How to gain single-thread performance: Instruction pipelines, CPU cache optimisation, and SIMD

Hans Dembinski, TU Dortmund
CORSIKA 8 Workshop, July 2022, Heidelberg

Take-home message

- Single-instruction-multiple-data (SIMD) give great single-thread performance
  - Up to 8x on current processors
  - Calculations in float precision 2x as fast as double precision
  - Orthogonal to parallelization via multi-threading
- Structs of arrays often faster than arrays of structs
- Let compiler write SIMD code for you
- CPU optimization is complex and unintuitive: Measure performance, do not guess!

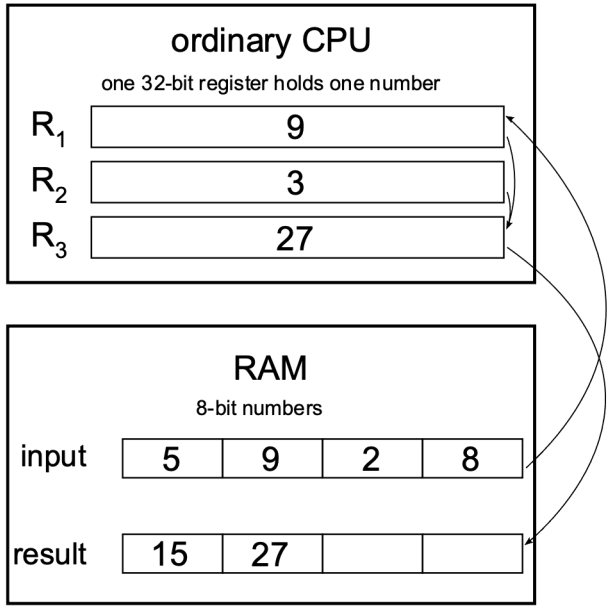
Python, Numpy, and Numba

- Code examples here are Python, compiled with Numba
  - Numba makes optimal use of available SIMD instructions for your CPU
  - Numpy may not use all SIMD instructions unless self-compiled, interpreter still runs in between array operations

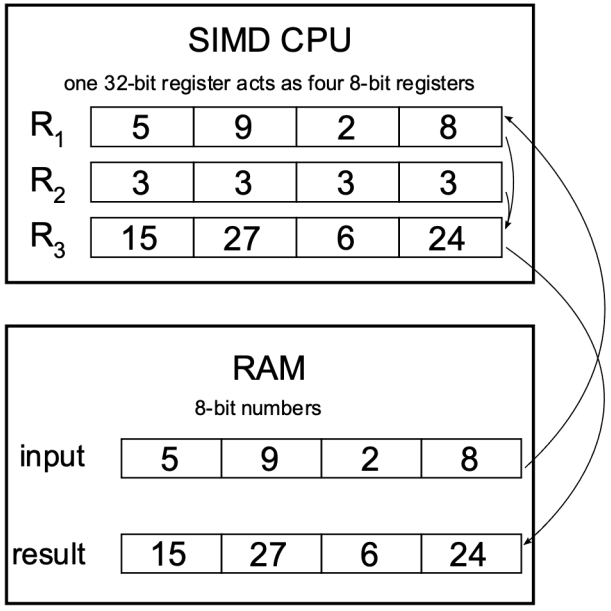
- Results apply to C++ as well
  - Numba compiles numerical code with LLVM using LLVM optimizer that is also used by clang
  - Speed of Numba-compiled functions is on par with optimized C++

SIMD: Single-instruction-multiple-data

- Instructions sets: MMX 1997, 3DNow! 1998, SSE (2, 3, 4) 1999-2008, AVX (2, 512) 2011-2016
- Compiler support for AVX-512: gcc 4.9+, clang-3.9+, icc-15.0.1+
- Vectorized operations: **add**, **sub**, **mul**, **div**, **min**, **max**, **mov**, **sqrt**, ...



Operation count:
4 loads, 4 multiplies, and 4 saves



Operation count:
1 load, 1 multiply, and 1 save

Decora on Wikipedia, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=30547507

Decora on Wikipedia, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=30547549

Array of struct vs. struct of arrays

Array of structs (AOS) memory layout: $x_1, y_1, z_1, x_2, y_2, z_2, \ldots$

Struct of arrays (SOA) memory layout: $x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots$

```
class Particle:
    x: float
    y: float
    z: float
```

```
class Particles:
    x: np.array
    y: np.array
    z: np.array
```

```
aos = np.empty((n, 6))
soa = np.empty((6, n))
```

- Array of structs usually the intuitive design choice, but...
- SOA often more efficient (but measure to make sure)
- SIMD instructions most efficient if arguments adjacent in memory

Motivating example

- Particle: 3 position coordinates, 3 momentum coordinates
- Task: Move location of a particle along its current momentum by *step*

- Naive implementation
  - Loop over particles
  - Compute displacement vector
  - Add to location
- Not SIMD efficient: only 3 **mul** and 3 **add** can be vectorized per iteration

```
def move_aos(aos, step):
    for i in range(len(aos)):
        r = aos[i, :3]
        p = aos[i, 3:]
        pn = np.sqrt(p[0] * p[0] + p[1] * p[1] + p[2] * p[2])
        ps = step / pn
        r += p * ps
    return aos
```

Motivating example

- SIMD-friendly improved version
  - Do all **mul**s first (creating temporary arrays)
  - Then do all **add**s
  - Then do all **sqrt**s
  - More efficient without modifying data structures
- Still inefficient, because px1, px2, ... not adjacent in memory (cache misses)

```
def move_aos_improved(aos, step):
    r = aos[:, :3]
    p = aos[:, 3:]
    pn = np.sqrt(p[:,0] * p[:,0] + p[:,1] * p[:,1] + p[:,2] * p[:,2])
    ps = step / pn
    for i in range(3):
        r[:, i] += p[:, i] * ps
    return aos
```

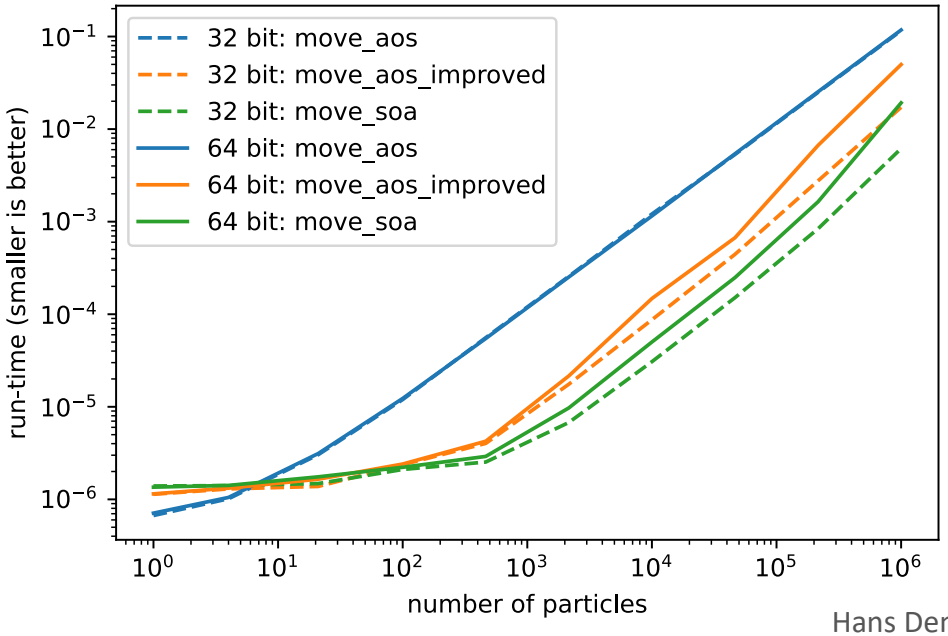
Motivating example

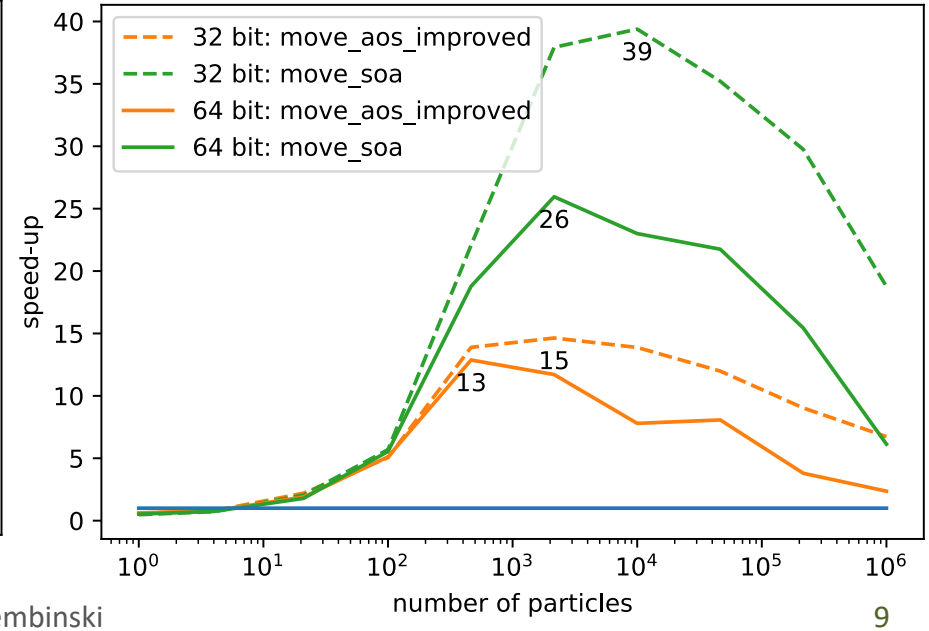
- Most SIMD-friendly version with SOA instead of AOS
  - All operations vectorized: **mul**, **add**, **div**, **sqrt**
  - All operation arguments adjacent in memory, except for **add**
- Creates several temporary arrays: slow/bad, but fixed cost
- Potential for further optimization: process particles in fixed-size chunks to use temporary arrays of fixed that fit in L1 cache

```
def move_soa(soa, step):
    r = soa[:3]
    p = soa[3:]
    pn = np.sqrt(p[0] * p[0] + p[1] * p[1] + p[2] * p[2])
    ps = step / pn
    for i in range(3):
        r[i] += p[i] * ps
    return soa
```

Benchmark

- 2.8 GHz Quad-Core Intel Core i7, Numba 0.55.1
- **move_soa** up to **39x** faster than naive implementation
  - Peak efficiency for chunks of 400 to 10000 particles processed at once; depends on size of L1 cache and single vs. double precision
  - Peak efficiency can be reached for larger arrays by processing particles in chunks
- **move_aos_improved** and **move_soa** profit from calculation in single precision



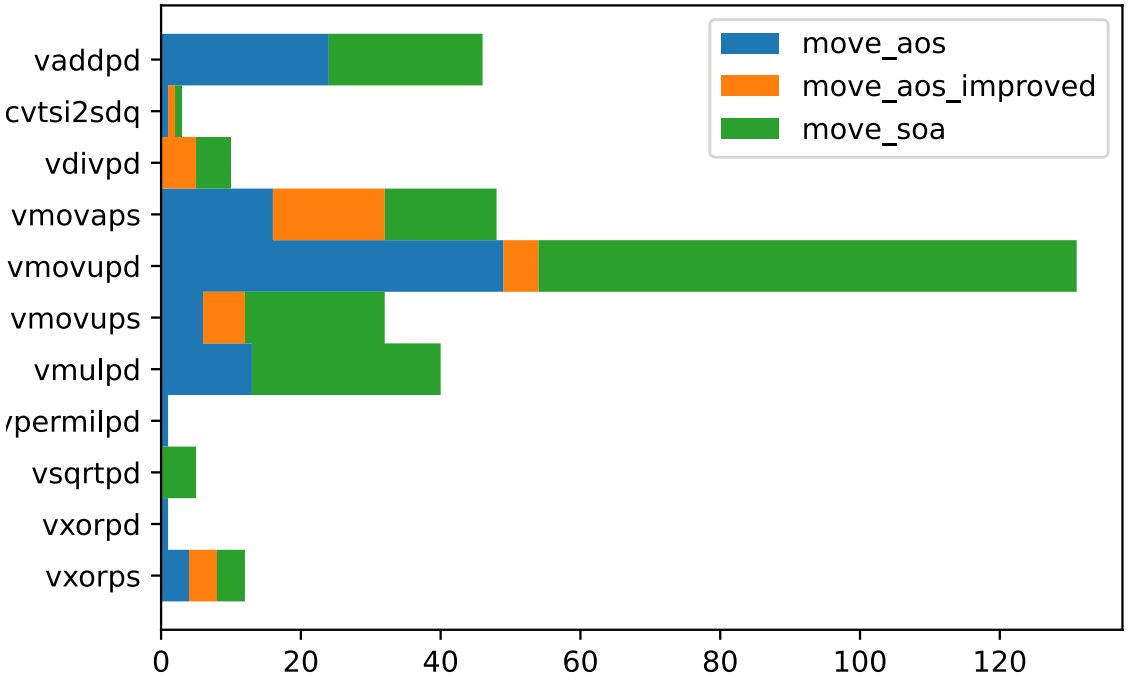


Usage of SIMD instructions

- Numba allows one to inspect compiled assembler
- Compare number of SIMD instructions

cvtsi2sdq: convert int to double
vpermilpd: permute pairs of doubles

move_aos also uses many SIMD instructions, nevertheless very inefficient



move_aos cannot use vectorized **div**

move_soa uses more aligned **mov**

only move_soa profits from vectorized **sqrt**

No fused multiply-add instructions, because associative math not enabled (but does not improve this code)

Summary

- Things to consider to get fast code
  - **Measure! Measure! Measure!**
  - Optimize use of SIMD instructions, CPU cache, CPU instruction pipelines
  - Avoid frequent memory allocations (infrequent are ok, small fixed buffers are ok)
  - Calculate in single precision if feasible (may require numerically stable algorithms)
  - Let compile write SIMD instructions for you
  - div slow compared to mul; replace div with mul if possible (or see next point)
  - Enable associative math (reassoc, contract, arcp); should be safe, but check results
- Rules-of-thumb (but don't trust them, measure)
  - "Local parallelism": Organize code so same Op is applied to adjacent values in memory
  - Calculating with arrays is good (Numpy style)
  - SOA often outperforms naive AOS

Jim Pivarsky: "If you don't use multi-threading, another process can use the extra threads. If you don't use SIMD instructions, no one else can use them."