SIMD in Scientific Computing

Tim Haines

(terminal) PhD Candidate
University of Wisconsin-Madison
Department of Astronomy

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

State of Tree(PM)-based N-Body solvers in Astronomy

	SPH?	GPU?	Xeon Phi?
Gadget2 (G2X)	Yes	Yes	No
Gadget3	Yes	No	No
Bonsai (1&2)	No	Yes	No
PKDGrav3	No	Yes	No
GOTHIC	No	Yes	No
2HOT	Yes	Yes	No
HACC	Yes	Yes	Yes (implicit)
ChaNGa	Yes	Yes	Soon!

Compute Density

Machine	nCores	Rmax TFLOPS	Power (kW)	Type
Sunway TaihuLight	10,649,000	93,014.6	15,371	Custom RISC
Tianhe-2	3,120,000	33,862.7	17,808	Xeon Phi
Piz Daint	361,760	19,590.0	2272	Tesla P100
Gyoukou	19,860,000	19,135.8	1,350	PEZY-SC
Titan	560,640	17,590.0	8,209	Kepler

SIMD Support in ChaNGa

- Supported architectures
  - SSE2 (single and double precision)
  - AVX (double only)
- Very invasive
  - Lots of user-level macros

```
SSELoad(SSEcosmoType, activeParticles, idx, ->position.x)

SSEcosmoType(
   activeParticles[idx+0]->position.x,
   activeParticles[idx+1]->position.x,
   activeParticles[idx+2]->position.x,
   activeParticles[idx+3]->position.x);
```

Wishlist

- Single source
  - Write once, use anywhere
- Simple to use
  - User only specifies precision (via template param)
  - All other details are handled by library
- Specifically catered to scientific code style in N-body solvers
  - Which means...

AoS to SoA

- AoS dominates
  - Easy to reason about in particle-based N-body solvers
  - “natural” representation
- Cons
  - Terrible data locality
    - Most particles are >64 bytes!
  - Compiler will not auto-vectorize
- Permeates entire codebase
  - Very hard to transition to SIMD-friendly SoA
  - Need something to help us out
    - Explicit SIMD vectorization

What is “single source?”

- Easier to answer what is *not* single source
- ISA-specific name
  - sse_float, __m256
- Data type or vector size in name
  - simd_float, fvec8
- No overloaded operators
  - sse_add_float4(…)
- I want to write once, use anywhere
  - simd<T> x;

State of SIMD Libraries

	Single-source	Easy AoS->SoA	KNL
Vc	No		
Agner Fog	No		
libsimd	No		
The simd library	No		
GROMACS	No		
Intel C++ SIMD Classes	No		
QuickVec	No		
dimsum	Yes	No	
libsimdpp	Yes	No	
Boost.SIMD	Yes	Yes	No**

DIY SIMD in C++

- Two key ingredients:
  - Type traits
  - Tag dispatch
- These are C++98 features!
- A sprinkle of TMP to fix the edge cases

DIY SIMD in C++

```
template <typename T>
struct cksimd {
    using value_type = T;
    using category = typename simd_category<value_type>::type;
    using simd_t = typename simd_type<value_type>::type;
    static constexpr auto size = sizeof(simd_t) / sizeof(value_type);
    using bool_t = typename bool_type<value_type>::type;

    simd_t val;

    cksimd operator +(cksimd x) const { return add(val, x.val, category()); }
};
```

DIY SIMD in C++

- Which `add` gets called?

```
#if defined(__AVX512F__) && defined(__AVX512ER__) && defined(__AVX512PF__) &&
defined(__AVX512CD__) && !defined(CMK_SIMD_DISABLE_KNL)
#    include "cksimd_knl.h"
#elif defined(__AVX__) && !defined(CMK_SIMD_DISABLE_AVX)
#    include "cksimd_avx.h"
#elif defined(__SSE4_2__) && !defined(CMK_SIMD_DISABLE_SSE)
#    include "cksimd_sse.h"
#else
#    include "cksimd_scalar.h"
#endif
```

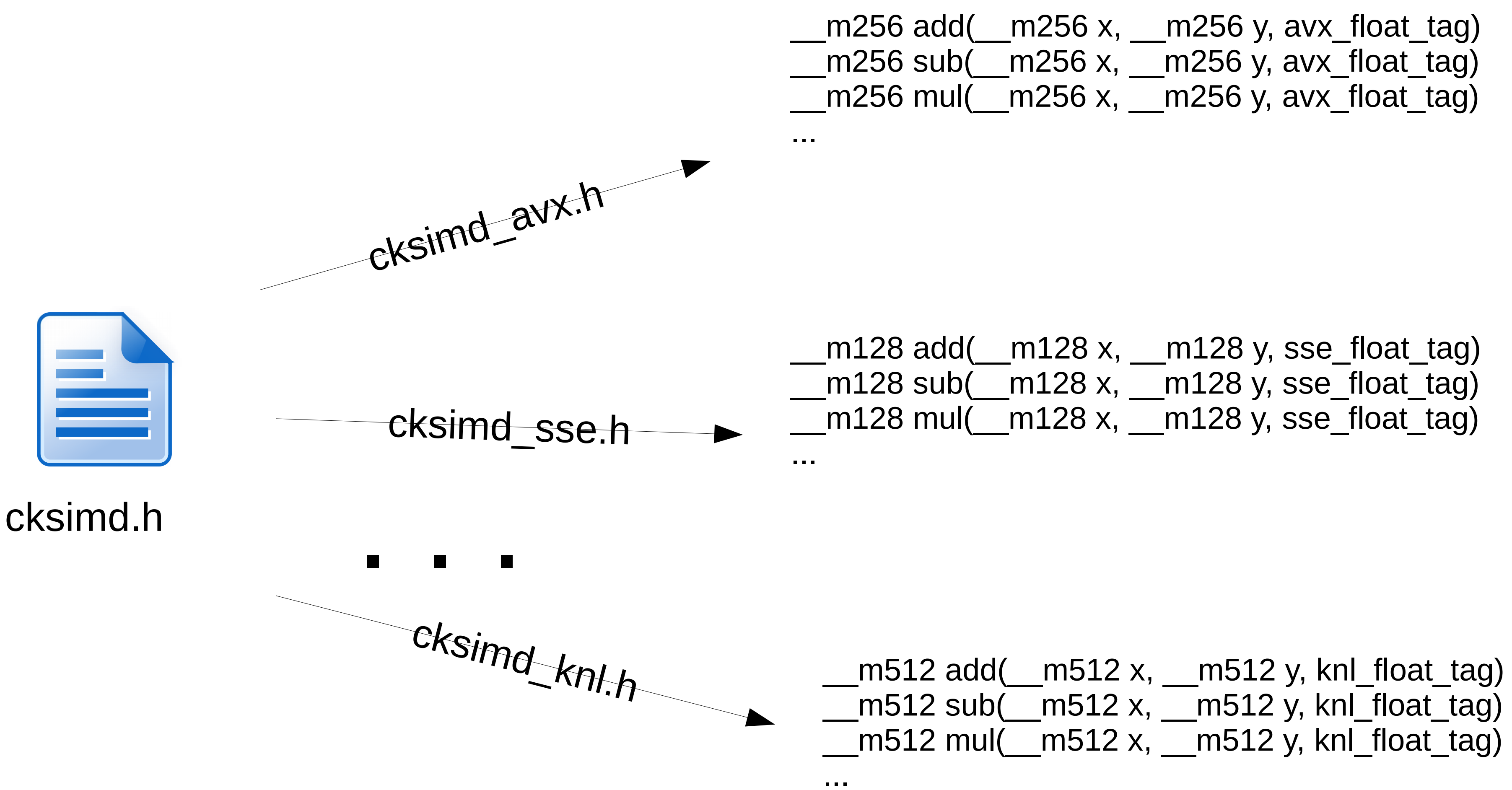
- The __XXX__ macros are set by the compiler
  - e.g., g++ -mavx

DIY SIMD in C++

- g++ -mavx
  - In cksimd_avx.h, we have

```
static inline __m256 add(__m256 x, __m256 y, avx_float_tag) {
    return _mm256_add_ps(x, y);
}
static inline __m256d add(__m256d x, __m256d y, avx_double_tag) {
    return _mm256_add_pd(x, y);
}
```

DIY SIMD in C++


__m256 add(__m256 x, __m256 y, avx_float_tag)
__m256 sub(__m256 x, __m256 y, avx_float_tag)
__m256 mul(__m256 x, __m256 y, avx_float_tag)
...
cksimd_avx.h
cksimd_sse.h
__m128 add(__m128 x, __m128 y, sse_float_tag)
__m128 sub(__m128 x, __m128 y, sse_float_tag)
__m128 mul(__m128 x, __m128 y, sse_float_tag)
...
cksimd.h
. . .
cksimd_knl.h
__m512 add(__m512 x, __m512 y, knl_float_tag)
__m512 sub(__m512 x, __m512 y, knl_float_tag)
__m512 mul(__m512 x, __m512 y, knl_float_tag)
...

Simplify User Code

```
#ifndef __SSEDEFS_H__
#define __SSEDEFS_H__

#include "cosmoType.h"

#if  CMK_USE_AVX
        #if !defined(__AVX__)
                #undef CMK_USE_AVX
                #define CMK_USE_AVX 0
        #else
                #warning "using␣AVX"
        #endif
#endif

#if CMK_USE_SSE2 && !defined(__SSE2__)
        #undef CMK_USE_SSE2
        #define CMK_USE_SSE2 0
#endif

#if CMK_USE_AVX || CMK_USE_SSE2
        #define CMK_SSE 1
#endif

#if CMK_USE_AVX
        #ifdef COSMO_FLOAT
                #error "single-precision␣AVX␣is␣not␣supported"
        #else
                #include "SSE-Double.h"
                #define SSE_VECTOR_WIDTH 4
                #define FORCE_INPUT_LIST_PAD 3
                typedef SSEDouble SSEcosmoType;
                #define SSELoad(where, arr, idx, field) where(arr[idx]field, arr[idx+1]field, arr[idx+2]field, arr[idx+3]field)
                #define SSEStore(what, arr, idx, field) { \
                  double p[4];  \
                  storeu(p, what); \
                  arr[idx]field = p[0]; \
                  arr[idx+1]field = p[1]; \
                  arr[idx+2]field = p[2]; \
                  arr[idx+3]field = p[3]; \
                }
                enum {cosmoMask=0xf};
        #endif
#elif CMK_USE_SSE2
        #ifdef COSMO_FLOAT
                #define SSE_COSMO_FLOAT
                #if defined(__SSE2__)
                        #include "SSE-Float.h"
                        #define SSE_VECTOR_WIDTH 4
                        #define FORCE_INPUT_LIST_PAD 3
                        typedef SSEFloat SSEcosmoType;
                        #define SSELoad(where, arr, idx, field) where(arr[idx]field, arr[idx+1]field, arr[idx+2]field, arr[idx+3]field)
                        #define SSEStore(what, arr, idx, field) { \
                          float p[4]; \
                          storeu(p, what); \
                          arr[idx]field = p[0]; \
                          arr[idx+1]field = p[1]; \
                          arr[idx+2]field = p[2]; \
                          arr[idx+3]field = p[3]; \
                        }
                        enum {cosmoMask=0xf};
                #else
                      #error("SSE␣not␣available");
                #endif
        #else
                #if defined(__SSE2__) && !defined(SSE_COSMO_FLOAT)
                        #include "SSE-Double.h"
                        #define SSE_VECTOR_WIDTH 2
                        #define FORCE_INPUT_LIST_PAD 1
                        typedef SSEDouble SSEcosmoType;
                        #define SSELoad(where, arr, idx, field) where(arr[idx]field, arr[idx+1]field)
                        #define SSEStore(what, arr, idx, field) { \
                          storel(&arr[idx]field, what); \
                          storeh(&arr[idx+1]field, what); \
                        }
                        enum {cosmoMask=0x3};
                #else
                      #error("SSE␣not␣available");
                #endif
        #endif
#endif

#endif


#if !CMK_SSE
inline
void SPLINEQ(cosmoType invr,cosmoType r2,cosmoType twoh,cosmoType& a,
             cosmoType& b,cosmoType& c,cosmoType& d)
{
  cosmoType u,dih,dir=(invr);
  if ((r2) < (twoh)*(twoh)) {
    dih = COSMO_CONST(2.0)/(twoh);
    u = dih/dir;
    if (u < COSMO_CONST(1.0)) {
      a = dih*(COSMO_CONST(7.0)/COSMO_CONST(5.0)
               - COSMO_CONST(2.0)/COSMO_CONST(3.0)*u*u
               + COSMO_CONST(3.0)/COSMO_CONST(10.0)*u*u*u*u
               - COSMO_CONST(1.0)/COSMO_CONST(10.0)*u*u*u*u*u);
      b = dih*dih*dih*(COSMO_CONST(4.0)/COSMO_CONST(3.0)
                       - COSMO_CONST(6.0)/COSMO_CONST(5.0)*u*u
                       + COSMO_CONST(1.0)/COSMO_CONST(2.0)*u*u*u);
      c = dih*dih*dih*dih*dih*(COSMO_CONST(12.0)/COSMO_CONST(5.0)
                               - COSMO_CONST(3.0)/COSMO_CONST(2.0)*u);
      d = COSMO_CONST(3.0)/COSMO_CONST(2.0)*dih*dih*dih*dih*dih*dih*dir;
    }
    else {
      a = COSMO_CONST(-1.0)/COSMO_CONST(15.0)*dir
        + dih*(COSMO_CONST(8.0)/COSMO_CONST(5.0)
               - COSMO_CONST(4.0)/COSMO_CONST(3.0)*u*u + u*u*u
               - COSMO_CONST(3.0)/COSMO_CONST(10.0)*u*u*u*u
               + COSMO_CONST(1.0)/COSMO_CONST(30.0)*u*u*u*u*u);
      b = COSMO_CONST(-1.0)/COSMO_CONST(15.0)*dir*dir*dir
        + dih*dih*dih*(COSMO_CONST(8.0)/COSMO_CONST(3.0) - COSMO_CONST(3.0)*u
                       + COSMO_CONST(6.0)/COSMO_CONST(5.0)*u*u
                       - COSMO_CONST(1.0)/COSMO_CONST(6.0)*u*u*u);
      c = COSMO_CONST(-1.0)/COSMO_CONST(5.0)*dir*dir*dir*dir*dir
        + COSMO_CONST(3.0)*dih*dih*dih*dih*dir
        + dih*dih*dih*dih*dih*(COSMO_CONST(-12.0)/COSMO_CONST(5.0)
                               + COSMO_CONST(1.0)/COSMO_CONST(2.0)*u);
      d = -dir*dir*dir*dir*dir*dir*dir
        + COSMO_CONST(3.0)*dih*dih*dih*dih*dir*dir*dir
        - COSMO_CONST(1.0)/COSMO_CONST(2.0)*dih*dih*dih*dih*dih*dih*dir;
    }
  }
  else {
    a = dir;
    b = a*a*a;
    c = COSMO_CONST(3.0)*b*a*a;
    d = COSMO_CONST(5.0)*c*a*a;
  }
}
#else
inline
void SPLINEQ(SSEcosmoType invr, SSEcosmoType r2, SSEcosmoType twoh,
             SSEcosmoType& a, SSEcosmoType& b,
             SSEcosmoType& c, SSEcosmoType& d)
{
  SSEcosmoType dir;
  dir = invr;
  SSEcosmoType select0 = r2 < twoh * twoh;
  int compare0 = movemask(select0);
  // make common case fast, at the cost of some redundant code
  //   in the infrequent case
  if (!compare0) {
    a = dir;
    b = a*a*a;
    c = COSMO_CONST(3.0)*b*a*a;
    d = COSMO_CONST(5.0)*c*a*a;
  }
  else {
    SSEcosmoType u, dih;
    SSEcosmoType select1;
    SSEcosmoType a0,b0,c0,d0,a1,b1,c1,d1,a2,b2,c2,d2;
    int compare1;

    dih = COSMO_CONST(2.0)/twoh;
    u = dih/dir;

    if ((~compare0) & cosmoMask) {
      a0 = dir;
      b0 = a0*a0*a0;
      c0 = COSMO_CONST(3.0)*b0*a0*a0;
      d0 = COSMO_CONST(5.0)*c0*a0*a0;
    }

    select1 = u < COSMO_CONST(1.0);
    compare1 = movemask(select1);
    if (compare1) {
      a1 = dih*(COSMO_CONST(7.0)/COSMO_CONST(5.0)
                - COSMO_CONST(2.0)/COSMO_CONST(3.0)*u*u
                + COSMO_CONST(3.0)/COSMO_CONST(10.0)*u*u*u*u
                - COSMO_CONST(1.0)/COSMO_CONST(10.0)*u*u*u*u*u);
      b1 = dih*dih*dih*(COSMO_CONST(4.0)/COSMO_CONST(3.0)
                        - COSMO_CONST(6.0)/COSMO_CONST(5.0)*u*u
                        + COSMO_CONST(1.0)/COSMO_CONST(2.0)*u*u*u);
      c1 = dih*dih*dih*dih*dih*(COSMO_CONST(12.0)/COSMO_CONST(5.0)
                                - COSMO_CONST(3.0)/COSMO_CONST(2.0)*u);
      d1 = COSMO_CONST(3.0)/COSMO_CONST(2.0)*dih*dih*dih*dih*dih*dih*dir;
    }
    if ((~compare1) & cosmoMask) {
      a2 = COSMO_CONST(-1.0)/COSMO_CONST(15.0)*dir
        + dih*(COSMO_CONST(8.0)/COSMO_CONST(5.0)
               - COSMO_CONST(4.0)/COSMO_CONST(3.0)*u*u + u*u*u
               - COSMO_CONST(3.0)/COSMO_CONST(10.0)*u*u*u*u
               + COSMO_CONST(1.0)/COSMO_CONST(30.0)*u*u*u*u*u);
      b2 = COSMO_CONST(-1.0)/COSMO_CONST(15.0)*dir*dir*dir
        + dih*dih*dih*(COSMO_CONST(8.0)/COSMO_CONST(3.0) - COSMO_CONST(3.0)*u
                       + COSMO_CONST(6.0)/COSMO_CONST(5.0)*u*u
                       - COSMO_CONST(1.0)/COSMO_CONST(6.0)*u*u*u);
      c2 = COSMO_CONST(-1.0)/COSMO_CONST(5.0)*dir*dir*dir*dir*dir
        + COSMO_CONST(3.0)*dih*dih*dih*dih*dir
        + dih*dih*dih*dih*dih*(COSMO_CONST(-12.0)/COSMO_CONST(5.0)
                               + COSMO_CONST(1.0)/COSMO_CONST(2.0)*u);
      d2 = -dir*dir*dir*dir*dir*dir*dir
        + COSMO_CONST(3.0)*dih*dih*dih*dih*dir*dir*dir
        - COSMO_CONST(1.0)/COSMO_CONST(2.0)*dih*dih*dih*dih*dih*dih*dir;
    }

    a = andnot(select0, a0)
      | (select0 & ((select1 & a1) | andnot(select1, a2)));
    b = andnot(select0, b0)
      | (select0 & ((select1 & b1) | andnot(select1, b2)));
    c = andnot(select0, c0)
      | (select0 & ((select1 & c1) | andnot(select1, c2)));
    d = andnot(select0, d0)
      | (select0 & ((select1 & d1) | andnot(select1, d2)));
  }
}
#endif
```

```
template <typename T>
void SPLINE(cksimd<T> r2, cksimd<T> twoh, cksimd<T> &a, cksimd<T> &b) {
        if (all(r2 >= (twoh * twoh))) {
                a = rsqrt(r2);
                b = a * a * a;
                return;
        }

        const cksimd<T> r = sqrt(r2);
        {
                auto mask = r >= twoh;
                a.blend(rsqrt(r2), mask);
                b.blend(a * a * a, mask);
        }

        auto const dih = static_cast<T>(2.0) / twoh;
        auto const u = r * dih;

        auto const u2 = u * u;
        auto const u3 = u2 * u;
        auto const u4 = u2 * u2;
        auto const u5 = u4 * u;
        auto const dih3 = (dih * dih) * dih;

        const auto mask = u < T(1.0);
        if (any(mask)) {
                const T c1 = 7.0 /  5.0, c2 = 2.0 /  3.0,
                                c3 = 3.0 / 10.0, c4 = 1.0 / 10.0,
                                c5 = 4.0 /  3.0, c6 = 6.0 /  5.0,
                                c7 = 1.0 /  2.0;

                const auto a1 = dih  * (c1 - c2 * u2 + c3 * u4 - c4 * u5);
                const auto b1 = dih3 * (c5 - c6 * u2 + c7 * u3);

                a.blend(a1, mask);
                b.blend(b1, mask);
        }

        if (any(!mask)) {
                auto const dir = rsqrt(r2), dir3 = (dir * dir) * dir;

                const T c1 = -1.0 / 15.0, c2 = 8.0 /  5.0,
                                c3 =  4.0 /  3.0, c4 = 3.0 / 10.0,
                                c5 =  1.0 / 30.0, c6 = 8.0 /  3.0,
                                c7 =  3.0               , c8 = 6.0 /  5.0,
                                c9 =  1.0 /  6.0;

                const auto a1 = c1 * dir  + dih  * (c2 - c3 * u2 + u3 - c4 * u4 + c5 * u5);
                const auto b1 = c1 * dir3 + dih3 * (c6 - c7 * u  + c8 * u2 - c9 * u3);

                a.blend(a1, !mask);
                b.blend(b1, !mask);
        }
}
```

Example

```
template <typename T>
struct particle { T x, y, z; };

template <typename T>
void test_gravity() {
     particle particles[10];

     for(auto chunk : simd::load(particles, std::end(particles)) {
          cksimd<T> x, y, z;
          auto end = x.pack(chunk, [](particle<T> const& p){ return p.x; });
          y.pack(chunk, [](particle<T> const& p){ return p.y; });
          z.pack(chunk, [](particle<T> const& p){ return p.z; });

          auto const r2 = x*x + y*y + z*z;
          auto const f = 3.14159 / sqrt(r2);
          x += f;
          y *= f;
          z += 2.0 * f;

          x.unpack(chunk, [](particle<T>& p, T val) { p.x = val; });
          y.unpack(chunk, [](particle<T>& p, T val) { p.y = val; });
          z.unpack(chunk, [](particle<T>& p, T val) { p.z = val; });
     }
}
```

Example (v2)

```
template <typename T>
struct particle { T x, y, z; };

template <typename T>
std::tuple<cksimd<T>,cksimd<T>,cksimd<T>>
pack_coords(simd::range const& r) {
     cksimd<T> x, y, z;
     auto end = x.pack(chunk, [](particle<T> const& p){ return p.x; });
     y.pack(chunk, [](particle<T> const& p){ return p.y; });
     z.pack(chunk, [](particle<T> const& p){ return p.z; });

     return std::make_tuple(x, y, z);
}
template <typename T>
void unpack_coords(simd::range& chunk, cksimd<T> x, cksimd<T> y, cksimd<T> z) {
     x.unpack(chunk, [](particle<T>& p, T val) { p.x = val; });
     y.unpack(chunk, [](particle<T>& p, T val) { p.y = val; });
     z.unpack(chunk, [](particle<T>& p, T val) { p.z = val; });
}
template <typename T>
void test_gravity() {
     particle particles[10];

     for(auto chunk : simd::load(particles, std::end(particles)) {
          cksimd<T> x, y, z;
          std::tie(x, y, z) = pack_coords(chunk);  // auto [x,y,z] = .. in C++17

          auto const r2 = x*x + y*y + z*z;
          auto const f = 3.14159 / sqrt(r2);
          x += f;
          y *= f;
          z += 2.0 * f;

          unpack_coords(chunk, x, y, z);
     }
}
```

A Bit of Optimization

- auto const f = 3.14159 / sqrt(r2);
- This is a fairly common operation in ChaNGa's gravity kernels, so we apply a little more work to make this fast
- X / sqrt(Y) → X * rsqrt(Y)
  - rsqrt is a single instruction
    - Terrible accuracy
  - Use Newton-Raphson in SP to get ~1 ulp accuracy
    - Speedup is roughly equal on all tested arches compared to c-lib sqrt+div
  - 5^{th}-order polynomial in DP to get ~1.5 ulps accuracy
    - Speedup is between 10% (Skylake) and 3x (Sandybridge) compared to sqrt+div

Future Work

- More math
  - Log, exp, sin, cos, erf
- Better algorithms
  - Make transforms easier
  - Iterator wrappers to better integrate with STL containers/algorithms
  - Better AoS → SoA
    - Improve codegen for pack operation
- Blue Gene/Q and /P backends
  - HELP WANTED
- Full implementation of ChaNGa's gravity kernels
  - SPH kernels
- Code cleanup for public release