Practical vectorization of C++ code for SIMD acceleration

Bloomberg

Mathias Gaunard

Bloomberg LP

May 10, 2016

What's SIMD?

Parallelism is increasingly present

Obvious parallelism

- Multi-core architectures
- Many-core architectures
- Distributed systems

Less obvious parallelism

- Superscalar execution + pipelining
- SIMD extensions

Parallelism is increasingly present

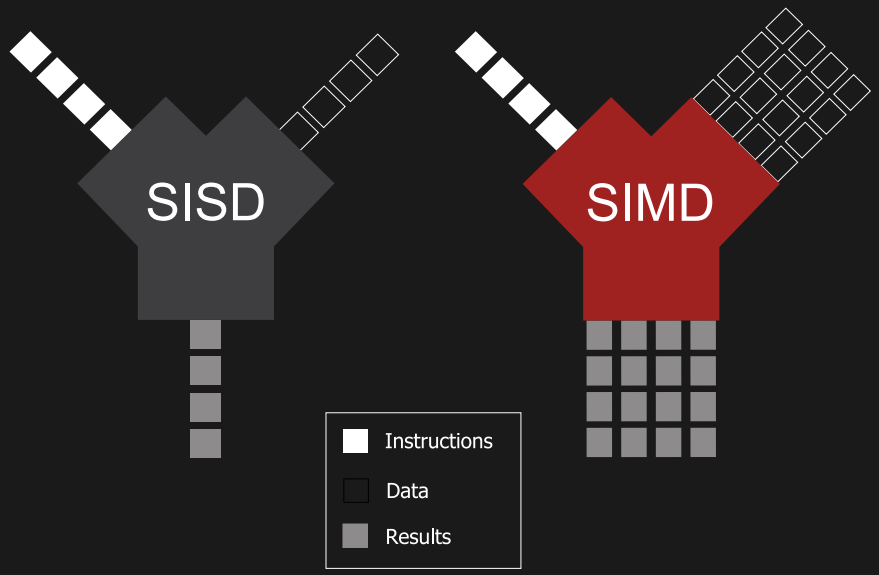
Obvious parallelism

- Multi-core architectures
- Many-core architectures
- Distributed systems

Less obvious parallelism

- Superscalar execution + pipelining
- SIMD extensions

What's SIMD?



Principles

- Single Instruction, Multiple Data
- Each operation is applied to *N* values in a single register (if 128-bit register, 4 32-bit values or 2 64-bit values)
- In principle, processes *N* times as much data in a single cycle as the normal ALU/FPU

1001 flavours of SIMD

x86 extensions

- MMX 64-bit float, double
- SSE 128-bit float
- SSE2 128-bit int8, int16, int32, int64, double
- SSE3, SSSE3
- SSE4a (AMD)
- SSE4.1, SSE4.2
- AVX 256-bit float, double
- AVX2 256-bit int8, int16, int32, int64
- FMA3
- FMA4, XOP (AMD)
- MIC/KNC 512-bit float, double, int32, int64

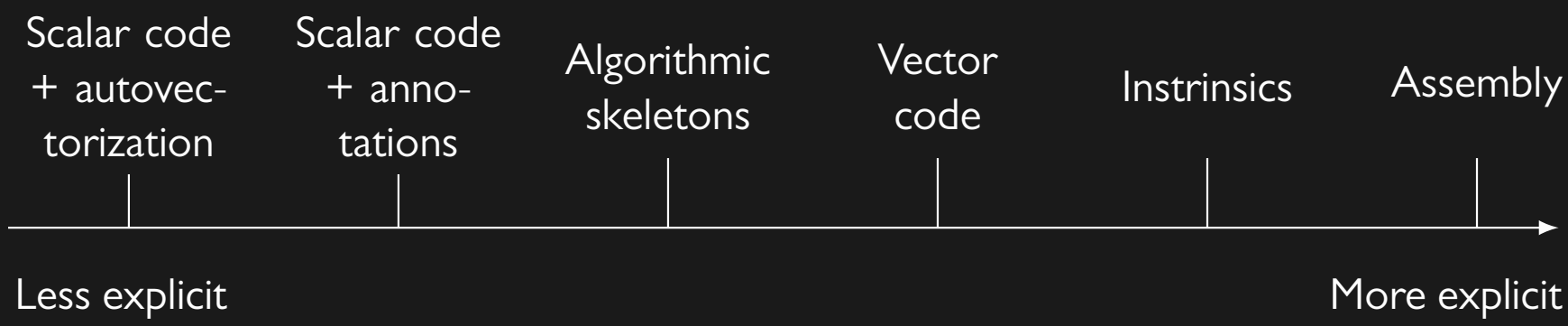
PowerPC extensions

- AltiVec 128-bit int8, int16, int32, int64, float
- Cell SPU and VSX, 128-bit int8, int16, int32, int64, float, double
- QPX 512-bit double

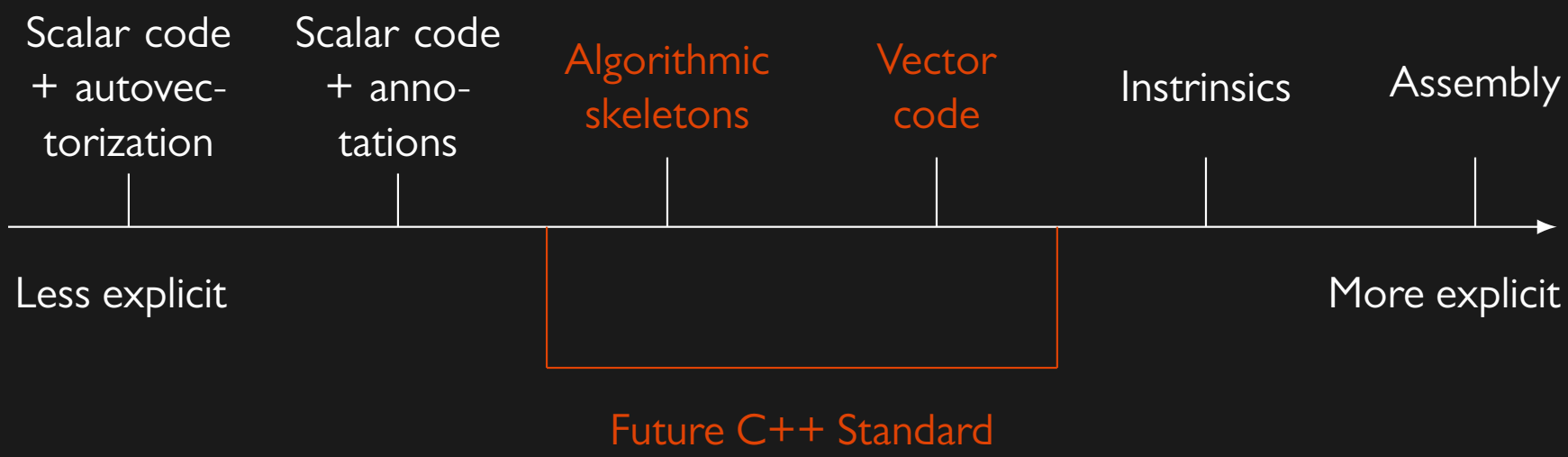
ARM extensions

- VFP 64-bit float, double
- NEON 64-bit et 128 bits float, int8, int16, int32, int64

How to use SIMD?



How to use SIMD?



Overview of Techniques

Auto-vectorization

- Only happens if:
  - memory is well laid out
  - code is intrinsically vectorizable
  - the complexity of the code is manageable by the compiler's heuristics
- Works at loop level; external functions not vectorized
- If it vectorizes, it tends to do it in a silly way as it lacks the high-level information to see the logic behind the code.

Intrinsics

- Pretty much assembly in C, no need to do register allocation manually
- Explicit programming of low-level details
- Efficient but ISA-specific

Overview of Techiques – Towards the Sweet Spot?

Algorithmic Skeletons

- High-level, clearly describes intention
- Easy to use and combine
- Limited to hard-wired patterns

Vector code

- Describe algorithms as operations on vectors
- Like intrinsics, but arbitrary width, portable and C++
- Flexible with small abstraction overhead

SIMD in the C++ Standard

History

- Proposal N3571 by Mathias Gaunard et. al., based on the Boost.SIMD library.
- Proposal N4184 by Matthias Kretz, based on Vc library.

$\Rightarrow$ Unifying efforts and expertise to provide an API to use SIMD portably within C++ (P0203, P0214).

Schedule

- Targeting an optional Technical Specification, Parallelism TS 2, due in 2017.
- Might be integrated later in the C++19 International Standard.

What is SIMD programming like today?

Handwritten intrinsics, `int32 * int32 -> int32`

```
// NEON
return vmul_s32(a0, a1);   // 64-bit
return vmulq_s32(a0, a1); // 128-bit
```

Handwritten intrinsics, `int32 * int32 -> int32`

```
// SSE4.1
return _mm_mullo_epi32(a0, a1);
```

Handwritten intrinsics, int32 * int32 -> int32

```
// SSE2
// get the two parts of the 64-bit result
__m128i lo = _mm_mul_epu32(a0, a1);
__m128i hi = _mm_mul_epu32(
    _mm_srli_si128(a0, 4),
    _mm_srli_si128(a1, 4)
);

// mask, shift and combine to interleave the low parts
__m128i mask = _mm_setr_epi32(0xffffffff, 0, 0xffffffff, 0);
return _mm_or_si128(
    _mm_and_si128(lo, mask),
    _mm_slli_si128(_mm_and_si128(hi, mask), 4)
);
```

Handwritten intrinsics, int32 * int32 -> int32

```
// Altivec
// reinterpret as u16
short0 = (__vector unsigned short)a0;
short1 = (__vector unsigned short)a1;

// shifting constant
shift = vec_splat_u32(-16);
sf = vec_rl(a1, shift_);

// Compute high part of the product
high = vec_msum( short0, (__vector unsigned short)sf
               , vec_splat_u32(0)
               );

// Complete by adding low part of the 16 bits product
return vec_add( vec_sl(high, shift_)
              , vec_mulo(short0, short1)
              );
```

datapar API

The datapar abstraction

`std::datapar<T, N, Abi>`

`datapar<T, N>`	SIMD register holding `N` elements of type `T`
`datapar<T>`	same with optimal `N` for the currently targeted architecture
`Abi`	Defaulted ABI marker to make types with incompatible ABI different

Behaves like a value of type `T` but applying each operation on the *N* values it contains, possibly in parallel.

Constraints

- `T` must be an integral or floating-point type (tuples/struct of those once we get reflection)
- `N` parameter under discussion, probably will need to be power of 2.

Operations on datapar

Built-in operators

- All usual binary operators are available, for all $\oplus$:
  `datapar<T, N>` $\oplus$ `datapar<U, N>`,
  `datapar<T, N>` $\oplus$ `U`, `U` $\oplus$ `datapar<T, N>`
- Compound binary operators and unary operators as well
- `datapar<T, N>` convertible to `datapar<U, N>`
- `datapar<T, N>(U)` broadcasts the value
- No promotion:
  `datapar<uint8_t>(255) + datapar<uint8_t>(1) == datapar<uint8_t>(0)`

Comparisons and conditionals

- `==`, `!=`, `<`, `<=`,`>` and >= perform element-wise comparison
  return `mask<T, N, Abi>`
- `if(cond) x = y` is written as `where(cond, x) = y`
  `cond ? x : y` is written as `if_else(cond, x, y)`

Memory management

Load

- Constructor:

```
U* ptr;
datapar<T> data(t, options);
```

- Member function:

```
U* ptr;
datapar<T> data;
data.load(ptr, options)
```

- Free function:

```
U* ptr;
datapar<T> data = load< datapar<T> >(ptr, options);
```

Memory management

Store

- Member function:

```
U* ptr;
datapar<T> data;
data.store(ptr, options);
```

- Free function:

```
U* ptr;
datapar<T> data;
store(data, ptr, options);
```

Optional compile-time options

- non-aligned (default) / aligned
- normal (default) / non-temporal

Memory management

Access

- `size()` constexpr static member
- `operator[](size_t)` allows to access `i`-th element
- Returns a proxy rather than a reference to make avoiding memory more likely

Deinterleave and gather/scatter

- Load/store requires contiguous elements
- Auto-deinterleaving and index-based memory access considered
- Not finalized yet

Register re-arrangement

Unary and binary shuffle

- Re-order elements within a datapar
- Combine two datapars

```
int a_data[] = {10, 11, 12, 13};
datapar<int, 4> a(a_data);

datapar<int, 4> r = shuffle<0, 2, 1, 3>(a);

int r_data[] = {10, 12, 11, 13};
assert(all_of(r == datapar<int, 4>(r_data)));
```

```
int b_data[] = {20, 21, 22, 23};
datapar<int, 4> b(b_data);

datapar<int, 4> r2 = shuffle<0, 5, 1, 6>(a, b);

int r2_data[] = {10, 20, 11, 21};
assert(all_of(r2 == datapar<int, 4>(r2_data));
```

Generalized conversion

N to M conversion

- Convert 4 datapar<int64_t, N> to a single datapar<int8_t, N*4> and all other combinations
- Can also be used for combining/slicing
- Either return a datapar or an array/tuple of datapar depending on input and output types

```
template<class T, class U, class... Us>
conditional_t< (T::size() == (U::size() + Us::size()...)),
               T,
               array<T, (U::size() + Us::size()...) / T::size()>
             >
datapar_cast(U, Us...);
```

Mask Reductions

- Reduce all values in the mask to a scalar
- Also defined on bool

```
bool all_of(mask<T, N, Abi>);
bool any_of(mask<T, N, Abi>);
bool none_of(mask<T, N, Abi>);
bool some_of(mask<T, N, Abi>);
int  popcount(mask<T, N, Abi>);
int  find_first_set(mask<T, N, Abi>);
```

General Reductions

General-purpose reduction to a scalar

- Compute sum, product, minimum, etc.
- $log_2(N)$ shuffles and operations

```
datapar<T> data(&memory[i]);

auto f = [](auto a, auto b) { return a + b; };
T r = reduce(data, f);

datapar<T> r2 = reduce_broadcast(data, f);
// faster than reduce+broadcast
```

Mathematical Functions

Math Functions:				
abs	cbrt	ceil	copysign	fdim
floor	fma	fmax	fmin	fmod
frexp	hypot	ilogb	ldexp	logb
lrint	lround	modf	nan	nanf
nanl	nearbyint	nextafter	nexttoward	
remainder	remquo	rint	round	scalbln
scalbn	sqrt		trunc	
Classification/comparison Functions:				
fpclassify	isfinite	isgreater	isgreaterequal	isinf
isless	islessequal	islessgreater	isnan	isnormal
isunordered	signbit			
Integer Functions:				
div	ldiv	lldiv		

Practical Example: The Sum

Difference in logic

Scalar code

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result = 0;
    for(; first != last; ++first)
        result += *first;
  return result;
}
```

Parallel logic

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result0 = 0,
             result1 = 0,
             result2 = 0,
             result3 = 0;
    for(; first != last; first += 8)
    {
        result0 += first[0];
        result1 += first[1];
        result2 += first[2];
        result3 += first[3];

        result0 += first[4];
        result1 += first[5];
        result2 += first[6];
        result3 += first[7];
    }
    return result0 + result1 + result2 + result3;
}
```

Difference in logic

Scalar code

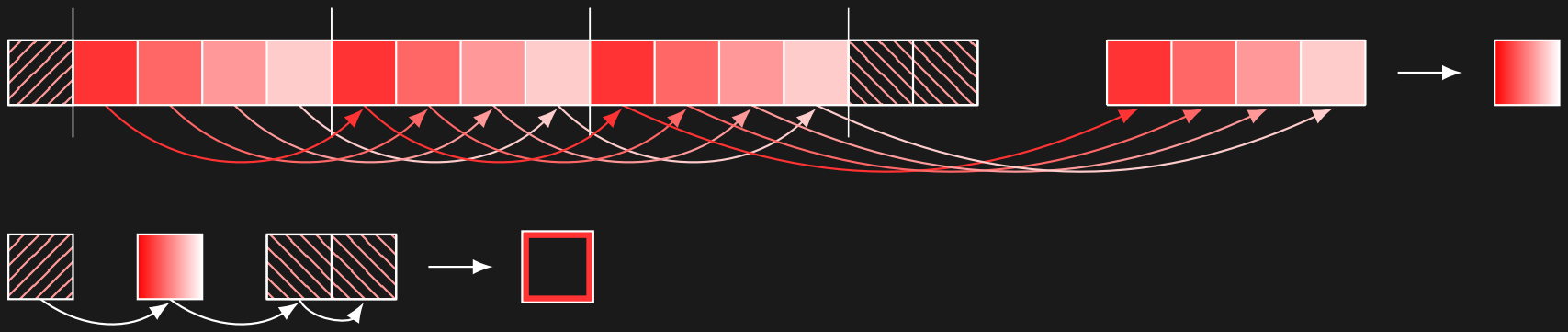
```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result = 0;
    for(; first != last; ++first)
        result += *first;
  return result;
}
```

Parallel logic

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result0 = 0,
             result1 = 0,
             result2 = 0,
             result3 = 0;
    for(; first != last; first += 8)
    {
        result0 += first[0];
        result1 += first[1];
        result2 += first[2];
        result3 += first[3];

        result0 += first[4];
        result1 += first[5];
        result2 += first[6];
        result3 += first[7];
    }
    return result0 + result1 + result2 + result3;
}
```

Vectorized Sum Algorithm



- Compute 4 adjacent sums in parallel
- Reduce the 4 partial sums to a single one
- Add with leading and trailing data to get result.

Vector intrinsics

Scalar code

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result = 0;
    for(; first != last; ++first)
        result += *first;
    return result;
}
```

SSE2 Intrinsics

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
  __m128i result = _mm_set1_epi32(0);
  for(; first != last; first += 8)
  {
    __m128i in = _mm_load_si128((__m128i*)first);
    __m128i lo = _mm_unpacklo_epi16(in, _mm_setzero_si128());
    __m128i hi = _mm_unpackhi_epi16(in, _mm_setzero_si128());

    result = _mm_add_epi32(result, lo);
    result = _mm_add_epi32(result, hi);
  }

  result = _mm_add_epi32(result,
    _mm_shuffle_epi32(result, _MM_SHUFFLE(1, 0, 3, 2))
  );
  result = _mm_add_epi32(result,
    _mm_shuffle_epi32(result, _MM_SHUFFLE(0, 1, 2, 3))
  );
  return _mm_cvtsi128_si32(result);
}
```

Assembly

General-purpose x86-64 Assembly

```
    xor     eax, eax
    cmp     rdi, rsi
    je      .L4
.L3:
    movzx   edx, WORD PTR [rdi]
    add     rdi, 2
    add     eax, edx
    cmp     rsi, rdi
    jne     .L3
.L4:
```

SSE2 Assembly

```
    cmp         rdi, rsi
    je          .L15
    pxor        xmm2, xmm2
    pxor        xmm3, xmm3
.L55:
    movdqa      xmm0, XMMWORD PTR [rdi]
    add         rdi, 16
    cmp         rsi, rdi
    movdqa      xmm4, xmm0
    movdqa      xmm1, xmm0
    punpcklwd   xmm4, xmm3
    punpckhwd   xmm1, xmm3
    movdqa      xmm0, xmm4
    paddd       xmm0, xmm2
    paddd       xmm0, xmm1
    movdqa      xmm2, xmm0
    jne         .L55
.L15:
```

Portable SIMD vectors (1)

Scalar code

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
  uint32_t result = 0;
  for(; first != last; ++first)
    result += *first;
  return result;
}
```

Vectorized by best vector width for output

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint32_t, 4> out_t;
    out_t result = 0;
    for(; first != last; first += 4)
    {
        result += out_t(first);
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Portable SIMD vectors (1)

Scalar code

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
  uint32_t result = 0;
  for(; first != last; ++first)
    result += *first;
  return result;
}
```

Vectorized by best vector width for output

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint32_t> out_t;
    out_t result = 0;
    for(; first != last; first += out_t::size())
    {
        result += out_t(first);
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Portable SIMD vectors (2)

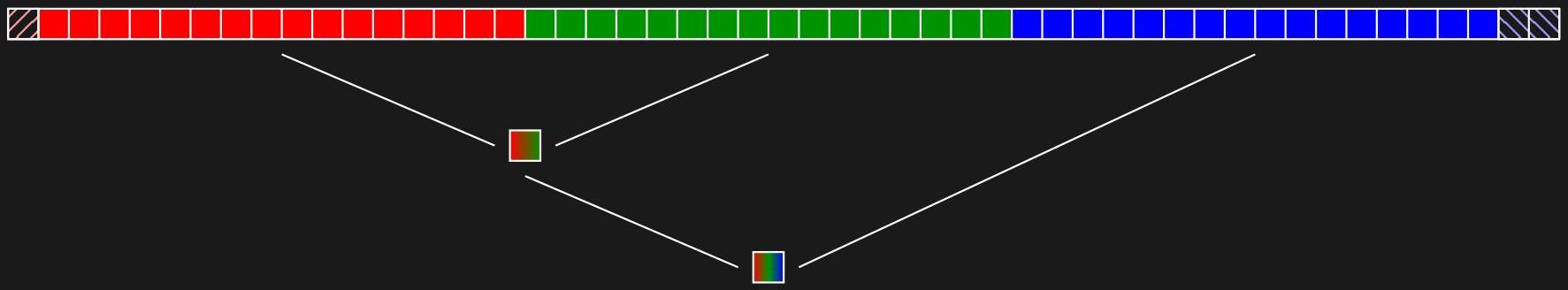
Vectorized by best vector width for input

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint16_t> in_t;
    typedef datapar<uint32_t, in_t::size()> out_t;
    out_t result = 0;
    for(; first != last; first += in_t::size())
    {
        result += out_t(in_t(first));
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Explicit promotion with high/low parts

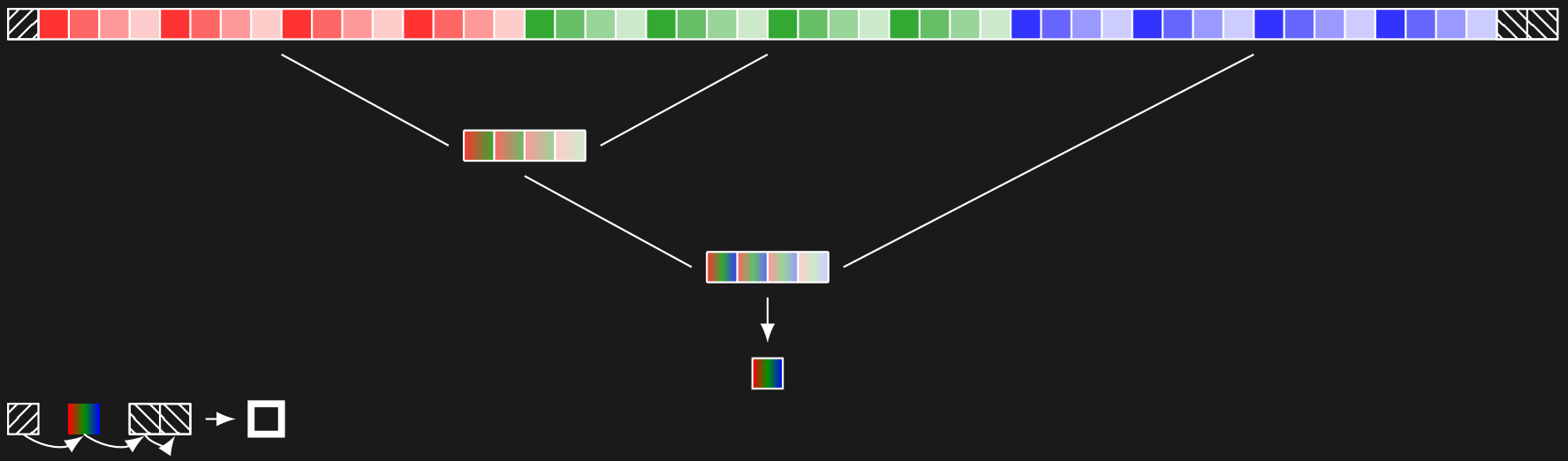
```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint16_t> in_t;
    typedef datapar<uint32_t, in_t::size()/2> out_t;
    out_t result = 0;
    for(; first != last; first += in_t::size())
    {
        in_t in(first);
        out_t lo, hi;
        tie(lo, hi) = datapar_cast<out_t>(in);
        result += lo;
        result += hi;
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Comparison with multi-core sum



- Split range into a number of subranges that get processed independently
- Preferably give full cache lines to each thread to avoid false sharing
- Accumulate the results pairwise to avoid global synchronization

Multi-core SIMD sum



- Combine the two approaches
- No need for each thread to deal with leading/trailing data if splitting on cache line boundary

Parallel Algorithm Skeletons

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(datapar, first, last,
        overload(
            [](uint16_t a, uint16_t b)
            {
                return a + b;
            },
            [](datapar<uint16_t> a, datapar<uint16_t> b)
            {
                return datapar<uint32_t, a.size()>(a)
                     + datapar<uint32_t, b.size()>(b);
            }
        )
    );
}
```

- Re-usable SIMD-enabled skeleton
- Handles leading and trailing data based on scalar overload
- Handles SIMD vector to scalar reduction with $log_2(N)$ algorithm

Parallel Algorithm Skeletons

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(par_datapar, first, last,
        overload(
            [](uint16_t a, uint16_t b)
            {
                return a + b;
            },
            [](datapar<uint16_t> a, datapar<uint16_t> b)
            {
                return datapar<uint32_t, a.size()>(a)
                     + datapar<uint32_t, b.size()>(b);
            }
        )
    );
}
```

- Re-usable SIMD-enabled skeleton
- Handles leading and trailing data based on scalar overload
- Handles SIMD vector to scalar reduction with $log_2(N)$ algorithm
- Also parallelized on multi-core

Parallel Algorithm Skeletons, Polymorphic

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(datapar, first, last,
        [](auto a, auto b)
        {
            return cast<uint32_t>(a) + cast<uint32_t>(b);
        }
    );
}
```

- C++14 polymorphic lambdas allow to express a single generic overload for both scalar and SIMD
- Goal is to make it possible to write SIMD-agnostic code as much as possible.

SIMD STL

Existing algorithms

- `std::for_each`, `std::transform`
- `std::reduce`, `std::transform_reduce`

Future work

- Generalizing to all STL algorithms
- `inclusive_scan`, `exclusive_scan`
- `std::sort`, `std::find` etc.

Many Sums

Multiple Directions for Parallelism

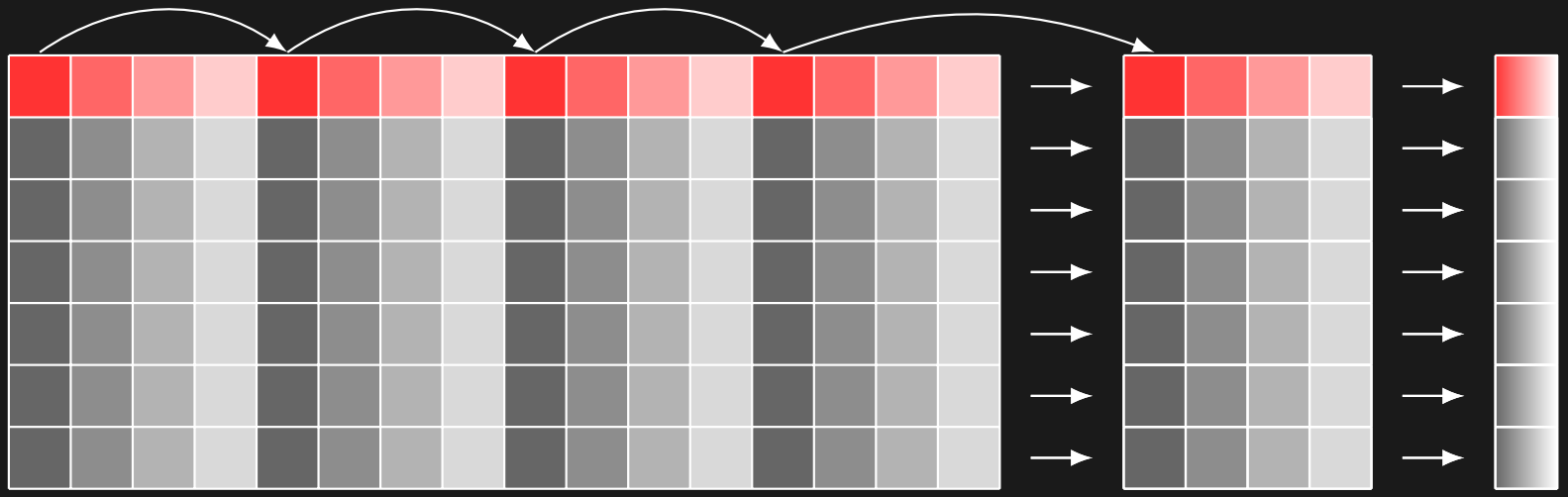
Two options

- Process a single sum in parallel, all lanes for one
- Process multiple sums in parallel, one per lane

Horizontal vs Vertical

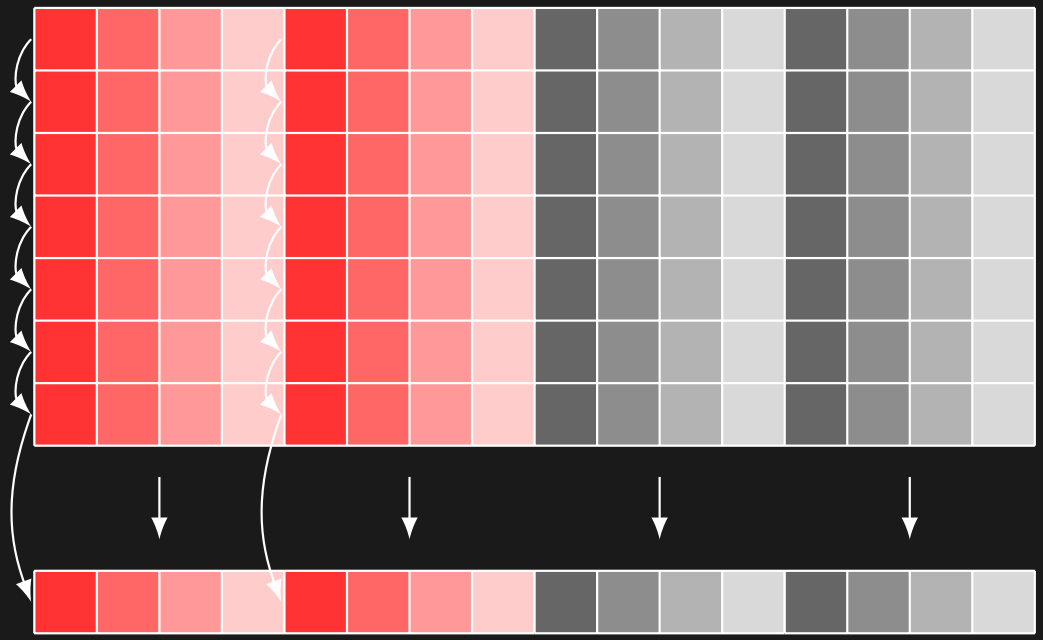
- Common theme when dealing with SIMD
- Best method depends on algorithm and layout
- Having multiple streams to process in lockstep doesn't require the operation itself to be intrinsically parallelizable

Horizontal Sum



- Process row-by-row
- Reorder the way data is processed
- Vector reduction + write in scalar mode

Vertical Sum



- Process multiple columns at the same time (tile to re-use cache)
- Logical order
- Vector operations only

Array-of-Structures vs Structures-of-Arrays

Side-by-side

AoS

```
struct Sample
{
  double portfolio_value;
  double external_flow;
};

double time_weighted_return(vector<Sample> const& samples)
{
  double ret = 1.;
  for(size_t i=1; i<samples.size(); ++i)
  {
    ret *= (samples[i].portfolio_value - samples[i].external_flow)
         / samples[i-1].portfolio_value;
  }
  return ret;
}
```

SoA

```
struct Samples
{
  vector<double> portfolio_values;
  vector<double> external_flows;
};

double time_weighted_return(Samples const& samples)
{
  double ret = 1.;
  for(size_t i=1; i<samples.size(); ++i)
  {
    ret *= (samples.portfolio_values[i] - samples.external_flows[i])
         / samples.portfolio_values[i-1];
  }
  return ret;
}
```

- AoS more natural, humans organize things into objects
- SoA makes it easier to vectorize, contiguous memory makes load/store easy
- AoS is more compact as it puts things into the same cache line

Towards datapar of structures

```
struct Sample
{
  double portfolio_value;
  double external_flow;
};
```

```
template<>
struct datapar<Sample>
{
  datapar<double> portfolio_value;
  dapatar<double> external_flow;
};
```

- For better use of SIMD, best to adapt structures of primitive types into structures of datapar types
- This could be automated with static reflection coming in a future C++ TS
- Works today with tuples and Boost.Fusion sequences in Boost.SIMD
- Automatic deinterleaving/interleaving on load/store and operator[] access

Applications and Benchmarks

Basic functions

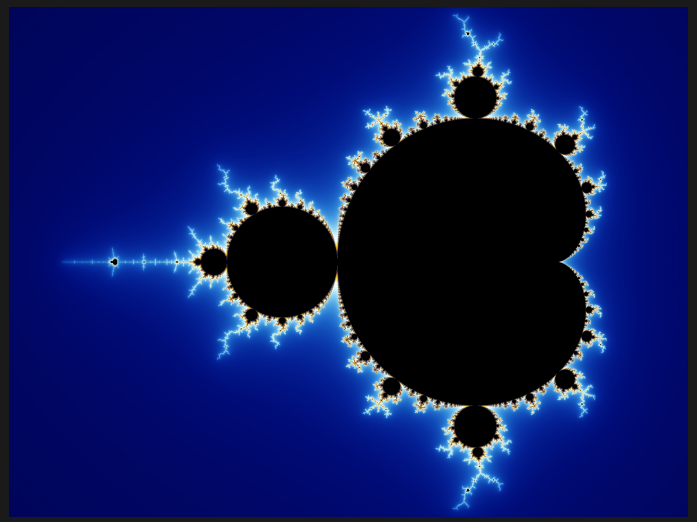
Single-precision trigonometric functions

Architecture : Core i7 SandyBridge, AVX in cycles/value

Function	Interval	std	Scalar	SIMD
exp	$[-10, 10]$	46	38	7
log	$[-10, -10]$	42	37	5
asin	$[-1, 1]$	40	35	13
cos	$[-20\pi, 20\pi]$	66	47	6
fast_cos	$[-\pi/4, \pi/4]$	32	9	1.3

Fractal generator

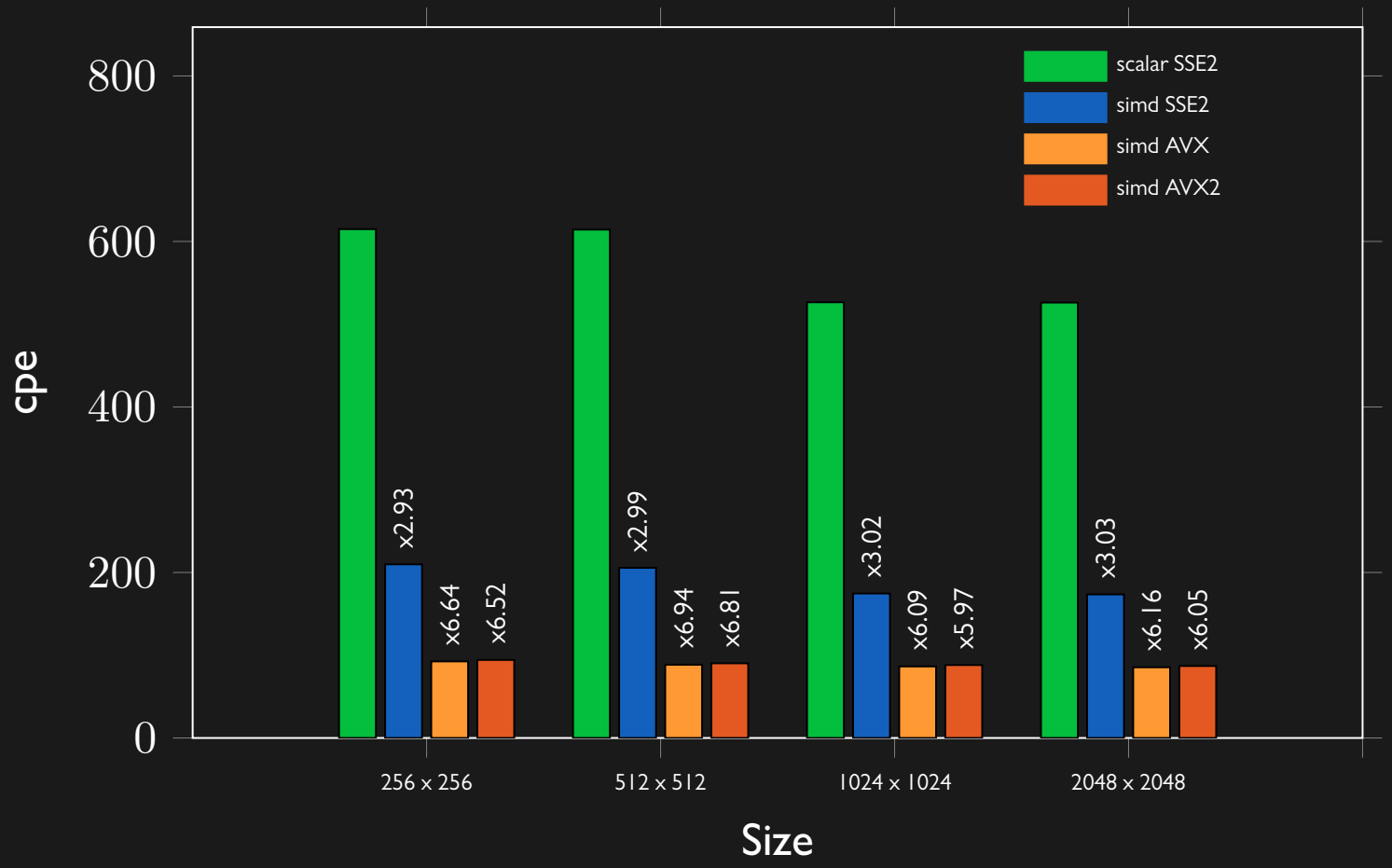
- Generates a fractal image by evaluationg a complex function
- Compute-bound application
- Challenge: Amount of work depends on the pixel



Fractal generator

```
template<class T>
auto julia(datapar<T> a, datapar<T> b) -> datapar<int, a.size()>
{
  datapar<int, a.size()> iter(0);
  mask<int, a.size()> mask;
  int i = 0;
  T x, y;
  do
  {
    T x2 = x * x;
    T y2 = y * y;
    T xy = s_t(2) * x * y;
    x = x2 - y2 + a;
    y = xy + b;
    mask = x2 + y2 < T(4);
    where(mask, iter)++;
  }
  while(any_of(mask) && i++ < 256);
  return iter;
}
```

Fractal generator



Movement detection

- Sigma-Delta algorithm by Manzanera et al.
- Mono-modal approach based on background extraction
- Models a variation of intensity by a per-pixel gaussian
- Challenge: Very weak arithmetic intensity

Movement detection

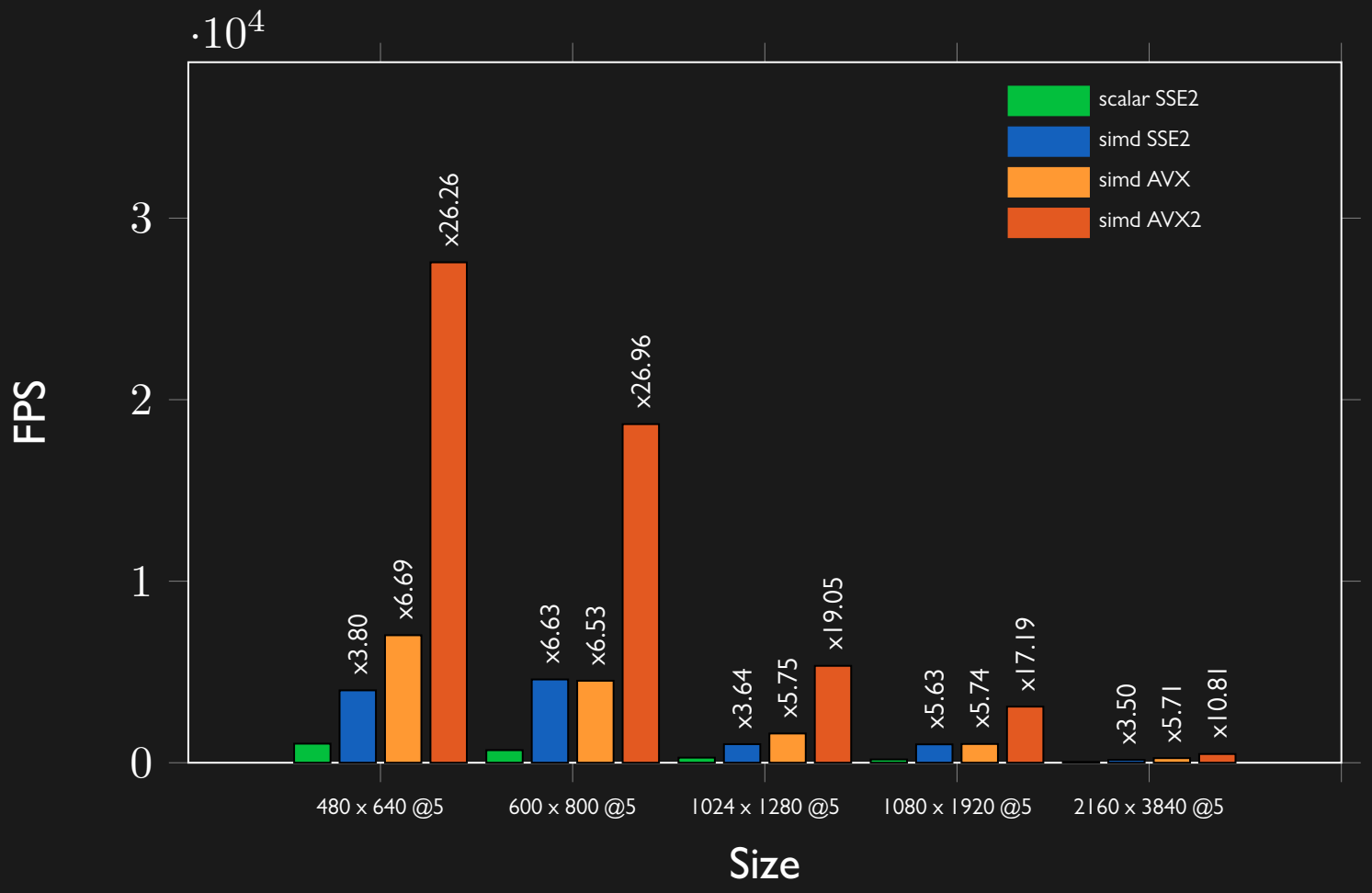
```
template<typename T>
T sigma_delta(T& bkg, T const& frm, T& var)
{
  where(bkg > frm, bkg)--;
  where(bkg < frm, bkg)++;

  // avoid overflow with max/min and satured arithmetic
  T dif = max(bkg, frm) - min(bkg, frm);
  T mul = adds(dif, adds(dif, dif));

  auto d = dif != T(0);
  where(d && var > mul, var)--;
  where(d && var < mul, var)++;

  T result = T(1);
  where(dif < var, result, 0);
  return result;
}
```

Movement detection



Sparse tridiagonal system solver

Algorithm 1: Thomas algorithm: solve Ax=s.

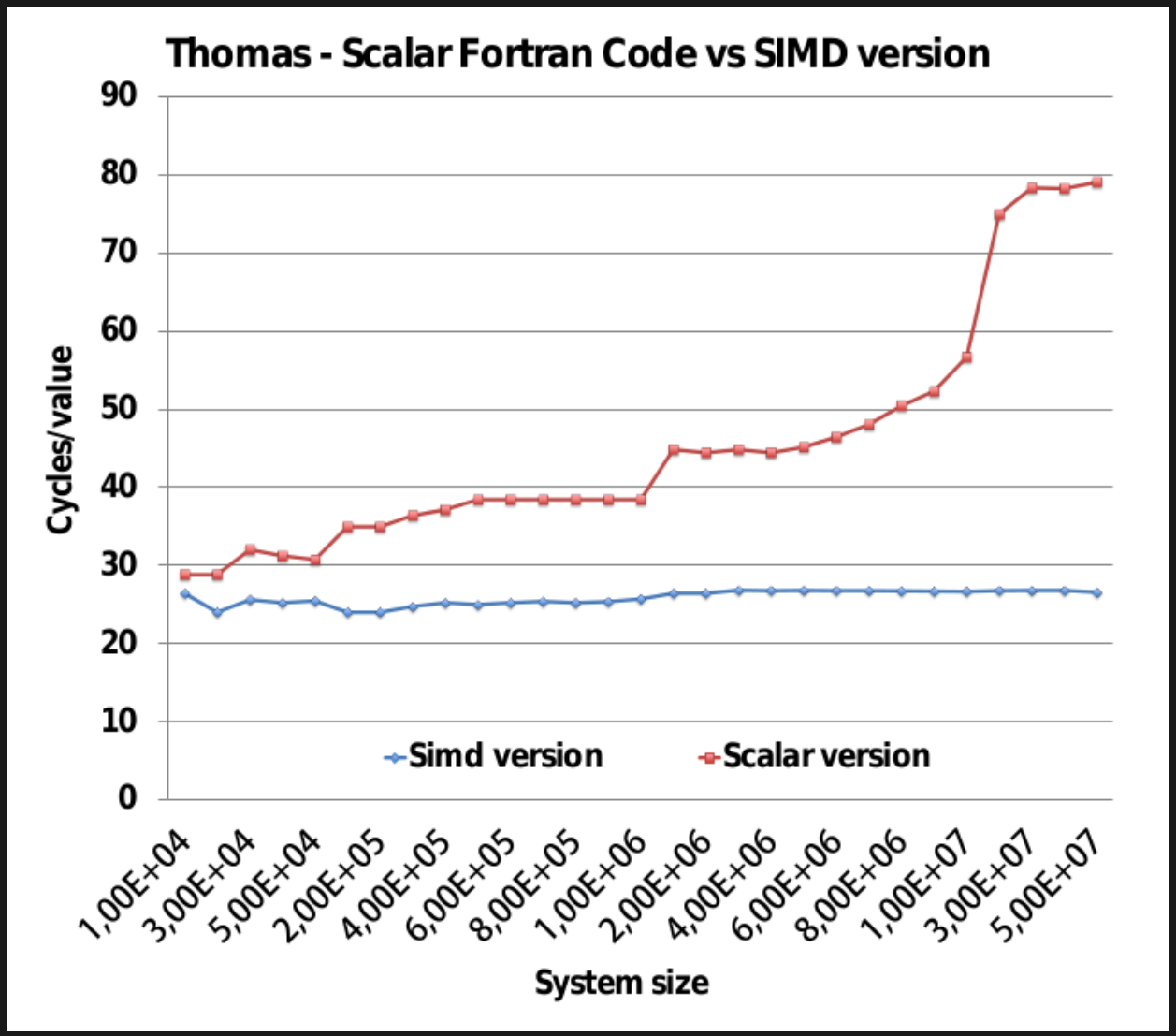
input : A tridiagonal system.

- a[n] : sub-diagonal
- b[n] : main diagonal
- c[n] : sup-diagonal
- s[n] : right hand side

```
begin
    Forward elimination:
    for i=2,...,n do
        δ = a[i]c[i − 1]/b[i − 1]
        b[i] = b[i] − δ
        δ_s = a[i]s[i − 1]/b[i − 1]
        s[i] = s[i] − δ_s
    end
    Backward substitution: x[n] = s[n]/b[n]
    for i=n-1,...,1 do
        x[i] = s[i] − c[i]x[i + 1]/b[i]
    end
end
```

- Solves $Ax = b$ with A sparse
- Application: fluid mechanics
- Challenge: vectorize despite the sparse aspect
- Solution: Generalized shuffles for re-compactification

Sparse tridiagonal system solver



Black-Scholes Options Pricing

- Pricing model for options

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

- Closed-form solution for C (call) and P (put):

$$C = cnd(d_1)S - cnd(d_2)Ke^{-rT}$$

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T\right]$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

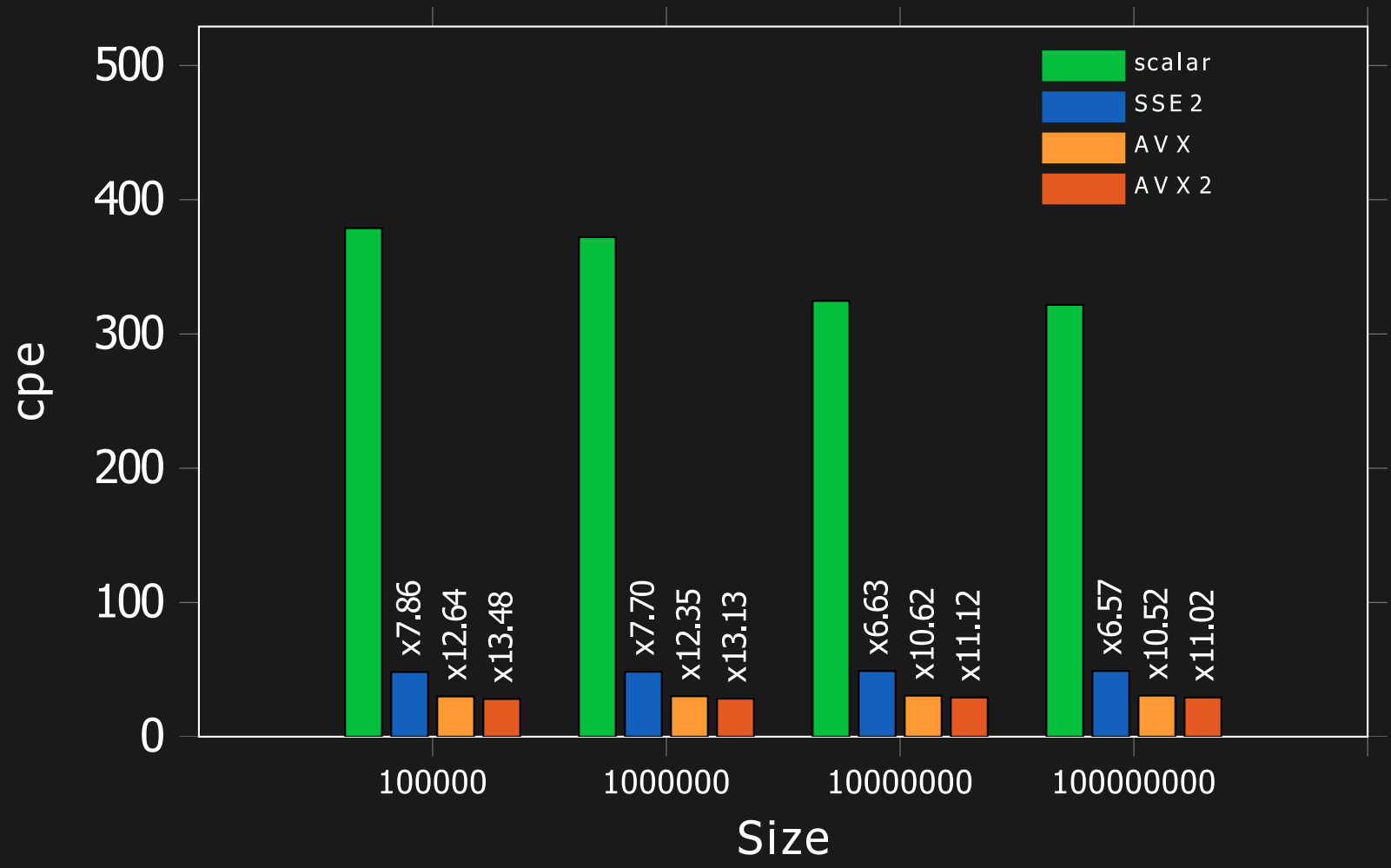
$$P = cnd(-d_2)Ke^{-rT} - cnd(-d_1)S$$

- Challenge: Embarrassingly parallel, but usage of several transcendental functions with high-latency and register pressure.

Black-Scholes Options Pricing

```
// returns (call, put)
template<class Ty>
pair<Ty, Ty> blackscholes(Ty S, Ty X, Ty T, Ty r, Ty v)
{
    Ty sqrtT = sqrt(T);
    Ty d1 = log(S / X) + ( sqr(v) * 0.5f + r ) * T /(v*sqrtT);
    Ty d2 = d1 - v * sqrtT;
    Ty expRT = exp(-r*T);
    Ty normcdf1 = normcdf(d1);
    Ty normcdf2 = normcdf(d2);
    return make_pair( S * normcdf1 - X * expRT * normcdf2,
                      X * expRT * (1.0f - normcdf2) - S * (1.0f - normcdf1)
                    );
}
```

Black-Scholes Options Pricing



Conclusion

Conclusion

SIMD

- SIMD is an instruction-level parallelism that is underused
- Difficult to make good use of it without programming it explicitly
- Can lead to significant performance improvements when used, applies to a variety of domains

Standard SIMD support in C++

- A unified interface, no need to deal with dozens of variants
- Both a low-level (vector types) and high-level (algorithms) API
- Experimental open-source implementations available: Boost.SIMD and Vc
- Parallelism TS 2 in 2017 and C++19

Questions?