Observations on Optimizing Scientific Computing Applications

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Computational Science
Assistant Vice President for High-Performance Computing

Temple University
Philadelphia PA, USA

a.kohlmeyer@temple.edu

TEMPLE UNIVERSITY®

Overview

- 1) Introduction:
  What makes computations faster?
- 2) Example 1: Post-install optimization
  Optimizing an application without changing it
- 3) Example 2a: Quick-n-dirty optimization
  How much speedup can you get in a weekend
- 4) Example 2b: Proper application optimization
  The power of the rewrite
- 5) Conclusions

How do (many) Computational Scientists view a CPU?

Calculations run faster in case we:

- Type faster, read faster (Faster RAM)
- Turn crank faster, use motor (Higher Clock)
- Use better technology (New Hardware)

What is really going on?

TEMPLE UNIVERSITY®

What is really going on? (2)

- CPU Cache memory (per core, per socket),
  => speeds up access to recently used data
  => can reduce memory bandwidth contention
  => caches can be per core or shared
  => multiple levels of caches with different size,
  speed and “closeness” to the CPU core
- Pipelined superscalar CPU (implicit parallelism),
  => one core can work on multiple instructions
  => speculative and out-of-order execution
- Vector instructions: process “wide” registers containing multiple data elements

TEMPLE UNIVERSITY®

Running Faster: Pipelining

- Multiple steps in one CPU "operation": fetch, decode, execute, memory, write back => multiple functional units in CPU design
- Using a pipeline allows for a faster clock
- Dependencies or branches can stall pipeline, only "fast" instructions pipelined: => branch prediction => no "if" in inner loop

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

TEMPLE UNIVERSITY®

How Would This Statement Be Executed?

`z = a * b + c * d;`

Actual steps:

z1 = a * b;

Data load can start while multiplying

z2 = c * d;

Start data load for next command

z= z1 + z2;

1. Load **a** into register **R0**
2. Load **b** into **R1**
3. Multiply **R2 = R0 * R1**
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply **R5 = R3 * R4**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

Pipeline savings:
1 step out of 8, plus 3 more if next operation independent

TEMPLE UNIVERSITY®

Running Faster: Superscalar

- Superscalar CPU => instruction level parallelism
- Redundant functional units in single CPU
  => multiple instructions executed at same time
- Often combined with pipelined CPU design
- No data dependencies,
  no branches
- **Not** SIMD/SSE/MMX
- Optimization:
  => loop unrolling

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Superscalar & Pipelined CPU Execution

`z = a * b + c * d;`

Actual steps:

`z1 = a * b;`

1. Load **a** into register **R0** **and** load **b** into **R1**
2. Multiply **R2 = R0 * R1** **and** load **c** into **R3** **and** load **d** into **R4**

`z2 = c * d;`

Start data load for next command

3. Multiply **R5 = R3 * R4**

`z= z1 + z2;`

4. Add **R6 = R2 + R5**
5. Store **R6** into **z**

Superscalar pipeline savings:
3 out of 8 steps, plus 3 if next operation independent

TEMPLE UNIVERSITY®

Vectorized Loop

```
for (i = 0; i < length; i++) {
  z[i] = a[i] * b[i] + c[i] * d[i];
}
```

Vector registers on a CPU can hold multiple numbers and load, store or process them in parallel (**SIMD**):

```
for (i = 0; i < length; i +=2) {
 z[i] = a[i]  *b[i]   + c[i]  *d[i];
 z[i+1]=a[i+1]*b[i+1] + c[i+1]*d[i+1];
}
```

Executed together

This is **<u>in addition</u>** to superscalar pipelining and with using special vector instructions (SSE,AVX,etc.)

TEMPLE UNIVERSITY®

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

- Importing well known compute kernels from libraries is quite common in HPC
  Examples: BLAS/LAPACK, FFT(W)
- For BLAS multiple compatible implementations exist: MKL, ACML, Goto-BLAS, ATLAS, ESSL
- Usually link time choice; with shared libs alternative compilations of same library can be provided via $LD_LIBRARY_PATH; some libs offer a “dynamic dispatch”, i.e. a selection between alternatives at run time (e.g. MKL)

TEMPLE UNIVERSITY®

There are less obvious libraries with optimization potential: e.g. libm

```
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]
```

TEMPLE UNIVERSITY®

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
```

TEMPLE UNIVERSITY®

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

TEMPLE UNIVERSITY®

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”

TEMPLE UNIVERSITY®

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
- $\exp(x) = \exp2(\log_2(e) * x)$

TEMPLE UNIVERSITY®

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)
- **<u>Wrong</u>** results for out-of-range arguments
- Most useful for post installation optimization
- URL: http://github.com/akohlmey/fastermath

TEMPLE UNIVERSITY®

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

- From the “HPC Hepldesk”: 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...”

TEMPLE UNIVERSITY®

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

TEMPLE UNIVERSITY®

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

TEMPLE UNIVERSITY®

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

TEMPLE UNIVERSITY®

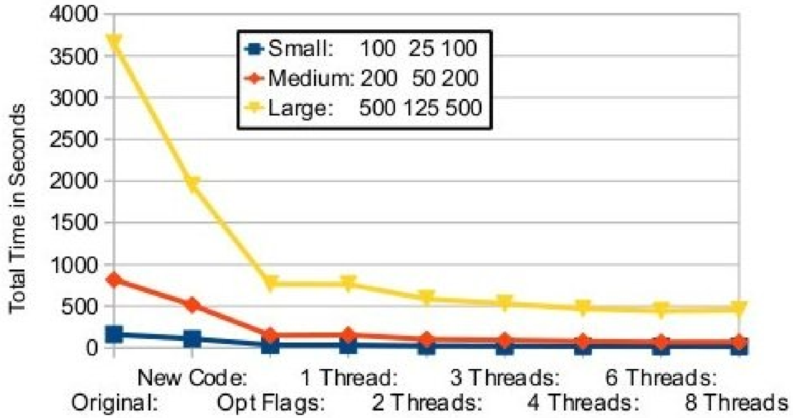
Optimization 3: Parallelization

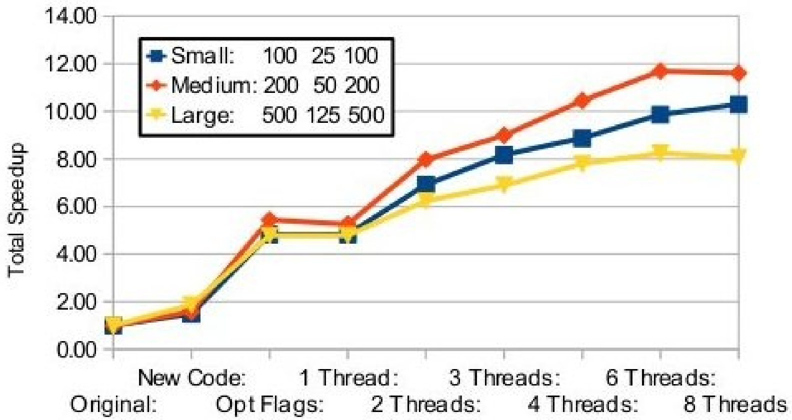
- The construction of the connection matrix has no data dependencies => multi-threading
- Using OpenMP requires only adding one directive and a little bit of code reorganization
- Speedup going from serial to 2 threads: 1.5x
- Speedup levels out at 6-8 threads: 2.5x total

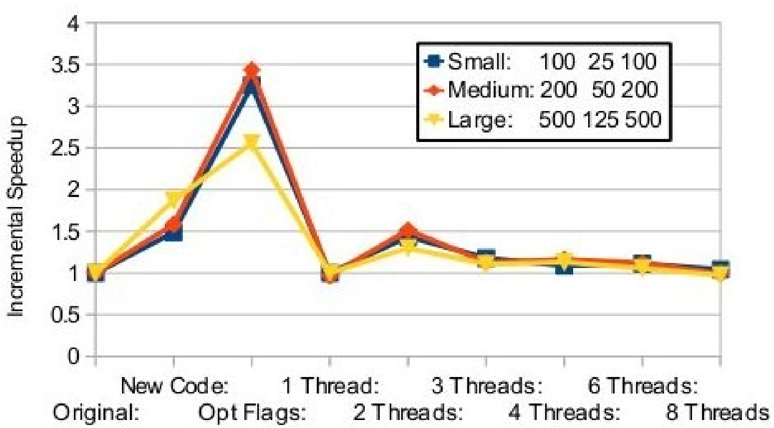
  => very little computation, mostly data access
  => performance limited by memory contention
- Total improvement: 8x-12x with 8 threads

TEMPLE UNIVERSITY®

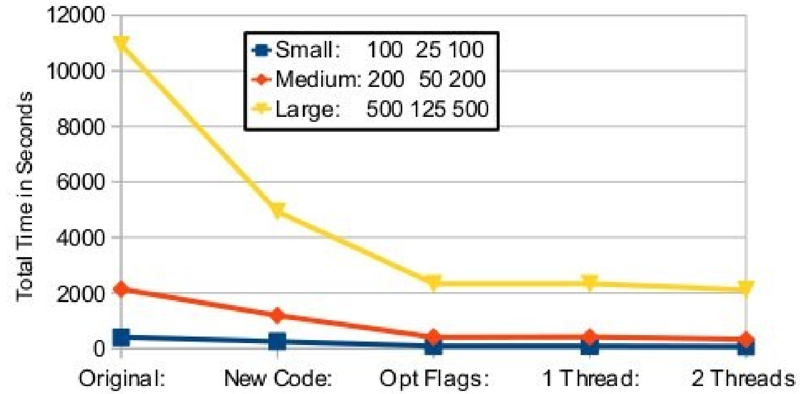
2x Intel Xeon X5677, 3.5GHz
96 GB 1333MHz DDR3 RAM

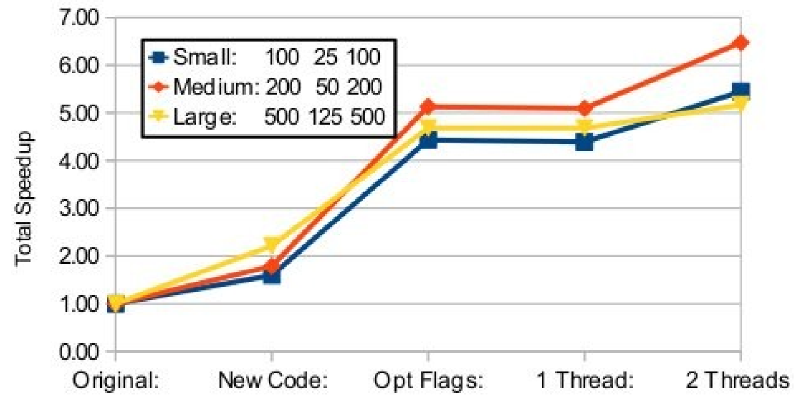


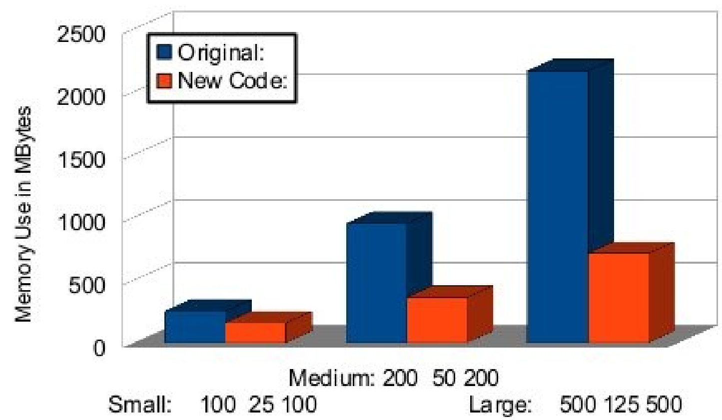




1x Intel Core2 Duo, 1.4GHz
4 GB 800MHz DDR2 RAM







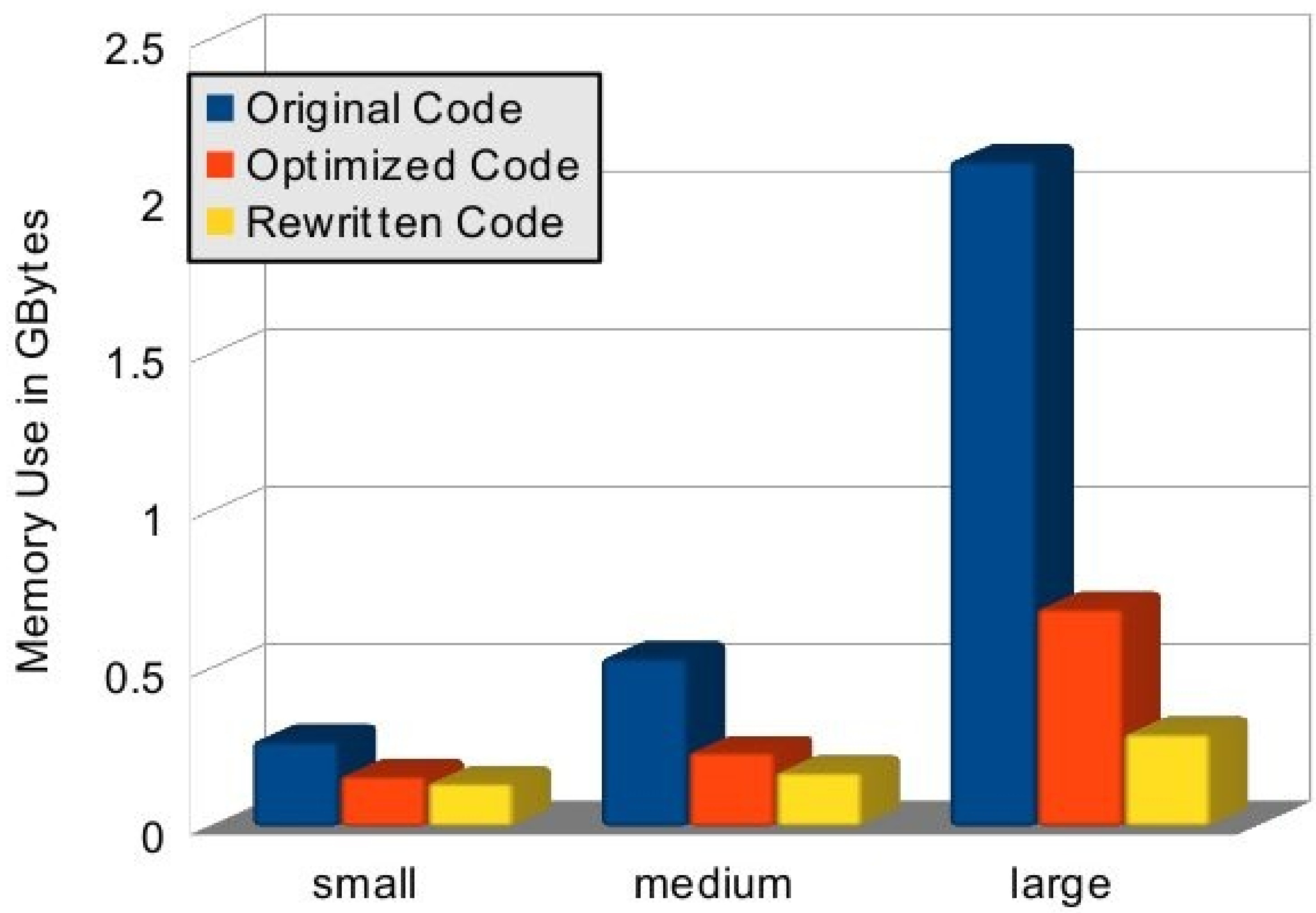
4) Proper Optimization or: The Power of the Rewrite

- 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

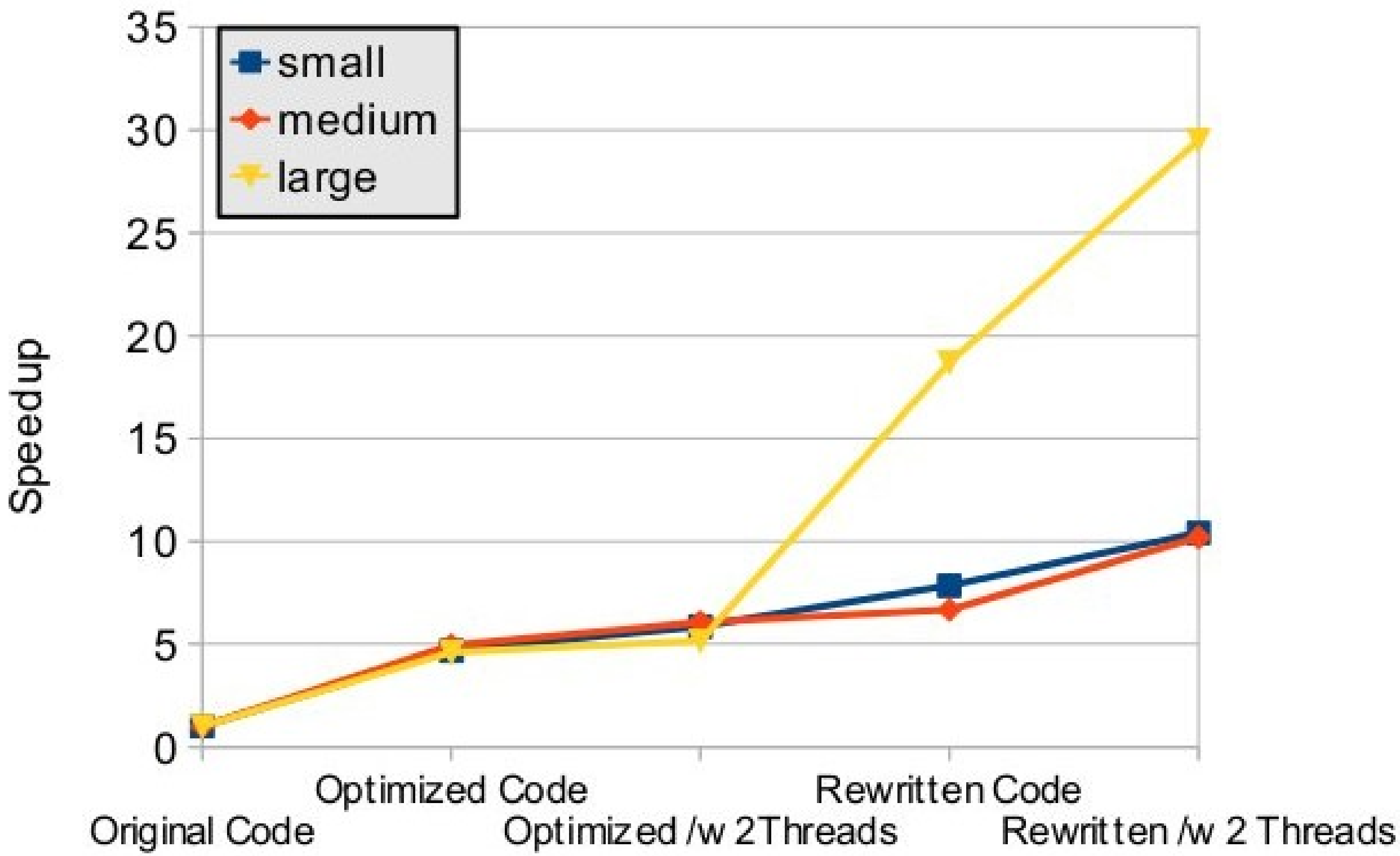
- Rewrite in C++ (more optimization hints than C)
- Use STL container classes
- std::vector<bool> uses single bit per entry
- Single list of structs for all network nodes, all references via index lists (std::vector<int>) => no more need to re-read data
- Leave data in place during pruning, maintain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

Memory Usage After Rewrite

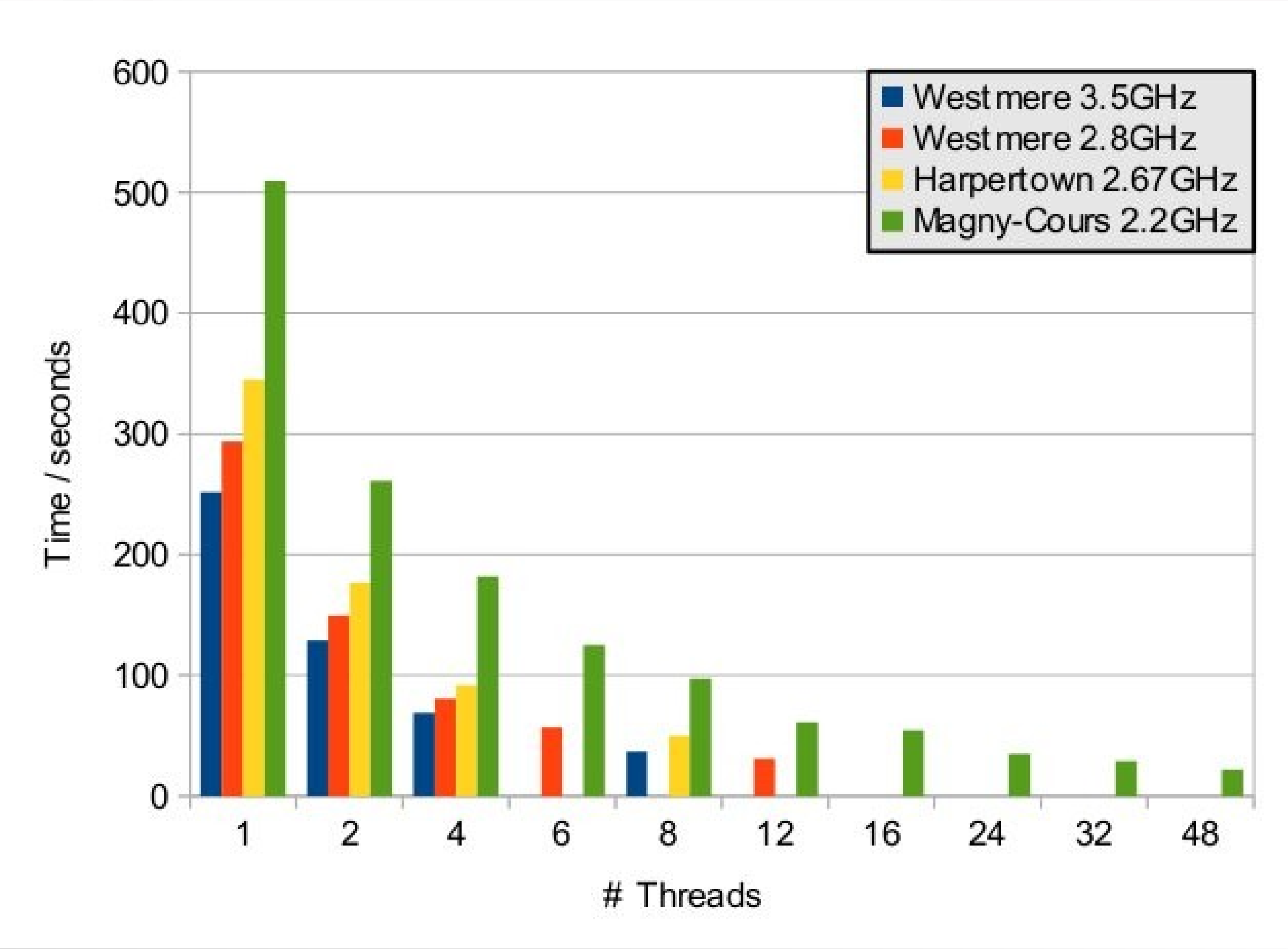


TEMPLE UNIVERSITY®

Performance After Rewrite



Parallel Performance After Rewrite



TEMPLE UNIVERSITY®

5) Conclusions

- “The free lunch is over”: CPU speed levels out
- Moore's law continues, but leads to multi-core, larger caches, vector units, more integration

  => Performance increase now <u>mostly</u> through optimization, vectorization, and parallelization
- Bottleneck has transitioned from CPU clock to memory access and efficient data structures
- We have to abandon our simplified image of a “serial” CPU and “think parallel” instead