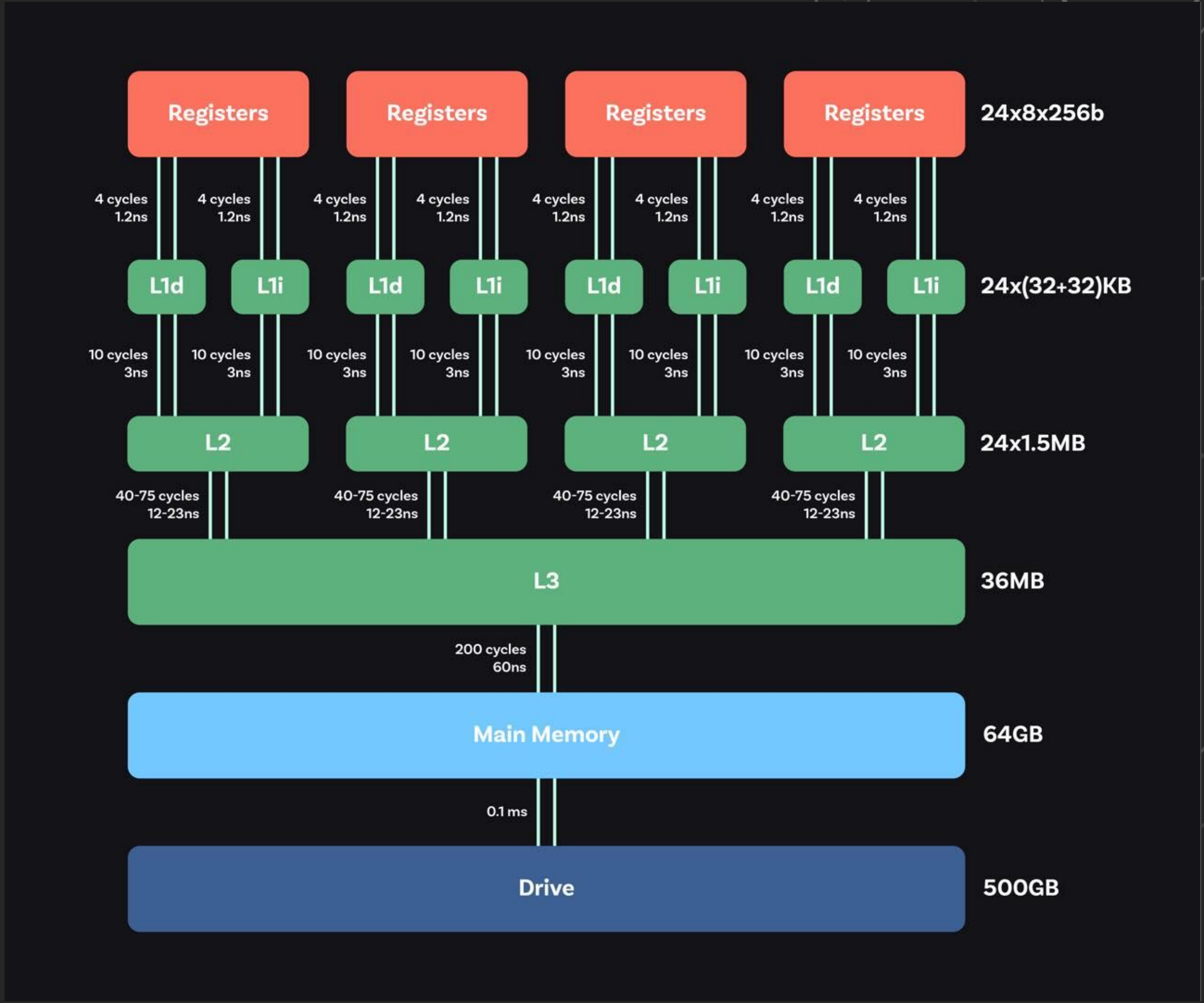
MODERN OPTIMISATION TECHNIQUES IN ACTUARIAL MODELLING

PIOTR GODLEWSKI

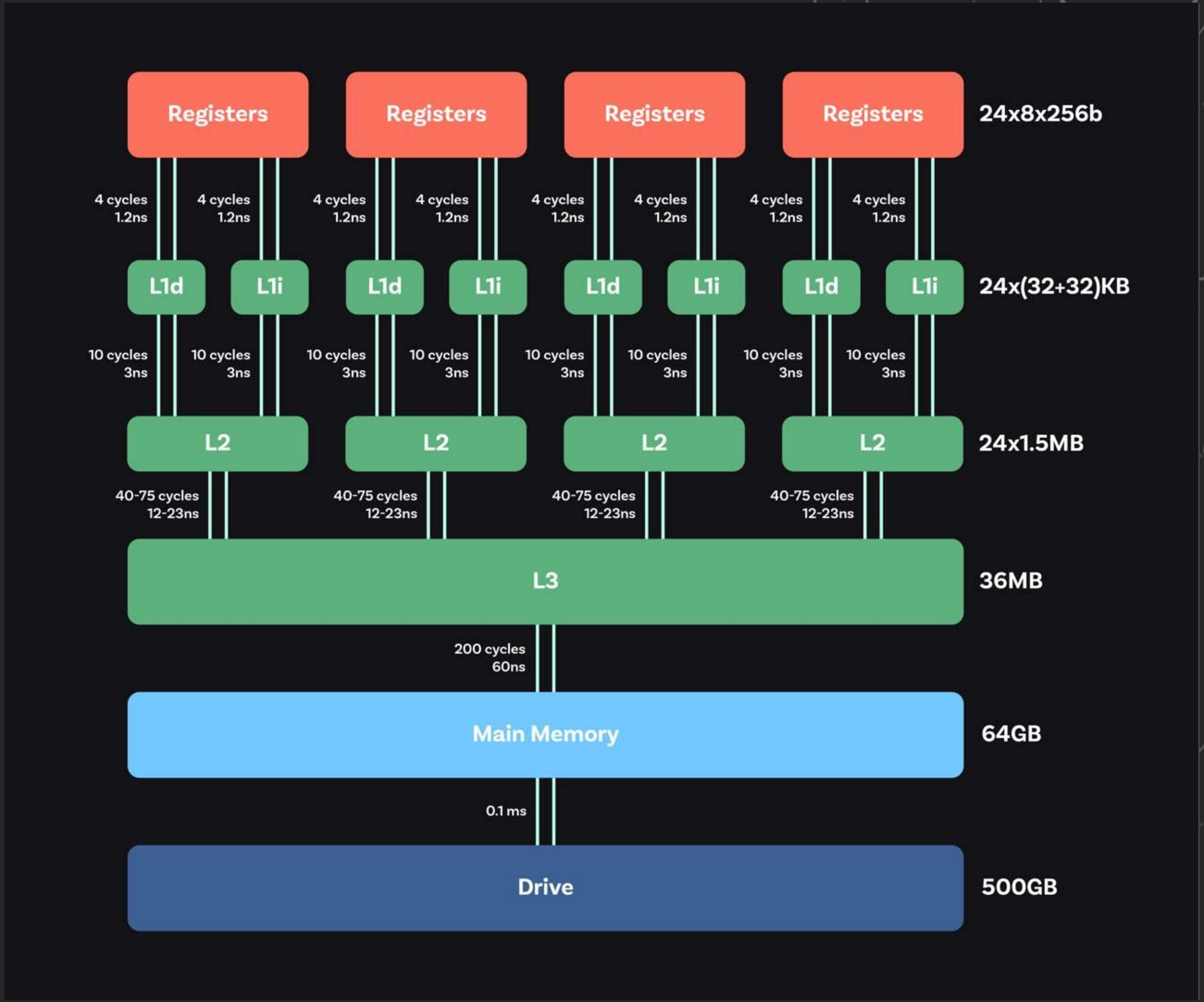
CACHE STRUCTURE

- Only data stored in CPU registers can be accessed directly by CPU
- Data needs to be transferred from main memory or drive to registers through layers of caches
- Caches closer to registers are smaller, but have lower latency



CACHE LATENCY

- Data transfers between main memory and cache might slow down calculations by orders of magnitude
- Data is transferred between caches and main memory in blocks (cache line), usually of 64-byte size
- Cache-friendly code should be based on the principle of *locality*
- Contiguous data structures should be preferred, e.g. std::vector in C++ or numpy.array in Python



PREFETCHING

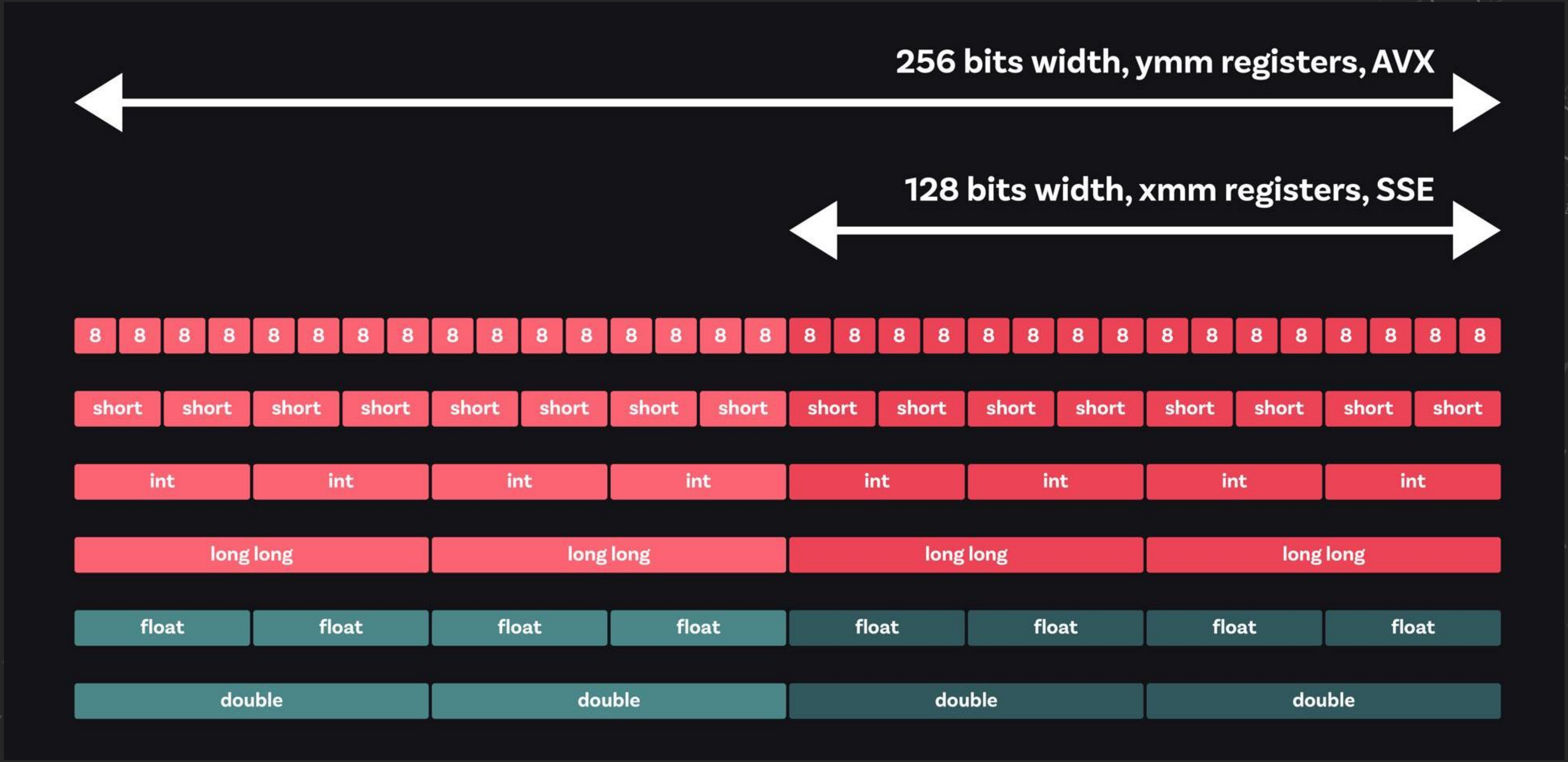
- Modern CPUs can recognise memory access patterns and copy data to L1 cache and registers before this data is requested by a program
- Prefetching is only possible when memory is accessed sequentially
- Contiguous data structures make prefetching more effective

BRANCH PREDICTION

- Modern CPUs try to guess which branch of an if-statement is going to be executed before the condition is evaluated
- The predicted branch is speculatively executed while the if-statement condition is being evaluated
- If the guess was wrong, the executed instructions are discarded and the correct branch is executed, causing a delay

```
while (condition1()) {
    if (condition2()) {
        // Do A
    }
    else {
        // Do B
    }
}
```

ADVANCED VECTOR EXTENSIONS



ADVANCED VECTOR EXTENSIONS

- Modern CPUs do floating-point arithmetic on 256-bit vector registers
- Each vector register can store up to 4 double precision or 8 single precision values
- AVX instructions are performed on all values stored in a vector register
- AVX can speed up calculations by up to 4 times for double precision and by up to 8 times for single precision
- Most CPUs support AVX instructions on integers (AVX2) and some have 512-bit registers (AVX512)

LOOPING ORDER

Inner loop over outer dimension

```
for (int t = 0; t < t_max; t++) {
    for (int i = 0; i < num_mps; i++) {
        top_model.liab[t] += policies[i].liab[t];
    }
}
```

Inner loop over inner dimension

```
for (int i = 0; i < num_mps; i++) {
    for (int t = 0; t < t_max; t++) {
        top_model.liab[t] += policies[i].liab[t];
    }
}
```

Benchmark (num_mps=1'000'000, t_max=600):

```
Aggregation time outer: 3668.530800 ms
Aggregation time inner: 248.836500 ms
Aggregation time inner AVX: 183.273100 ms
```

BATCHED CALCULATIONS

Inner loop over outer dimension

```
for (int s = 0; s < num_sims; s++) {
    for (int i = 0; i < num_lfs; i++) {
        agg_loss[s] += losses[i][s];
    }
}
```

Inner loop over inner dimension

```
for (int i = 0; i < num_lfs; i++) {
    for (int s = 0; s < num_sims; s++) {
        agg_loss[s] += losses[i][s];
    }
}
```

Benchmark (num_lfs=100, num_sims=1'000'000):

```
Aggregation scenario outer: 67.502400 ms
Aggregation scenario inner: 41.798400 ms
Aggregation batched: 28.353700 ms
```

Batched inner loop over inner dimension

```
for (int b = 0; b < num_sims; b += batch_size) {
    for (int i = 0; i < num_lfs; i++) {
        for (int s = 0; s < batch_size; s++) {
            agg_loss[b + s] += losses[i][b + s];
        }
    }
}
```

PROGRAMMING LANGUAGE TYPES

- Compiled
- Interpreted
- Just-In-Time (JIT) aka Virtual Machine (VM)

Table 4
Normalized global results for Energy, Time, and Memory.

Total					
	Energy (J)		Time (ms)		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Ranking programming languages by energy efficiency, Rui Pereira et al. (2021)

PROGRAMMING LANGUAGE TYPES

Compiled

Example: C, C++, Rust

Pros:

- Highly optimised
- Full control over hardware

Cons:

- Coding from scratch
- Code needs to be recompiled after every change

Interpreted

Example: Python, R, VBA, MATLAB

Pros:

- Compact and easy to understand
- Various libraries/packages available
- Code can be modified at runtime

Cons:

- Slow
- High memory usage
- Impossible to access some hardware features (AVX, multithreading in R or VBA)

OPTIMISATION FEATURES

- Code manipulation
- Profile-guided optimisation
- Auto-vectorisation
- Inlining
- Fast floating-point arithmetic
- Compile-time evaluation (C++)
- Metaprogramming (C++)

AUTO-VECTORISATION

AVX intrinsics

```
void avx_div(int n, double* a, double* b, double* c) {
    for (int i = 0; i < n; i += 4) {
        _mm256_store_pd(&c[i], _mm256_div_pd(_mm256_load_pd(&(a[i])), _mm256_load_pd(&(b[i]))));
    }
}

void avx_exp(int n, double* a, double* c) {
    for (int i = 0; i < n; i += 4) {
        _mm256_store_pd(&c[i], _mm256_exp_pd(_mm256_load_pd(&(a[i]))));
    }
}
```

Auto-vectorisation

```
void div(int n, double* a, double* b, double* c) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] / b[i];
    }
}

void div_noalias(int n, double* a, double* b, double* __restrict c) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] / b[i];
    }
}

void exp(int n, double* a, double* c) {
    for (int i = 0; i < n; i++) {
        c[i] = exp(a[i]);
    }
}
```

Benchmark (n=1000):

```
Division: 838.976200 ms
Division auto-vectorised: 425.157700 ms
Division auto-vectorised noalias: 394.536900 ms
Division vectorised: 391.428100 ms
Exponential: 1344.057900 ms
Exponential vectorised: 549.899900 ms
```

AUTO-VECTORISATION

Modern compilers can auto-vectorise code quite efficiently. However, in some cases implicit vectorisation cannot be performed, including:

- Noncontiguous data structures
- Nonsequential data access patterns
- Code branches (e.g. if statements)
- Data dependency (aliasing)
- Data alignment

FUNCTION INLINING

- Calling a function is a relatively costly process, especially in interpreted languages
- For example, a function call in Python is 2-3 orders of magnitude slower than in C++
- Compilers are able to significantly optimise performance by inlining small functions, essentially "copy-pasting" function's code in-place
- Modern compilers are extremely efficient in determining whether to inline a function or not
- Not every function should be inlined!

FAST FLOATING-POINT ARITHMETIC

- Floating-point arithmetic is not exact due to rounding errors
- The most common convention for FP calculations is IEEE-754
- Fast FP arithmetic allows compiler to reorder, combine or simplify calculations under assumptions of perfect arithmetic
- This might result in a different output, but not necessarily worse

Benchmark (size=100'000'000):

```
/fp:precise: 130.152800 ms
/fp:fast: 75.551800 ms
```

```
for (int s = 0; s < size; s++) {
    a = a * b + c;
    // With /fp:fast equivalent to
    // a = fma(a, b, c);
}
```

COMPILE-TIME EVALUATION

- If data required to perform computation is available to compiler, it can evaluate it in advance of a model run
- Thanks to compile-time evaluation, costly function calls are omitted, since the result has already been computed and stored in memory
- This is most useful when function would be called many times, e.g. per model point
- Compile-time evaluation makes auto-vectorisation easier
- In C++20 standard most of arithmetic functions became constexpr

```
constexpr int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

void test(double x) {
    constexpr int m = factorial(3);
    // Equivalent to
    // constexpr int m = 6;

    const int k = factorial(m);
    // Equivalent to
    // const int k = 720;

    double y = pow(x, 1. / 12.);
    // Equivalent to
    // double y = pow(x, 0.08333333333333333);
}
```

METAPROGRAMMING

Function

```cpp
void f(bool b) {
    if (b) {
        // Do A
    }
    else {
        // Do B
    }
}

void test() {
    f(true);
    f(false);
}
```

Template specialisation

```cpp
template<bool b> void f() {
    if constexpr (b) {
        // Do A
    }
    else {
        // Do B
    }
}

void test() {
    f<true>();
    f<false>();
}
```

ACTUARIAL MODELLING

- Actuarial models (cash flow, capital, ESG) are most often memory bound
- Cache-friendly code is the key to high performance
- Models with vectorised calculations are faster due to prefetching and branch prediction
- They can be further accelerated with AVX
- Compiled languages offer superior performance in both run time and memory usage

MODERN OPTIMISATION TECHNIQUES IN ACTUARIAL MODELLING

Piotr Godlewski

piotrgodlewski391@gmail.com

linkedin.com/in/piotr-godlewski/