Corrfunc: Blazing fast correlation functions with SIMD Intrinsics

Dr. Manodeep Sinha
ASTRO 3D Centre of Excellence, Swinburne

Repo: github.com/manodeep/Corrfunc/
Collaborators: Lehman Garrison
Contributors: Andrew Hearin, Nick Hand
ΛCDM Picture: Galaxies live in Halos

LasDamas Simulations, XSEDE/TACC
ACDM Picture: Galaxies live in Halos

Baryon Physics (messy)
Quantifying the Galaxy Distribution
What is a Correlation Function?

- Measures the **excess probability** of finding a pair at some separation.

Groth & Peebles, 1977
Galaxy Clustering on different scales

Gravity + Cosmology

Galaxy formation physics

Masjedi et al. (2006)
Correlation functions are fundamental to understand how galaxies populate halos.
for(int i=0;i<N1;i++) {
    for(int j=0;j<N2;j++) {
        double dist = @distance_metric@(point[i], point[j]);
        if(dist < mindist || dist >= maxdist) {
            continue;
        }

        int ibin = @dist_to_bin_index@(dist);
        numpairs[ibin]++;
    }
}
Code for a Correlation Function

```c
for(int i=0;i<N1;i++) {
    for(int j=0;j<N2;j++) {
        double dist = @distance_metric@(point[i], point[j]);
        if(dist < mindist || dist >= maxdist) {
            continue;
        }
    }
    int ibin = @dist_to_bin_index@(dist);
    numpairs[ibin]++;
}
```
Simple Code is … simple

- Ignores domain knowledge
  (maxdist $<< L$)
- Not optimal for hardware*
- Can not be vectorized by compiler
Hardware Detour
Memory access is slow

Memory Caching

CPU
Regs
FREE
L1 I$
L1 D$
L2
(small)
200+ CYCLES
20+ CYCLES
3 CYCLES
MAIN RAM
(large)
SCEC, Dec 2018
Memory access is slow

How fast code runs depends on memory access patterns
Vector Instructions (SIMD)

Scalar mode
(one instruction produces one result)

SIMD processing
(one instruction can produce multiple results)
Untapped Potential Can Be Huge!

Vectorize & Thread or Performance Dies
Threaded + Vectorized can be much faster than either one alone

Many codes are still here
Hardware —-> Performance

- Power scales as $\text{freq}^3$
  - Multi-cores at lower clock (instead of one core with a 3GHz clock, 2 cores with 2.1 GHz provides 1.4x op/s @ 70% power)

- Memory access is slow
  - Layers of (smaller, faster, dedicated) -> (larger, slower, shared) caches

- Only one instruction per clock cycle
  - but, clock speeds have stalled
    - More calculations per clock tick (SIMD/vectorization)

Vectorized operations with efficient memory access within independent kernels
Back to Corrfunc
Simple Code is … simple

- Ignores domain knowledge
  
  (maxdist $< L$)

- Not optimal for hardware*

- Can not be vectorized by compiler

```c
for(int i=0;i<N1;i++) {
    for(int j=0;j<N2;j++) {
        double dist = @distance_metric@((point[i], point[j]));
        if(dist < mindist || dist $\geq$ maxdist) {
            continue;
        }

        int ibin = @dist_to_bin_index@((dist));
        numpairs[ibin]++;
    }
}
```
How Corrfunc works

- Grids the particle distribution into 3D cells of size ~Rmax
- Stores particles contiguously within each cell
- Sorts particles within a cell in z
- Only associates pairs of cells that may contain pairs
- Uses vectorised kernels on cell-pairs
- Outer OpenMP loop over cell-pairs
Why Corrfunc is FAST

- Grids extent with cells of Rmax (domain knowledge)
- Stores particles contiguously within each cell (memory access)
- Uses sorting to prune (algorithmic complexity)
- Uses vector intrinsics (vectorization)
- Uses OpenMP (multi-core)
Why Corrfunc is FAST: 3D Grid
Why Corrfunc is FAST: Sorting

Sorted in the z-direction

$R_{\text{max}}$
Performance of SIMD Kernels

\[ w_p(r_p) \]

\[ \xi(r) \]

\[ \text{Runtime [sec]} \]

\[ r_{\text{max}}/L_{\text{box}} \]

\[ \propto r_{\text{max}}^3 \]
Speedup from Vectorization (AVX)

Vectorization gains for code from the same developer
Speedup from Vectorization (AVX512)
Corrfunc Performance: Single-core

on github:
paper/scripts/generate_code_comparison.py

![Graph showing Corrfunc speed-up compared to other tools]

- Corrfunc faster
- Corrfunc slower

N_{\text{particles}}

- halotools
- kdcound
- Treecorr
- CUTE_box
- scikit-learn KDTree
- SciPy cKDTree
- mlpack RangeSearch
Corrfunc Performance: Multi-core

on github:
paper/scripts/generate_code_comparison.py
Why I wrote open-sourced Corrfunc

- Inherited codes took ~5 mins. MCMC would have exceeded the funding duration.
  - fast private version for my specific use-case
- Created custom code for experts with 6000x speedup (took < 24 hrs to create)
- Demonstrated the need for a fast, flexible, open-source package
- That initial 5 min calc. now takes ~5 secs with **Corrfunc**
Writing Portable and Fast Software is Difficult

- Python removes the portability issue
  - but not fast
- Compiled extensions use very basic compiler options (defaults options are the ones used for compiling python)
- Compile with the highest compiler-supported ISA
  - Check ISA at runtime

*Usability/Sustainability trumps everything*
Conclusions

- **Corrfunc** is optimised using domain knowledge, good memory access pattern, vectorization and OpenMP

- **Corrfunc** is “blazing fast” and
  - modular, user-friendly, documented, tested, OpenMP parallel, flexible API access, …
  - GPU version coming - thanks to ADACS
  - my highest cited bib-entry for last year ([ascl.net/1703.003](ascl.net/1703.003))
Title: Corrfunc: Blazing fast correlation functions on the CPU
Authors: Sinha, Manodeep; Garrison, Lehman
Publication: Astrophysics Source Code Library, record ascl:1703.003
Publication Date: 03/2017
Origin: ASCL
Keywords: Software
Bibliographic Code: 2017ascl.soft03003S
Conclusions

- **Corrfunc** is optimised using domain knowledge, good memory access pattern, vectorization and OpenMP
- **Corrfunc** is “blazing fast” and
  - modular, user-friendly, documented, tested, OpenMP parallel, flexible API access, …
  - GPU version coming - thanks to ADACS
  - my highest cited bib-entry for last year ([ascl.net/1703.003](ascl.net/1703.003))
for(int64_t i=0;i<N0;i++) {
    const AVX512_FLOATS m_xpos = AVX512_SET_FLOAT(*x0++);
    const AVX512_FLOATS m_ypos = AVX512_SET_FLOAT(*y0++);
    const AVX512_FLOATS m_zpos = AVX512_SET_FLOAT(*z0++);
    DOUBLE *localx1 = x1, *localy1 = y1, *localz1 = z1;
    for(int64_t j=0;j<N1;j++) {
        AVX512_MASK m_mask_left = (N1-j) >= AVX512_NVEC ? ~0:masks_per_misalignment_value_DOUBLE[N1-j];
        const AVX512_FLOATS m_x1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localx1);
        const AVX512_FLOATS m_y1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localy1);
        const AVX512_FLOATS m_z1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localz1);
        /* this might actually exceed the allocated range but we will never dereference that */
        localx1 += AVX512_NVEC;
        localy1 += AVX512_NVEC;
        localz1 += AVX512_NVEC;
        const AVX512_FLOATS m_xdiff = AVX512_SUBTRACT_FLOATS(m_x1, m_xpos);
        const AVX512_FLOATS m_ydiff = AVX512_SUBTRACT_FLOATS(m_y1, m_ypos);
        const AVX512_FLOATS m_zdiff = AVX512_SUBTRACT_FLOATS(m_z1, m_zpos);
        const AVX512_FLOATS m_sqr_xdiff = AVX512_SQUARE_FLOAT(m_xdiff);
        const AVX512_FLOATS x2py2 = AVX512_FMA_ADD_FLOATS(m_ydiff, m_ydiff, m_sqr_xdiff);
        const AVX512_FLOATS r2 = AVX512_FMA_ADD_FLOATS(m_zdiff, m_zdiff, x2py2);
        const AVX512_FLOATS m_rpmx_mask = AVX512_MASK_COMPARE_FLOATS(m_mask_pimax, r2, m_sqr_rpmx, _CMP_LT_OQ);
        /* Create a combined mask */
        /* This gives us the mask for all sqr_rpm < r2 < sqr_rpmax */
        m_mask_left = AVX512_MASKCOMPARE_FLOATS(m_mask_pimax, r2, m_sqr_rpmx, _CMP_LT_OQ);
        if(m_mask_left == 0) {
            continue;
        }
        /* Loop backwards through nbins. m_mask_left contains all the points that */
        /* are less than rmmax at the beginning of the loop. */
        for(int kbin=nbin-1;kbin>=1;kbin--) {
            const AVX512_FLOATS m_bin_mask = AVX512_MASKCOMPARE_FLOATS(m_mask_left, r2, m_rupp_sq[kbin-1],_CMP_GE_OQ);
            npairs[kbin] += bits_set_in_avx512_mask_DOUBLE[m_bin_mask];
            /* ANDNOT(X, Y) -> NOT X AND Y */
            m_mask_left = AVX512_MASKBITWISE_AND_NOT(m_bin_mask, m_mask_left);
            if(m_mask_left == 0) {
                break;
            }
        }
    }
}
}
Corrfunc Kernel

https://gist.github.com/manodeep/ffdc60024fd6df8b5264657f0be2f967
Speedup from Vectorization (SSE4.2)
for(int64_t i=0;i<N0;i++) {
    const AVX512_FLOATS m_xpos = AVX512_SET_FLOAT(*x0++);
    const AVX512_FLOATS m_ypos = AVX512_SET_FLOAT(*y0++);
    const AVX512_FLOATS m_zpos = AVX512_SET_FLOAT(*z0++);
    DOUBLE *localx1 = x1, *localy1 = y1, *localz1 = z1;
    for(int64_t j=0;j<N1;j++) {
        AVX512_MASK m_mask_left = (N1-j) >= AVX512_NVEC ? ~0:masks_per_misalignment_value_DOUBLE[N1-j];
        const AVX512_FLOATS m_x1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localx1);
        const AVX512_FLOATS m_y1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localy1);
        const AVX512_FLOATS m_z1 = AVX512_MASKZ_LOAD_FLOATS_UNALIGNED(m_mask_left, localz1);
        /* this might actually exceed the allocated range but we will never dereference that */
        localx1 += AVX512_NVEC;
        localy1 += AVX512_NVEC;
        localz1 += AVX512_NVEC;
        const AVX512_FLOATS m_xdiff = AVX512_SUBTRACT_FLOATS(m_x1, m_xpos);
                    /* (x[j:j+NVEC-1] - x0) */
        const AVX512_FLOATS m_ydiff = AVX512_SUBTRACT_FLOATS(m_y1, m_ypos);
                    /* (y[j:j+NVEC-1] - y0) */
        const AVX512_FLOATS m_zdiff = AVX512_SUBTRACT_FLOATS(m_z1, m_zpos);
                    /* z2[j:j+NVEC-1] - z1 */
        const AVX512_FLOATS m_sqr_xdiff = AVX512_SQUARE_FLOAT(m_xdiff);
                    /* (x0 - x[j])^2 */
        const AVX512_FLOATS x2py2  = AVX512_FMA_ADD_FLOATS(m_ydiff, m_ydiff, m_sqr_xdiff);
                    /* dy*dy + dx^2*/
        const AVX512_FLOATS r2 = AVX512_FMA_ADD_FLOATS(m_zdiff, m_zdiff, x2py2);
                    /* dz*dz + (dy^2 + dx^2)*/
        const AVX512_MASK m_rpmax_mask = AVX512_MASK_COMPARE_FLOATS(m_mask_pimax, r2, m_sqr_rpmax, _CMP_LT_OQ);
                    /* Create a combined mask */
        /* This gives us the mask for all sqr_rpmin <= r2 < sqr_rpmax */
        m_mask_left = AVX512_MASK_COMPARE_FLOATS(m_mask_left, r2, m_rupp_sqr[kbin-1],_CMP_GE_OQ);
        if(m_mask_left == 0) {
            continue;
        }
        /* Loop backwards through nbins. m_mask_left contains all the points that */
        /* are less than rpmax at the beginning of the loop. */
        for(int kbin=nbin-1;kbin>=1;kbin--) {
            const AVX512_FLOATS m_bin_mask = AVX512_MASK_COMPARE_FLOATS(m_mask_left, r2, m_rupp_sqr[kbin-1],_CMP_GE_OQ);
            npairs[kbin] += bits_set_in_avx512_mask_DOUBLE[m_bin_mask];
            /* ANDNOT(X, Y) -> NOT X AND Y */
            m_mask_left = AVX512_MASK_BITWISE_AND_NOT(m_bin_mask, m_mask_left);
            if(m_mask_left == 0) {
                break;
            }
        }
    }
}
}
github.com/manodeep/Corrfunc/

Framework: PVD calculation reduced from 600+ hours to ~1 min
Framework: PVD calculation reduced from 600+ hours to ~1 min
Framework: PVD calculation reduced from 600+ hours to ~1 min
Framework: PVD calculation reduced from 600+ hours to ~1 min
Framework: PVD calculation reduced from 600+ hours to ~1 min
Framework: PVD calculation reduced from 600+ hours to ~1 min
Memory access is slow

- Speed of light limitations (30 cm/ns)
  - For a 3 GHz clock, light only travels 10 cm
- Many hardware layers between requesting memory and getting data
- CPUs need to perform many calculations simultaneously
Bottlenecks in computing

How fast code runs depends on memory access patterns

How fast code runs depends on memory access patterns
How a CPU keeps busy (also why we got “Meltdown”)

Figure 1: Example of 4-stage pipeline. The colored boxes represent instructions independent of each other.
CPU Performance Bottlenecks

The graph illustrates the performance gap between processor and memory over the years from 1980 to 2010. The performance of the processor has been increasing at a faster rate than that of memory, leading to a widening gap.

SCEC, Dec 2018
Memory access is slow

https://blog.codinghorror.com/the-infinite-space-between-words/
How fast code runs depends on memory access patterns. Memory access is slow.
Not all operations are equal

- Modern cpus are extremely complex and try to predict data access patterns (hence, MELTDOWN, SPECTRE hardware bugs)
  - avoid if conditions (use `?` for the ternary operator)
- Divisions are 5 times more time-consuming than multiplication
  - Beware of trigonometric functions (use trig. identities, if possible)
- Profile your code. I am wrong > 50% of the time
Users vs CPU vendor

- **User**: Fastest time to solution is better
- **Vendor**: Lowest power consumption for a fixed problem size while maintaining/improving time to solution
- These two metrics are not the same
User wishlist vs CPU Features

- **User**: Faster CPU clocks (write old-style code)

- Physics is a buzzkill

- **Vendor**: Slower and many more individual cores per cpu (think GPUs) with wider vector widths (more calculations per clock tick)