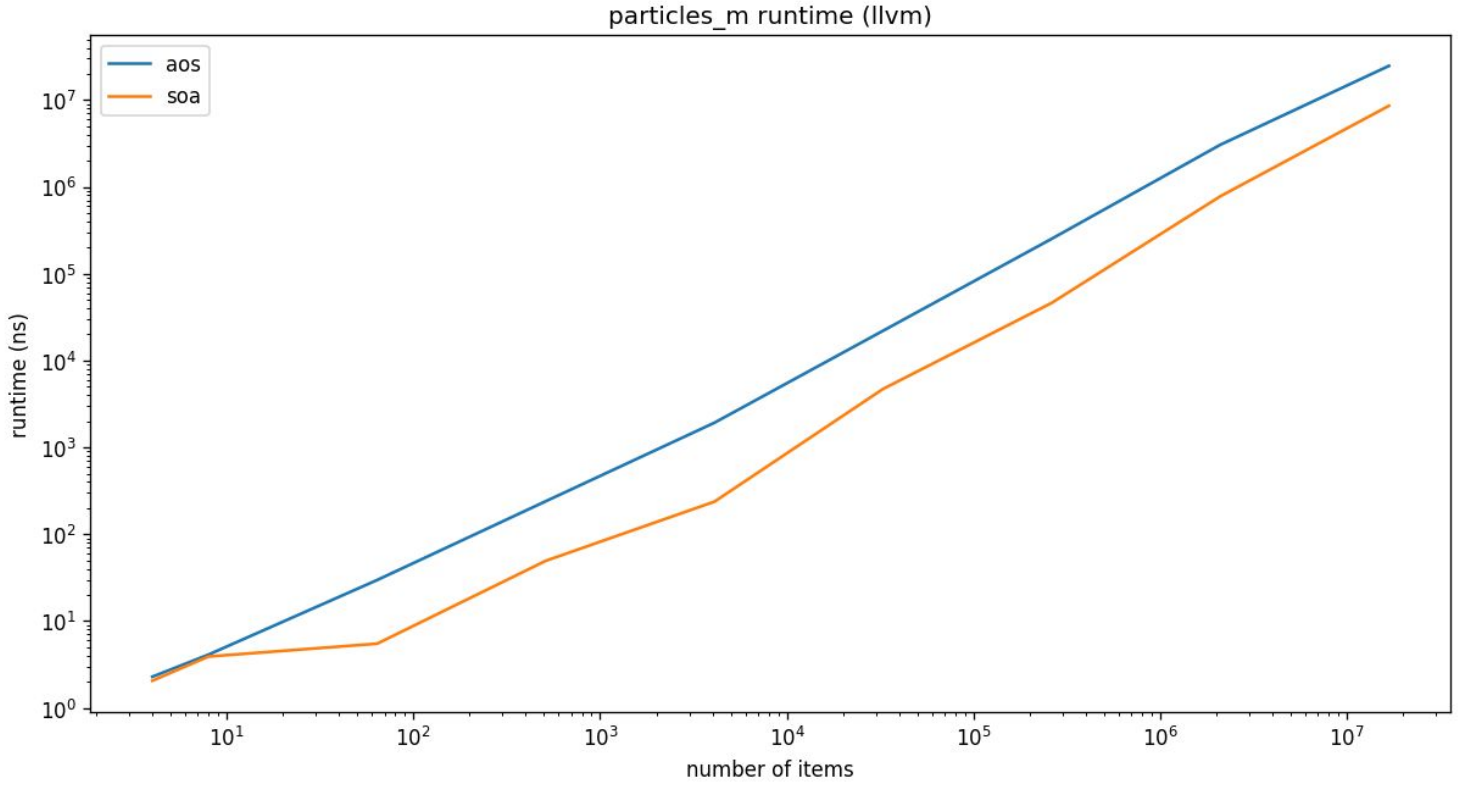
Ex 2 - SoA vs AoS

- Iterate through 64 particles summing x + y for each
  - SoA (two vectors of floats): 40ns
  - AoS (x,y,z,mass,active, 32byte struct): 20ns
  - AoS (x,y,z,mass,active, packed 21byte struct): 30ns
- Iterate through 4096 particles modifying mass for each
  - SoA (one vector of doubles): 400ns
  - AoS (x,y,z,mass,active, 32byte struct): 2000ns
  - AoS (x,y,z,mass,active, packed 21byte struct): 1900ns
- Relevant factors
  - Access patterns
  - Alignment of data for SIMD
  - Size and alignment of data for cache-lines

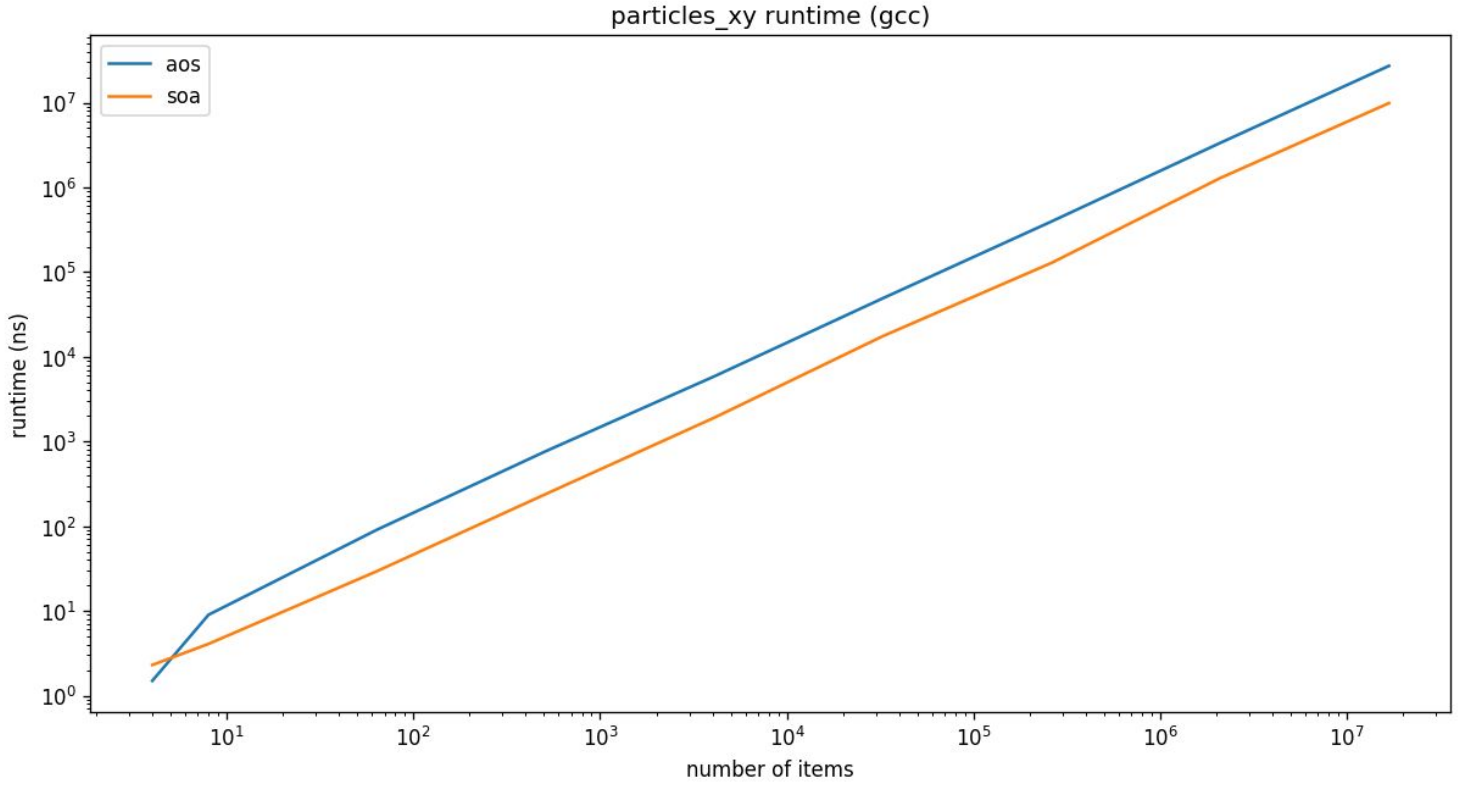
Ex 2 - SoA vs AoS



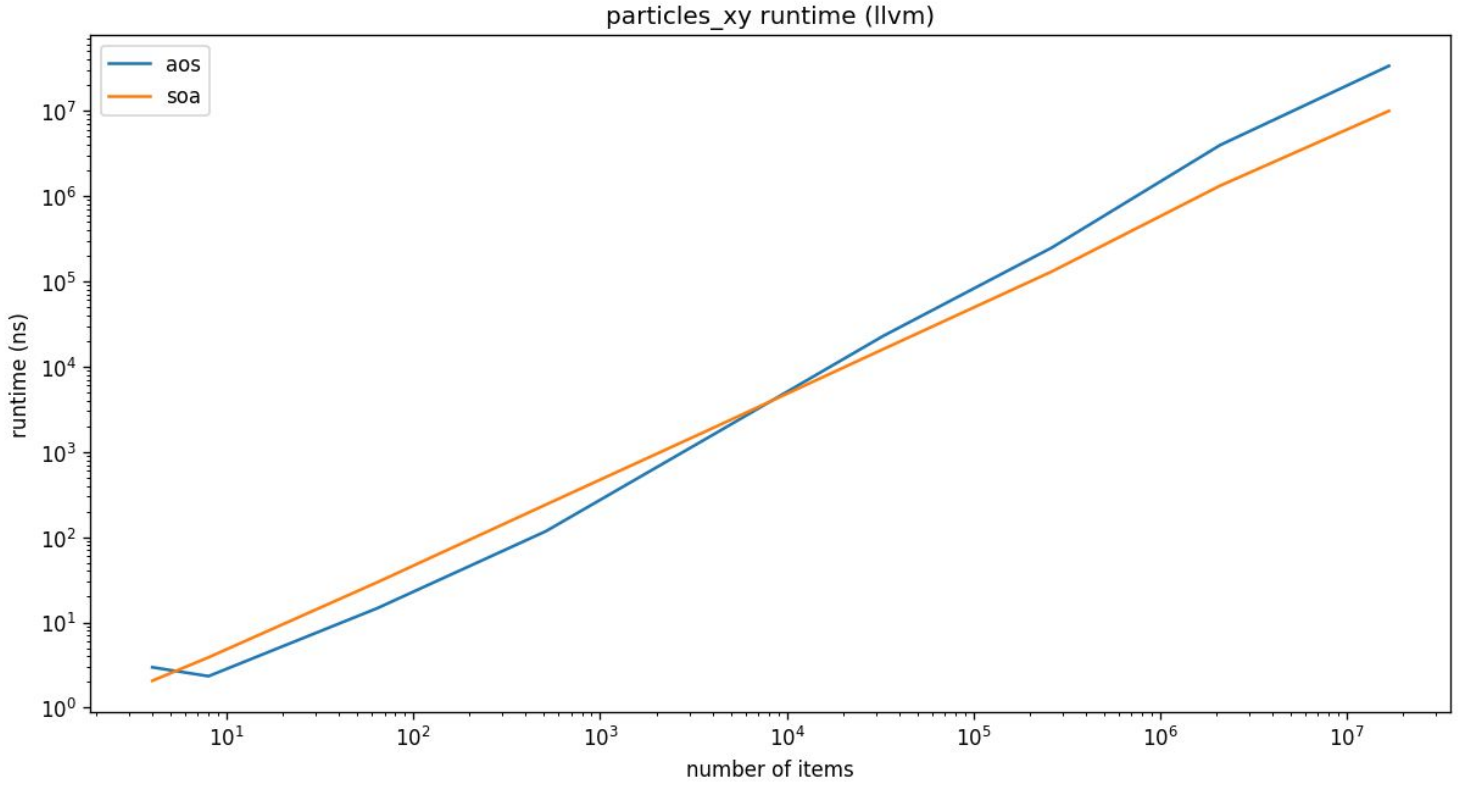
Ex 2 - SoA vs AoS



Ex 2 - SoA vs AoS



Ex 2 - SoA vs AoS



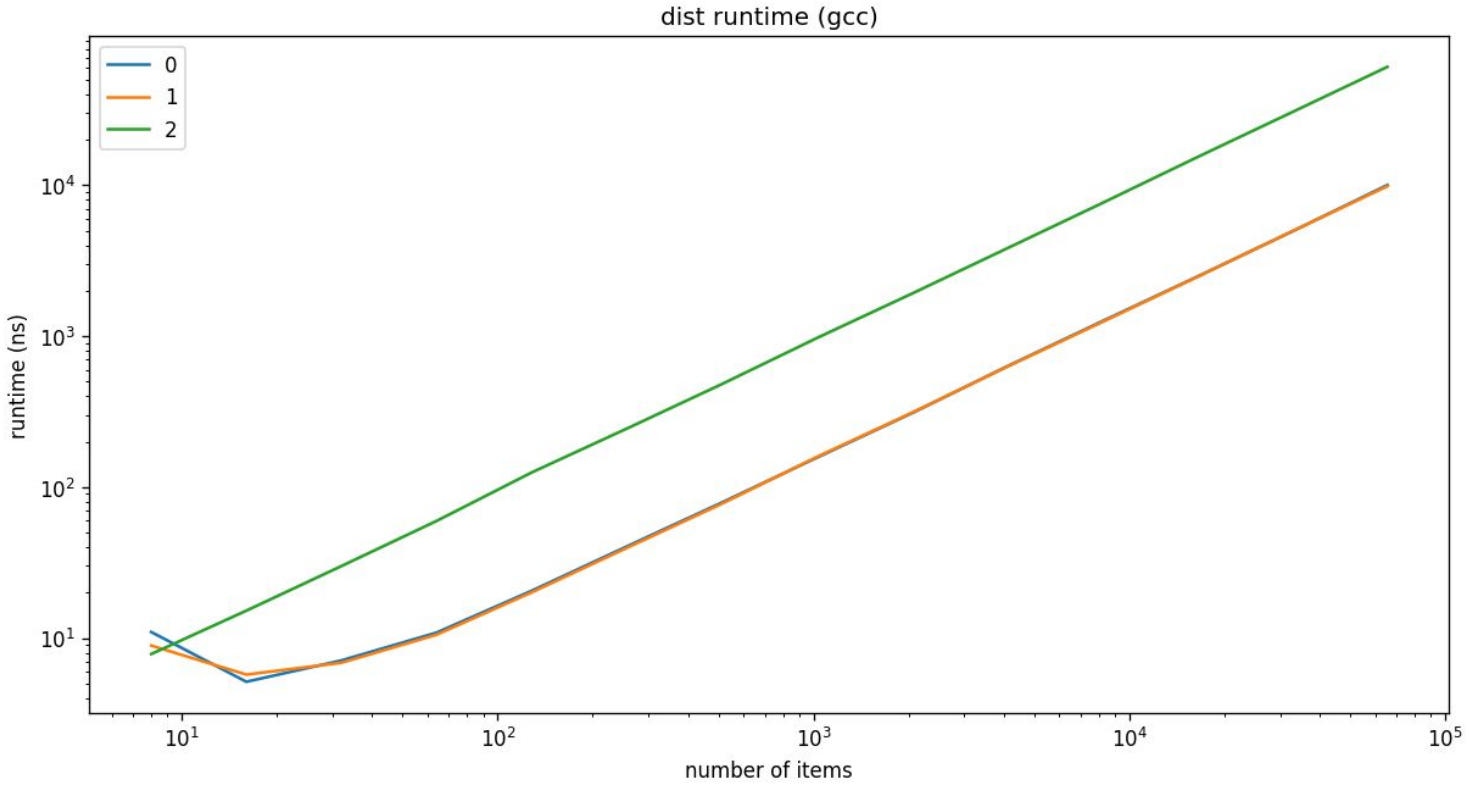
Ex 3 - hamming distance

- A distance measure between two genomes
  - For us, a genome is a string of ‘A’, ‘C’, ‘G’, ‘T’ and ‘-’ characters
  - We compare each character from two genomes
  - If they differ, we add 1 to the distance measure
  - Except: the ‘-’ character is special and has zero distance from any character
- Starting point
  - Iterate through each element in the two genomes
  - If they differ and neither are ‘-’ increment the distance
- Things to improve
  - Can we remove the branch condition?
  - Can we use a better data structure?
  - Can this be vectorized?
  - Can this be parallelized?

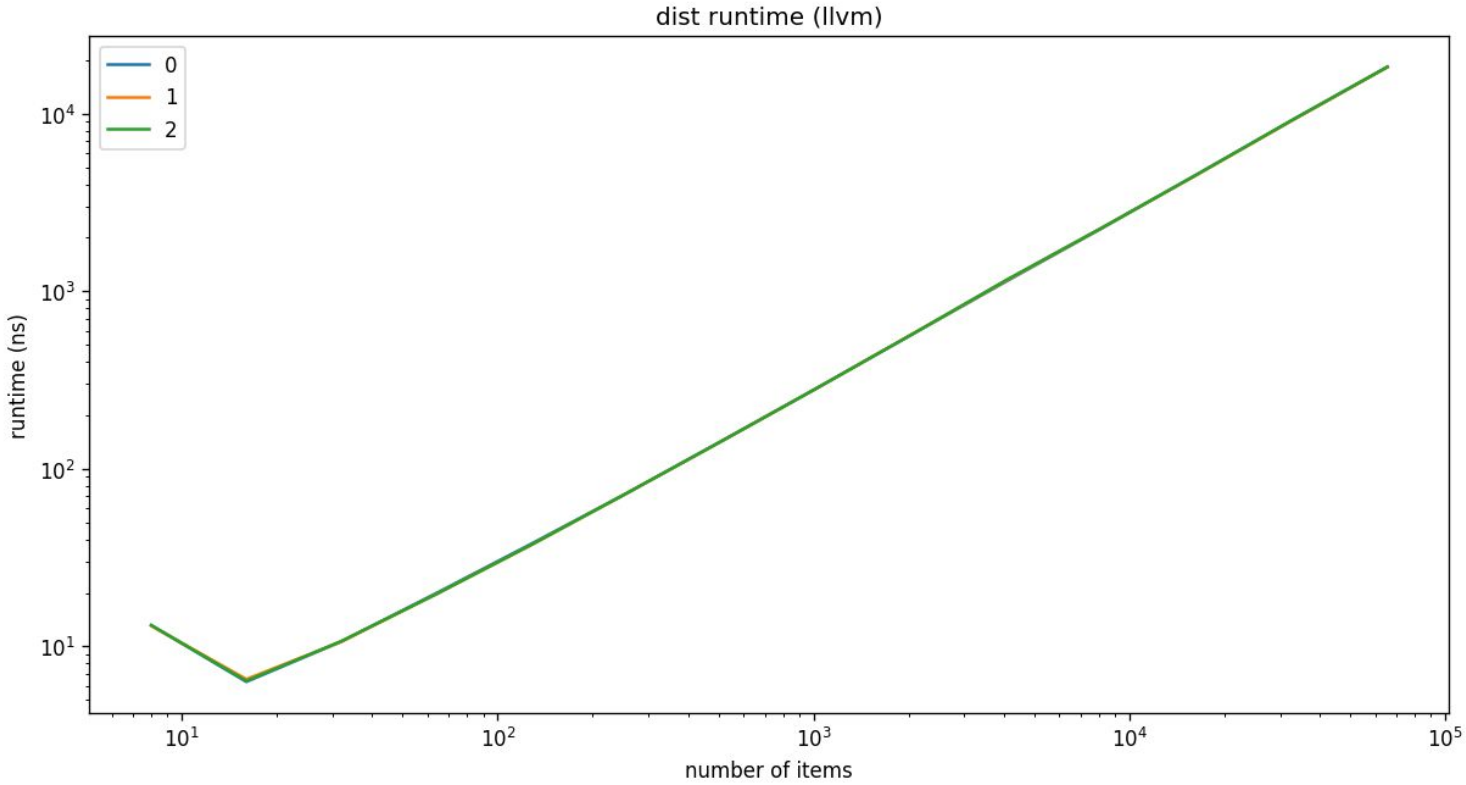
Ex 3 - hamming distance

- Things to try
  - Different compiler
  - Different compile flags
  - Different but equivalent code
  - Different omp pragmas
- All of the above can dramatically affect performance
  - Often in non-intuitive ways!
- Lessons to learn
  - Always need to profile/benchmark code (with real data/use-case)
  - Valuable to experiment with compilers/flags/pragmas
  - One advantage of writing SIMD intrinsics by hand: you know what you will get!

Ex 3 - hamming distance



Ex 3 - hamming distance



Ex 3 - hamming distance

- Real world implementation
  - https://github.com/ssciwr/hammingdist
- Contiguous data structure: 4 bits for each gene
  - Twice as many genes per cache-line compared to char
  - 4 bits still more than minimal
    - 4 bits can store 2^4 = 16 possible values, we only need 5
  - But allows distance calculation to be efficiently vectorized
- Vectorized distance function
  - Hand-written SSE2 (128-bit), AVX2 (256-bit), AVX512 (512-bit) versions
  - Up to 128 gene distances calculated in parallel
  - Run-time dispatch of best supported distance function depending on CPU
- OpenMP parallelization

Summary

Strategy

- Write clear, readable, correct code with performance in mind
  - Apply best practices, use good data structures & algorithms
  - But don’t prematurely optimize or sacrifice readability or maintainability (or correctness!)
- Then identify what needs to be improved
  - Profile your code, identify hotspots
  - You need benchmarks to understand performance impact of changes
- Refactor to improve performance
  - Ideally by using better data structures / algorithms / libraries
  - Sometimes by replacing a more general library with specialized custom micro-optimized code
- Keep tradeoffs in mind
  - Development and maintenance effort vs potential benefit
  - Future changes: requirements / data / hardware / compiler will likely all change
  - Reliability: library typically more robust and better tested

Recommended resources

Performance

- Mike Acton talk on data oriented design
  - https://www.youtube.com/watch?v=rX0ItVEVjHc
- Ulrich Drepper PDF “What Every Programmer Should Know About Memory”
  - https://www.akkadia.org/drepper/cpumemory.pdf
  - Not for “every programmer” but excellent info on RAM / cache / performance
  - From 2007 but still very relevant
- Fedor Pikus on designing for performance
  - https://www.youtube.com/watch?v=m25p3EtBua4
- Jason Turner on best practices for performance
  - https://www.youtube.com/watch?v=uzF4u9KgUWI
- Alan Talbot on everyday efficiency
  - https://www.youtube.com/watch?v=EovBkh9wDnM
- A curated list of c++ performance-related resources
  - https://fenbf.github.io/AwesomePerfCpp/

Recommended resources

Compilation

- Danila Kutenin on compiler flags
  - https://www.youtube.com/watch?v=tckHl8M3VXM

Hash maps

- Malte Skarupke talk on hash map implementations and performance
  - https://www.youtube.com/watch?v=M2fKMP47slQ
- Matt Kulukundis talks on Google's hash map
  - https://www.youtube.com/watch?v=ncHmEUmJZf4
  - https://www.youtube.com/watch?v=JZE3_0qvrMg

Recommended resources

Benchmarking

- Chandler Carruth talks on benchmarking/performance
  - https://www.youtube.com/watch?v=nXaxk27zwlk
  - https://www.youtube.com/watch?v=fHNmRkzxHWs
  - https://www.youtube.com/watch?v=2EWejmkKlxs
- Fedor Pikus talk on branchless programming
  - https://www.youtube.com/watch?v=g-WPhYREFjk

Profiling

- http://pramodkumbhar.com/2017/04/summary-of-profiling-tools/