Optimizing Applications

Dr. Axel Kohlmeyer

Assistant Dean for High-Performance Computing
Associate Director, ICMS
Associate Director, TMI
College of Science and Technology
Temple University, Philadelphia
axel.kohlmeyer@temple.edu

External Scientific Associate
International Centre for Theoretical Physics, Trieste, Italy
akohlmey@ictp.it

Post-Install Optimization or: How to Make an Application Faster Without Changing It?

```
 PerfTop:    8016 irqs/sec  kernel: 9.9%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
------------------------------------------------------------------------------------------------

             samples  pcnt function                  DSO
             _______ _____ ______________________ ______________________________________

            53462.00 52.2% __ieee754_log          /lib64/libm-2.12.so
            10490.00 10.3% R_binary               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             8704.00  8.5% clear_page_c           [kernel.kallsyms]
             5737.00  5.6% __ieee754_exp          /lib64/libm-2.12.so
             4645.00  4.5% math1                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             3070.00  3.0% __log                  /lib64/libm-2.12.so
             3020.00  3.0% __isnan                /lib64/libc-2.12.so
             2094.00  2.0% R_gc_internal          /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1643.00  1.6% do_summary             /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1251.00  1.2% __isnan@plt            /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1210.00  1.2% real_relop             /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1161.00  1.1% __GI___exp             /lib64/libm-2.12.so
              754.00  0.7% __isnan                /lib64/libm-2.12.so
              739.00  0.7% R_log                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              553.00  0.5% __kernel_standard      /lib64/libm-2.12.so
              550.00  0.5% do_abs                 /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              462.00  0.5% __mul                  /lib64/libm-2.12.so
              439.00  0.4% coerceToReal           /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              413.00  0.4% finite                 /lib64/libm-2.12.so
              358.00  0.3% log@plt                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              182.00  0.2% get_page_from_freelist [kernel.kallsyms]
              120.00  0.1% __alloc_pages_nodemask [kernel.kallsyms]
```

Optimization Step 1: Alternatives

- libm is part of standard C, thus it is ubiquitous, but not many alternatives for x86/x86_64 exist
- Focus is typically put on standard compliance (glibc) or extended accuracy (cephes)
- AMD offers libM (originally bundled with ACML), it is binary only and for x86_64 only

  => program a shared object providing a log() function which calls amd_log() and links to libM
  => override log() in libm via $LD_PRELOAD

... and here is the result

```
 PerfTop:    8020 irqs/sec  kernel:17.2%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
------------------------------------------------------------------------------------------------

             samples  pcnt function                  DSO
             _______ _____ _________________________ _____________________________________

            24702.00 19.5% __amd_bas64_log           /opt/libs/fastermath-0.1/libamdlibm.so
            22270.00 17.6% R_binary                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            18463.00 14.6% clear_page_c              [kernel.kallsyms]
            10480.00  8.3% __ieee754_exp             /lib64/libm-2.12.so
             9834.00  7.8% math1                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             9155.00  7.2% log                       /opt/libs/fastermath-0.1/fasterlog.so
             6269.00  5.0% __isnan                   /lib64/libc-2.12.so
             4214.00  3.3% R_gc_internal             /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             3074.00  2.4% do_summary                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2285.00  1.8% real_relop                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2257.00  1.8% __isnan@plt               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2076.00  1.6% __GI___exp                /lib64/libm-2.12.so
             1346.00  1.1% R_log                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1213.00  1.0% do_abs                    /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1075.00  0.8% __kernel_standard         /lib64/libm-2.12.so
              894.00  0.7% coerceToReal              /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              780.00  0.6% __mul                     /lib64/libm-2.12.so
              756.00  0.6% finite                    /lib64/libm-2.12.so
              729.00  0.6% amd_log@plt               /opt/libs/fastermath-0.1/fasterlog.so
              706.00  0.6% amd_log                   /opt/libs/fastermath-0.1/libamdlibm.so
              674.00  0.5% log@plt                   /opt/binf/R-2.13.0/lib64/R/bin/exec/R
```

Step 2: Can We Do Better?

- x86 FPU internal log() is <u>slower</u> than libm
- The log() in LibM is about 2.5x faster than libm
- Total execution time is reduced by ~30%
- Note: this is a <u>very</u> application specific speedup
- Other commonly used “expensive” libm functions are exp() and pow() (= log() + exp()); => fast pow(x,n) with integer n via multiplication
- exp() version in tested AMD's LibM was broken => try to optimize log()/exp() from cephes lib

How To Compute log() or exp()?

- Evaluating log(x) or exp(x) according to its definitions is too time consuming; floating point math requires only an approximation anyway

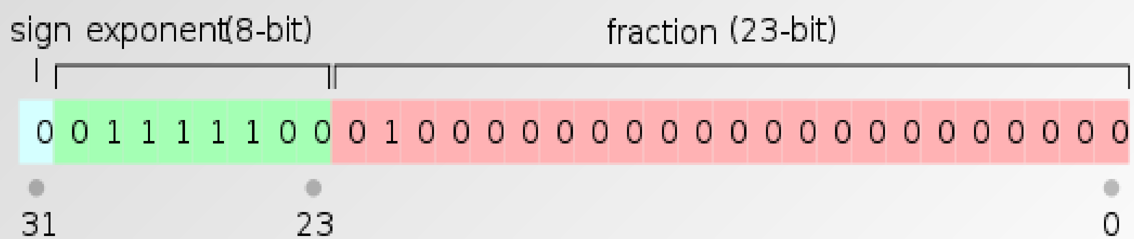
  => Four step process in cephes:

  1. Handle special cases, over-/underflow (-> skip it)
  2. Perform a “range reduction” (-> use IEEE754 tricks)
  3. Approximate log(x)/exp(x) in reduced x interval from polynomial or rational function or spline table
  4. Combine results of steps 2 & 3

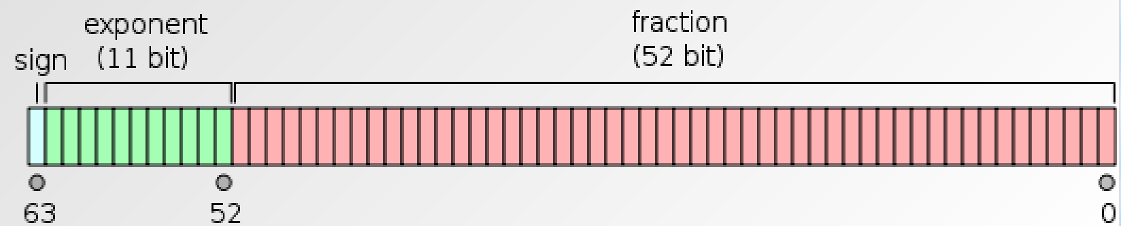
- Optimizer friendly C code with compiler “hints”

IEEE 754 Floating-point Numbers

- The IEEE 754 standard defines: storage format, result of operations, special values (infinity, overflow, invalid number), error handling
=> portability of compute kernels ensured
- Numbers are defined as bit patterns with a sign bit, an exponential field, and a fraction field
  - Single precision:
8-bit exponent
23-bit fraction



  - Double precision:
11-bit exponent
52-bit fraction



Fast Implementation of exp()

- Range reduction: $x = f + n; \;\; n \in \mathbb{Z}, -0.5 \leq f < 0.5$
  $$2^x = 2^{f+n} = 2^f \cdot 2^n$$
- Get 2^n from setting IEEE-754 exponent: zero mantissa bits (=1), exponent is n + 1023
- Padé Approximation: $2^f = 1.0 + \left(\frac{2f \cdot P_3(f^2)}{P_3(f^2) + Q_3(f^2)} \right)$
- Unroll & interleave $P_3(f^2)$ and $Q_3(f^2)$ evaluation
- Store coefficients for P/Q at aligned address
- $\exp(x) = \exp2(\log_2(e) * x)$

Fast Implementation of log(x)

- Range reduction: $x = f \cdot 2^n; \quad n \in \mathbb{Z}, 1.0 \leq f < 2.0$

$$\log_2(x) = \log_2(f \cdot 2^n) = \log_2(f) + \log_2(2^n) = \log_2(f) + n$$

- Get *n* from reading IEEE-754 exponent – 1023
  set exponent to 1023 (i.e. 0) and read/store *f*
- Truncate integer representation of *f* via bitshift to get spline table lookup index (12 bits)
- Approximation: evaluate cubic spline for log(*f*)
- log2(x)= n+$\log_2(e)$*log(f); log(x)= log(2)*n+log(f)
- Store pre-computed spline table at aligned address

The “Faster” Math Library

- exp() 1.5-3x, log() 2-4x times faster than libm
- Faster when compiled for SSE4 or AVX
- More speedup in 64-bit mode (more registers)
- No branches, gcc attributes for data access
- no vectorization (but uses SSE/AVX unit)
- **Wrong** results for out-of-range arguments
- Most useful for post installation optimization
- URL: http://github.com/akohlmey/fastermath

Optimizing a Force Computation Kernel in LAMMPS

- LAMMPS is a highly parallel molecular dynamics code tuned for large systems
- MPI parallelization with domain decomposition
- Models implemented as derived C++ classes Pair, Bond, Angle, Dihedral, KSpace, Fix, …
- Simplest application example is homogeneous liquid of Lennard-Jones particles (e.g. Argon)
- Force computation is in PairLJCut::compute()
- Create derived classes PairLJCutDemo#

Optimization 1: Change 2d-Array into 1d-Array

- In LAMMPS properties like position or force are stored in 2d-arrays: **`double **atom->x`**
- X coordinate of atom 0: **`atom->x[0][0]`**
- Less ‘pointer chasing’: **double **x = atom->x;**
- Underlying storage is ‘flat’ (like in Fortran): **`x[0][0],x[0][1],x[0][2],x[1][0],...`**
- **typedef struct {double x,y,z;}dbl3_t;**
- **dbl3_t *x = (dbl3_t *) atom->x[0];**
- **`X[0].x,x[0].y,x[0].z,x[1].x,...`**

Optimization 2: Cache Outer Loop Forces

- Force is computed for pairs of atoms i and j
- Outer loop over atoms i, inner loop over j
- Result is added to atom i and subtracted from j
- Compiler cannot know that it **never** is: i == j thus it must not cache results for atom i in and store only at the next outer loop iteration
- Thus we introduce **double ftmp[3]** to collect force results for atom i; set to 0 at beginning and add to atom->f at end of outer loop iteration

Optimization 3: Code Specialization with Templates

- Force kernel has if statements in inner loop: a) bad idea, b) most of the conditions constant
- Need multiple variants of the force kernel for different set of constants:
- Replace ::compute() with ::eval<int,int,int>()
- In ::compute() have decision tree that calls eval<0,0,1>() or eval<1,0,0>() and so on
- When compiling, the compiler will create multiple instances of ::eval<>() and optimize away the if statements where possible

Kernel Optimizations Benefits

- Test system: 32000 atoms for 1000 steps
- Optimization 1: 2d-array to 1d-array
  84.5s instead of 104s => 23%
- Optimization 2: cache outer loop results
  98.5s instead of 104s => 5.5%
- Optimization 3: templated compute kernel
  101.5s instead of 104s => 2.5%
- Total improvement:
  73.6s instead of 104s => 41.3%

Quick 'n' Dirty Optimization or: How Much Can You Optimize a Code Over the Weekend?

- From the “HPC Helpdesk”: hpc@temple.edu
  User requests access to HPC resource because his self-written program needs too much memory and runs too slow on desktop
- Next, the user asks for parallel programming assistance to handle large matrices
- Application is one file with ~1000 lines C code => could be perfect showcase for a “minimum effort” optimization and parallelization study

  => “The game is afoot...”

Structure of the Application

- Input data: a network, a list of nodes (names) and a list of connections between those nodes (e.g. “friends” in a social network)
- Objective: find a subset where the ratio of internal vs. external connections is maximal
  1) Clustering: pick a sample of connected nodes around a random seed, pick the most connected nodes as new seed, repeat until converged
  2) Pruning: Take connection matrix from 1), remove most unfavorable entry, record target function value and subset, repeat until matrix is of rank 1

Optimization 1: Reduce Memory

- The by far most time consuming step is the calculation of the “connection matrix” of the selected nodes
- The matrix elements are either 1 (if two nodes are connected) or 0 (if the are not connected)
- Storage element was **`unsigned long int`**

=> use **`char`** instead

=> 4x (32-bit) to 8x (64-bit) memory savings

=> 1.5-2x performance increase

Optimization 2: Compiler

- The reference executable was compiled with gcc using default settings, i.e. <u>no</u> optimization
- Using compiler optimizations leads to significant performance increase
- Compiler optimization can be improved through using **`const`** qualifiers in the code wherever possible and local code changes
- Hide complex data types with **`typedef`**

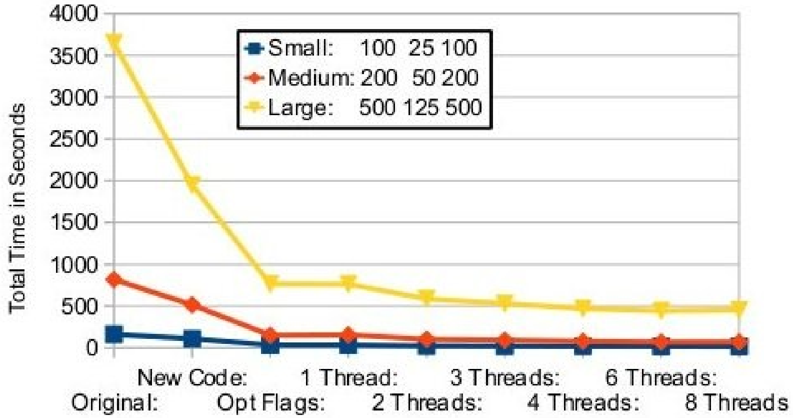
=> 2.5 – 3.5x speedup

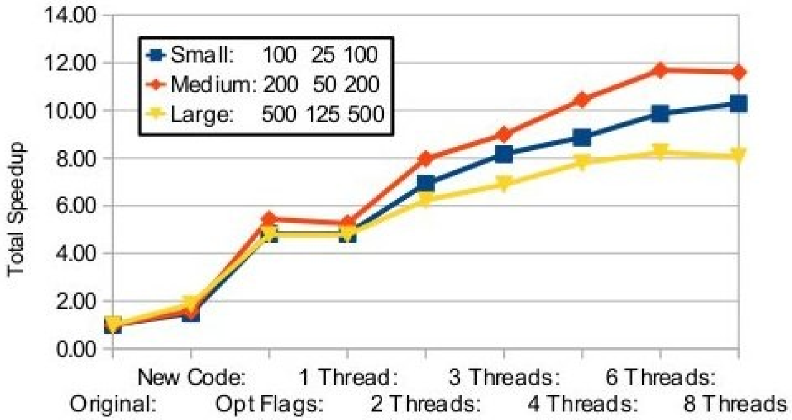
Optimization 3: Parallelization

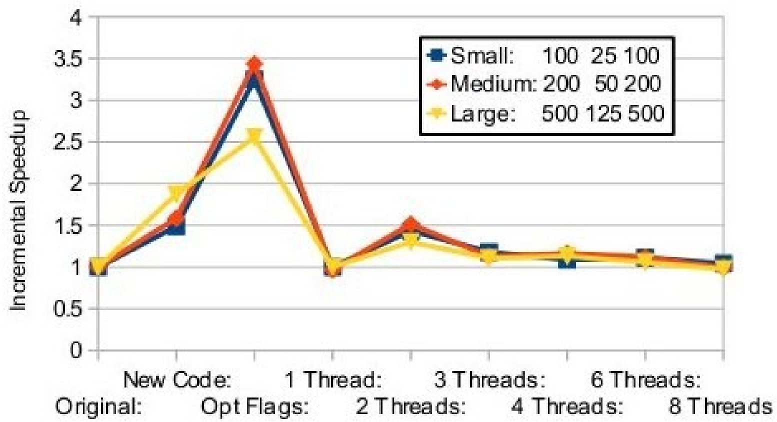
- The construction of the connection matrix has no data dependencies => multi-threading
- Using OpenMP requires only adding one directive and a little bit of code reorganization
- Speedup going from serial to 2 threads: 1.5x
- Speedup levels out at 6-8 threads: 2.5x total

  => very little computation, mostly data access
  => performance limited by memory contention
- Total improvement: 8x-12x with 8 threads

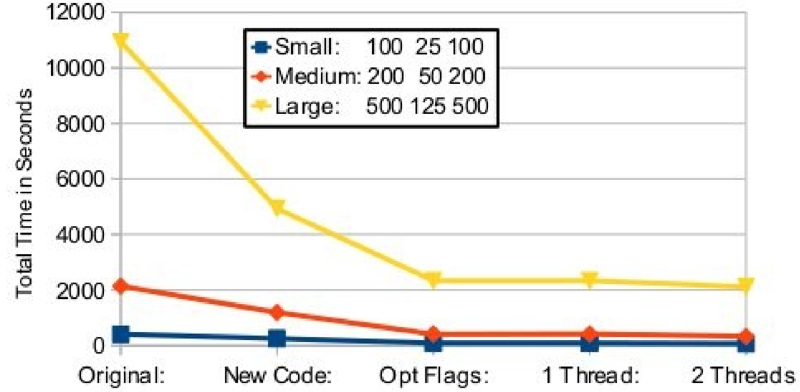
2x Intel Xeon X5677, 3.5GHz
96 GB 1333MHz DDR3 RAM

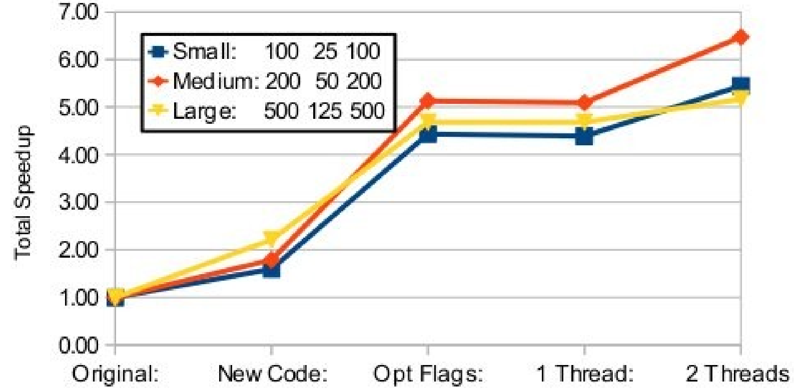


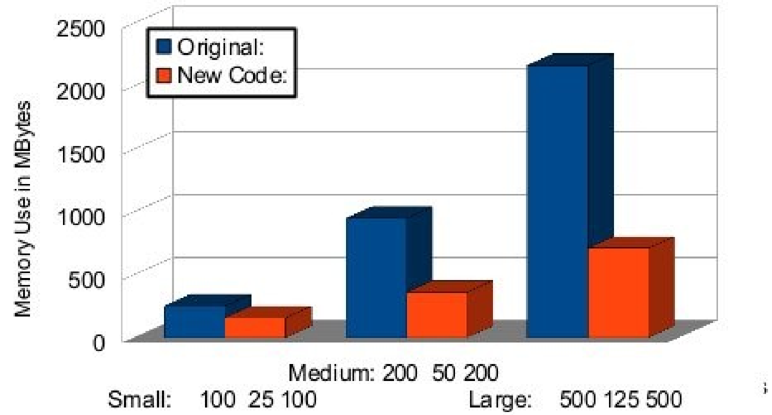




1x Intel Core2 Duo, 1.4GHz
4 GB 800MHz DDR2 RAM







Proper Optimization or: The Power of the Rewrite

- Quick'n'dirty optimizations of T-CLAP resulted in significant improvements in a short time
- More optimization potential with rewrite:
  - Connection matrix information requires only 1 bit => reduce storage by another factor of 8 (vs. **`char`**)
  - Network represented by structs and lists of pointers
    => pointers require more storage in 64-bit mode
    => many pointers point to the same data
    => C aliasing rules require re-reading data
  - Pruning implementation uses memmove() to compact matrix rows => bottleneck for large data

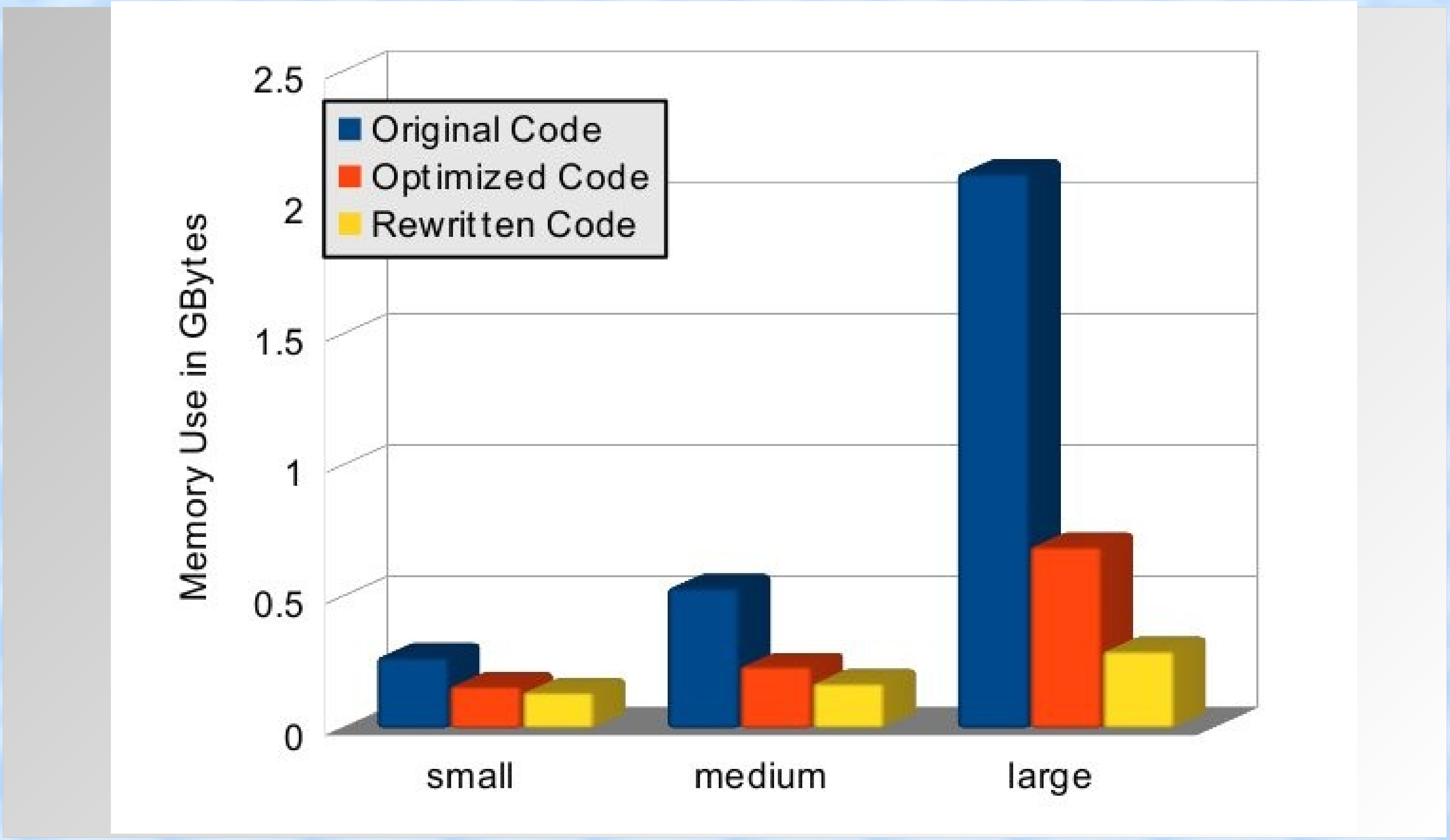
Proper Optimization or: The Power of the Rewrite

- Quick'n'dirty optimizations of T-CLAP resulted in significant improvements in a short time
- More optimization potential with rewrite:
  - Connection matrix information requires only 1 bit
    => reduce storage by another factor of 8 (vs. **`char`**)
  - Network represented by structs and lists of pointers
    => pointers require more storage in 64-bit mode
    => many pointers point to the same data
    => C aliasing rules still require re-reading data
  - Pruning implementation uses memmove() to compact matrix rows => bottleneck for large data

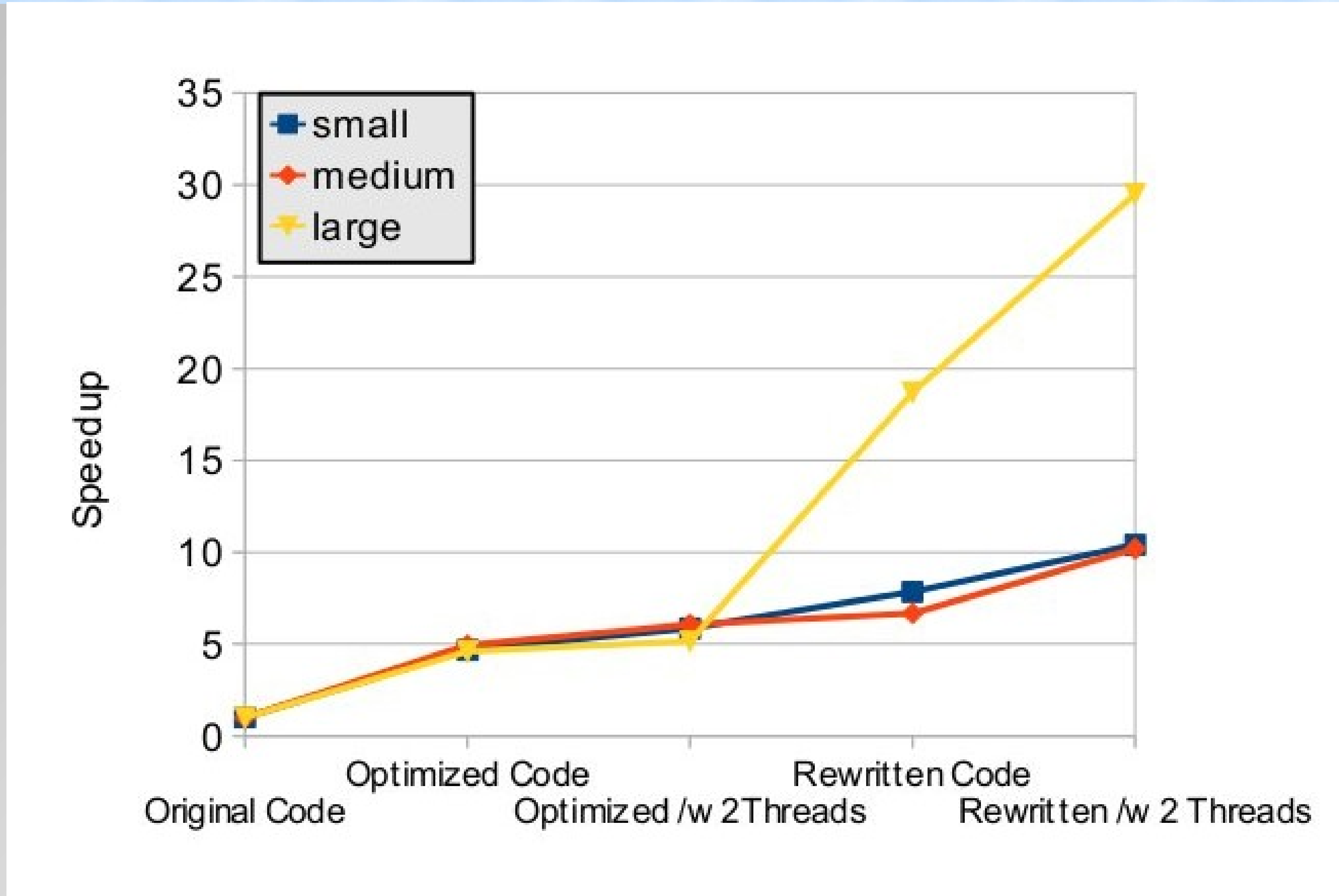
The Rewrite

- Rewrite in C++ (more optimization hints than C)
- Use STL container classes
- std::vector<bool> uses single bit per entry
- Single list of structs for all network nodes, all references via index lists (std::vector<int>) => no more need to re-read data
- Leave data in place during pruning, maintain ‘skip lists’ of valid rows and columns instead
- Rewrite piece-by-piece to reproduce original

Memory Usage After Rewrite



Performance After Rewrite



Parallel Performance After Rewrite