Hardware Specific Optimizations

or

What to do if there is no bigger hammer?

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

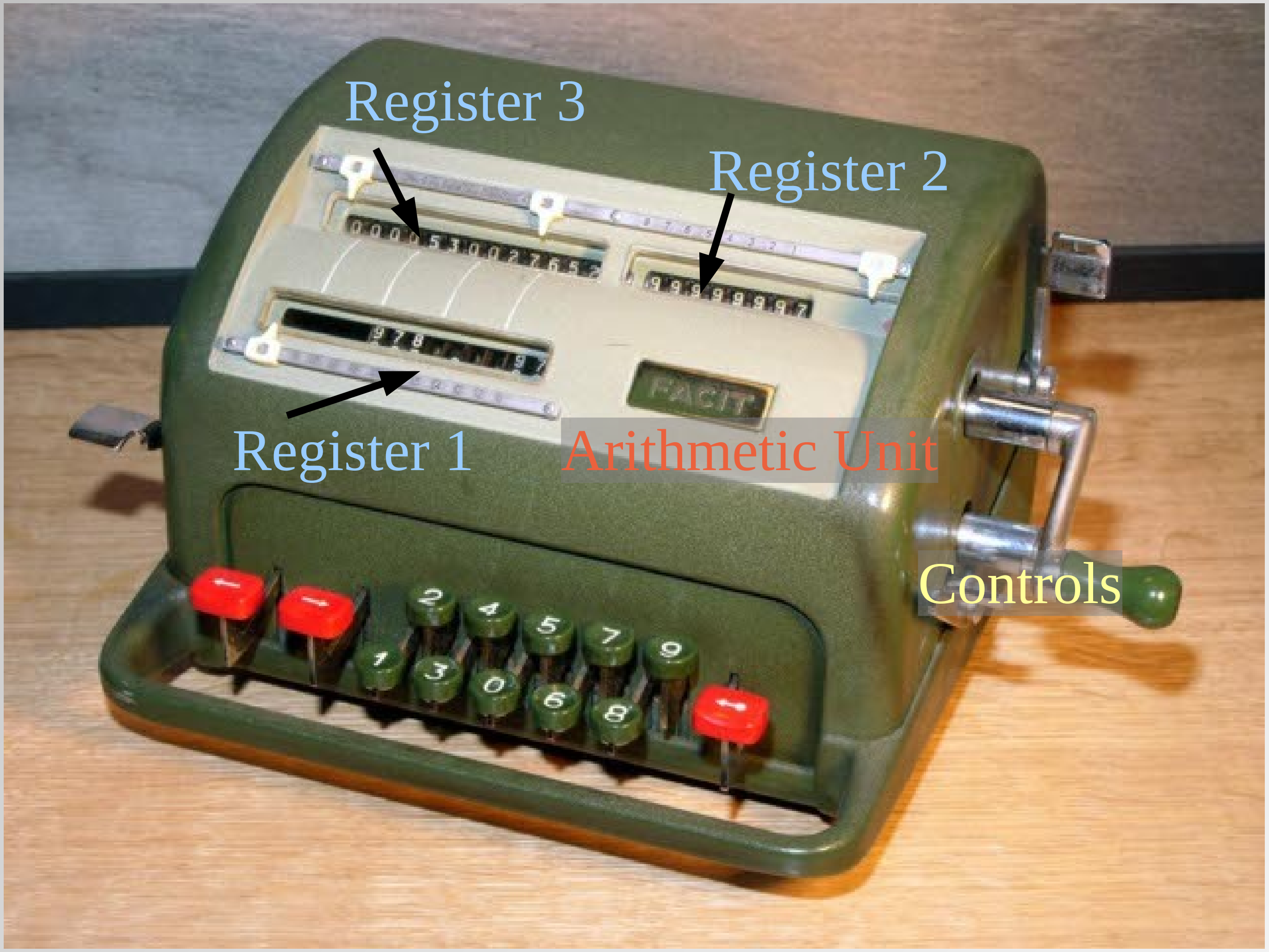
http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

A Bit of History or: How Did We Get Where We Are Now?

- The need for doing (numerical) calculations faster is as old as the underlying math
- There are multiple approaches to address this:
  - Use approximations, when possible
  - Develop more efficient algorithms
  - Get faster hardware
  - Use/write optimized software for your hardware
  - Parallelize
- The focus on each of these changes over time

Improving Hardware Performance



Optimizations:

- Type faster, read faster (Faster I/O)
- Turn handle faster, use faster motor (Higher Clock)
- Build better mechanics, use better technology (Better CPU)

A Bit of History or: How Did We Get Where We Are Now?

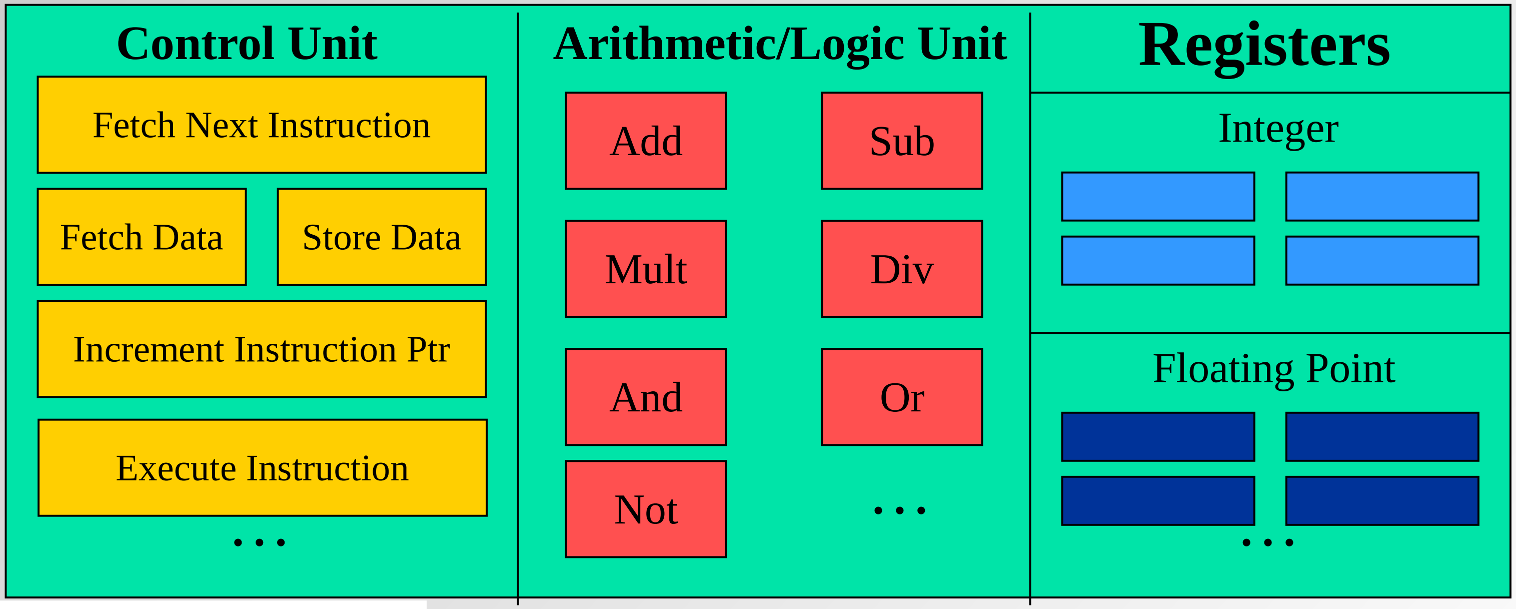
- The need for doing (numerical) calculations faster is as old as the underlying math
- There are multiple approaches to address this:
  - Use approximations, when possible
  - Develop more efficient algorithms
  - Get faster hardware
  - Use/write optimized software for your hardware
  - Parallelize
- The focus on each of these changes over time

Where Are The Problems?

- Increasing clock rates is technologically difficult => multi-core architectures => parallelization
- Compilers <u>can</u> optimize for vector units and superscalar, pipelined CPUs, but <u>only</u> if the original code (and its language) allows it => many codes underutilize current CPUs
- With multi-core and non-uniform memory access, performance is often limited by I/O => data structures and access patterns matter
- We are not used to “think like a CPU”

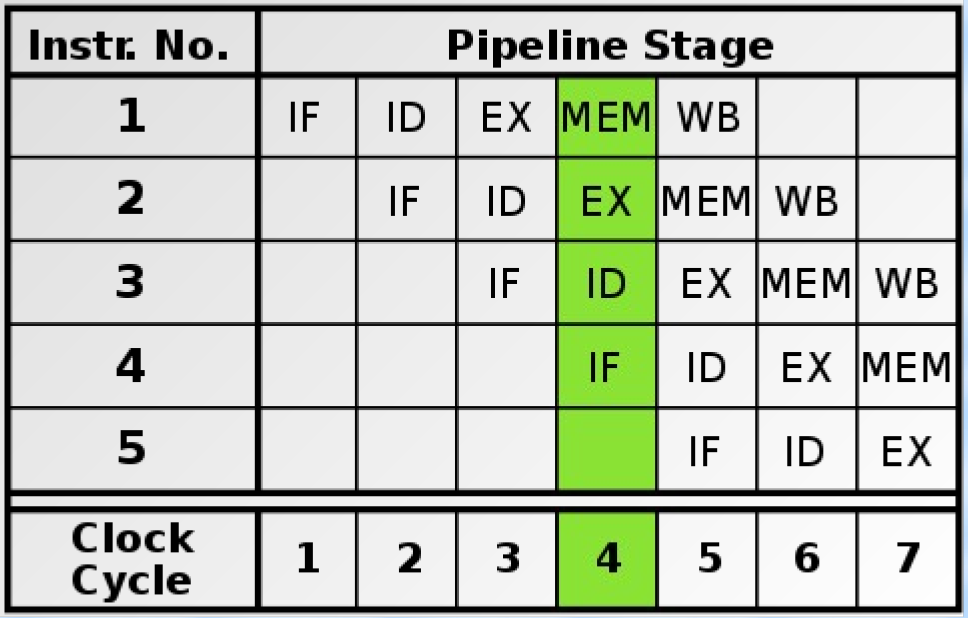
A Simple CPU

- The basic CPU design is not much different from the mechanical calculator.
- Data still needs to be fetched into registers for the CPU to be able to operate on it.



CPU Pipeline

- One CPU “operation” has multiple steps/stages: fetch instr, decode instr, execute instr, memory lookup, write back => multiple functional units
- Using a pipeline allows for a faster CPU clock => like assembly line
- Dependencies and branches may force CPU to stall pipeline
- Complex operations usually not pipelined



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

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

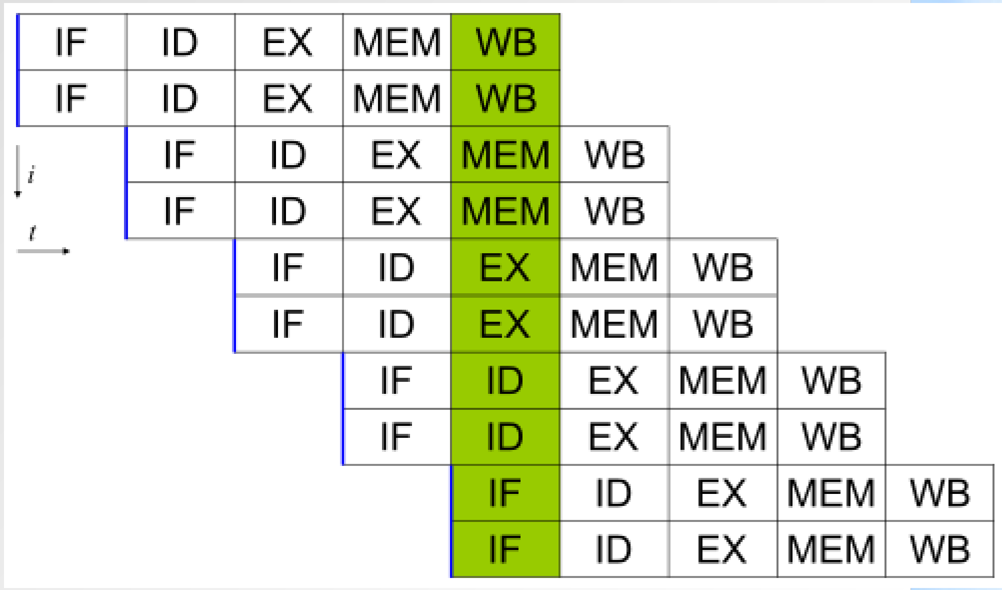
`z= z1 + z2;`

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

Superscalar CPU design

- Superscalar CPU => <u>instruction level parallelism</u>
- Redundant functional units in single CPU core
  => multiple instructions executed at same time
  => typically combined with pipelined CPU design
- This is **not** SIMD!
- How to program for this:
  - write simple code
  - no data dependencies
  - avoid branches
  - compiler optimization



Superscalar & Pipelined CPU Execution

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

Actual steps:

z1 = a * b;

z2 = c * d;

Start data load for next command

z= z1 + z2;

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**
3. Multiply **R5 = R3 * R4**
4. Add **R6 = R2 + R5**
5. Store **R6** into **z**

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

Superscalar & Pipelined Loop

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

1. Load `a[0]` into `R0` and load `b[0]` into `R1`
2. Multiply `R2 = R0 * R1` and load `c[0]` into `R3` and load `d[0]` into `R4`
3. Multiply `R5 = R3 * R4` and load `a[1]` into `R0` and load `b[1]` into `R1`
4. Add `R6 = R2 + R5` and load `c[1]` into `R3` and load `d[1]` into `R4`
5. Store `R6` into `z[0]` and multiply `R2 = R0 * R1` and multiply `R5 = R3 * R4` and load `a[2]` into `R0` and load `b[2]` into `R1`

Repeat steps 4. and 5. with increasing index until done

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.)

Fast and Slow Operations

- Fast (0.5x-1x): add, subtract, multiply
- Medium (5-10x): divide, modulus, sqrt()
- Slow (20-50x): most transcendental functions
- Very slow (>100x): power (x^y for real x and y)

Often **only** the fastest operations are pipelined, so code will be the fastest when using only add and multiply => linear algebra
=> BLAS (= Basic Linear Algebra Subroutines)
plus LAPACK (Linear Algebra Package)

Simple Optimization Techniques

(so simple a ~~cavemen~~ compiler could do it)

- Scalar optimizations, for example
  - Copy propagation
  - Constant folding, 'dead code' removal
  - Strength reduction
  - Common subexpression elimination
  - Variable renaming
- Loop Optimizations: loop unrolling, vectorization
- Inlining, Replacing code with a faster equivalent
  => prefer readability, let the compiler do it

Copy Propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

$$\Big\downarrow \text{Compile}$$

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before

```
add = 100;
aug = 200;
sum = add + aug;
```

After

```
sum = 300;
```

`sum` is the sum of two constants. The compiler can precalculate the result (once) at compile time and eliminate code that would otherwise need to be executed at (every) run time.

Strength Reduction

Before

```
x = pow(y, 2.0);
a = c / 2.0;
```

After

```
x = y * y;
a = c * 0.5;
```

Raising one value to the power of another, or dividing, is more expensive than multiplying.

If the compiler can tell that the power is a small integer, or that the denominator is a constant, it will use multiplication instead.

Easier to do with intrinsic functions (cf. Fortran).

Common Subexpression Elimination

Before

```
d = c * (a / b);
e = (a / b) * 2.0;
```

After

```
adivb = a / b;
d = c * adivb;
e = adivb * 2.0;
```

The subexpression `(a / b)` occurs in both assignment statements, so there's no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate, or the resulting code requires the use of less registers.

Variable Renaming

Before

```
x = y * z;
q = r + x * 2;
x = a + b;
```

After

```
x0 = y * z;
q = r + x0 * 2;
x = a + b;
```

The original code has an **output dependency**, while the new code **doesn't** – but the final value of **`x`** is still correct.

Hoisting Loop Invariant Code

Code that doesn’t change inside the loop is known as ***loop invariant***. It doesn’t need to be calculated over and over.

Before

```
DO i = 1, n
  a(i) = b(i) + c * d
  e = g(n)
END DO
```

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO
e = g(n)
```

Loop Unrolling

Before

```
DO i = 1, n
  a(i) = a(i)+b(i)
END DO
```

After

```
DO i = 1, n, 4
  a(i)   = a(i)  +b(i)
  a(i+1) = a(i+1)+b(i+1)
  a(i+2) = a(i+2)+b(i+2)
  a(i+3) = a(i+3)+b(i+3)
END DO
```

You generally **shouldn’t** unroll by hand.
Compilers are much more reliable.

Loop Interchange

Before

```
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

After

```
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements **`a(i,j)`** and **`a(i+1,j)`** are near each other in memory, while **`a(i,j+1)`** may be far, so it makes sense to make the **`i`** loop be the inner loop. (This is reversed in C, C++)

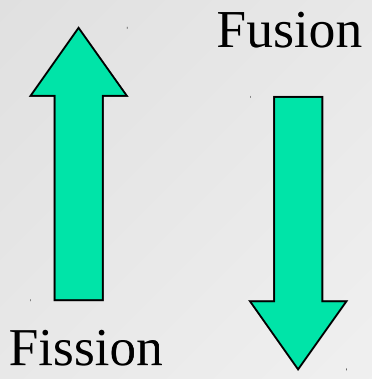
Loop Fusion / Fission

```
DO i = 1, n
   a(i) = b(i) + 1
END DO
DO i = 1, n
   c(i) = a(i) / 2
END DO
DO i = 1, n
   d(i) = 1 / c(i)
END DO
```

Separate

```
DO i = 1, n
   a(i) = b(i) + 1
   c(i) = a(i) / 2
   d(i) = 1 / c(i)
END DO
```

Together



Fusion: fewer branches (combine with unrolling).
fewer total memory references.
Fission: smaller cache footprint

Inlining

<u>**Before**</u>

```
DO i = 1, n
  a(i) = func(i)
END DO
…
REAL FUNCTION func (x)
  …
  func = x * 3
END FUNCTION func
```

<u>**After**</u>

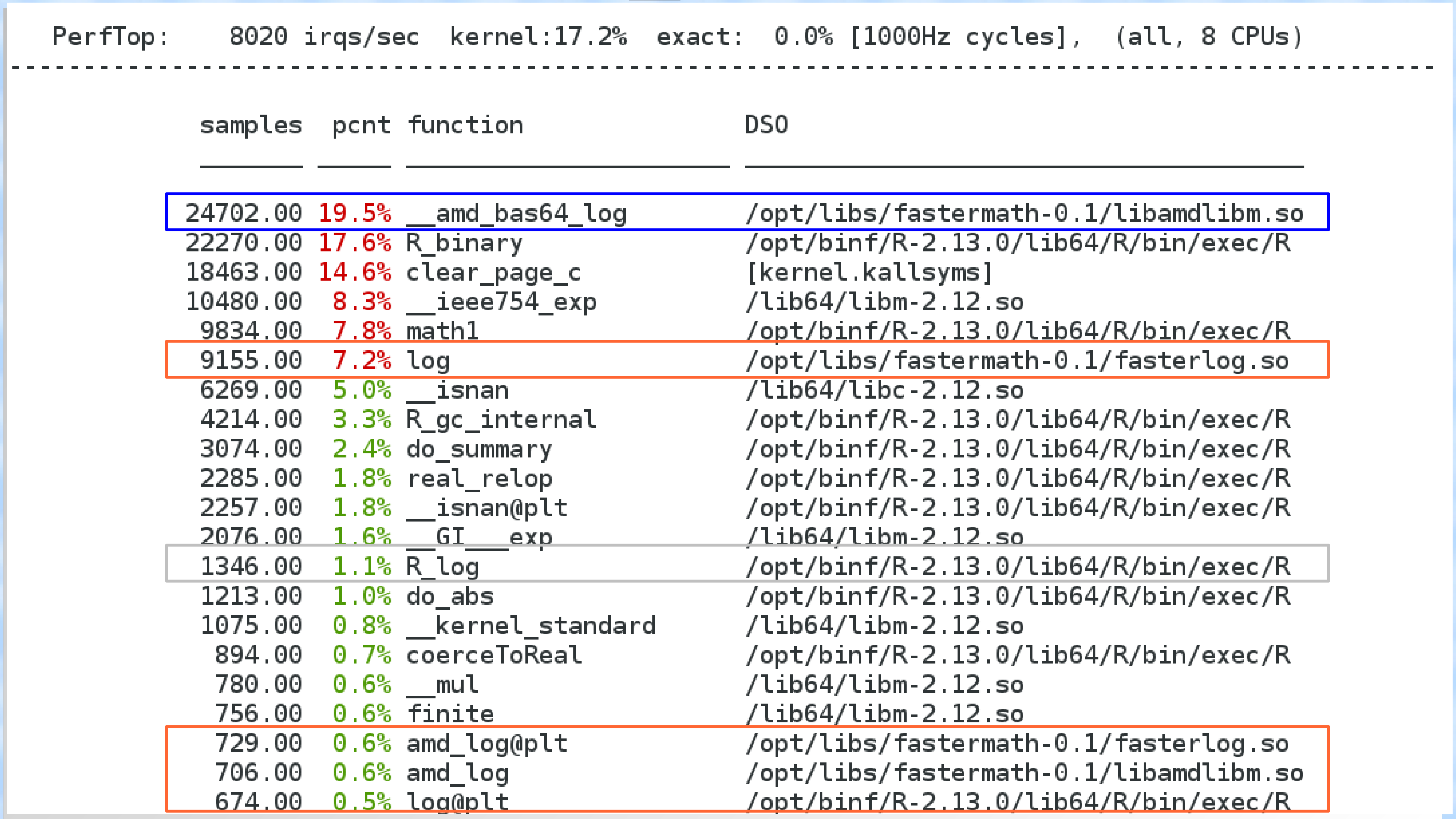
```
DO i = 1, n
  a(i) = i * 3
END DO
```

When a function or subroutine is <u>***inlined***</u>, its contents are transferred directly into the calling routine, and thus eliminating the overhead of making the call.
=> compilers use an inline library at high optimization
=> math is instrinsic in Fortran => better for compiler

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; few offer a “dynamic dispatch”, i.e. a selection between alternatives at run time (e.g. MKL)

Replacing Math Functions with LD_PRELOAD



```
  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
```

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 <u>very</u> usage and application specific
- 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 approximation of same error

  => 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”

Fast Implementation of exp2()

- Range reduction: $x = f + n; \quad n \in \mathbb{Z}, -0.5 \leq f < 0.5$
  $$2^x = 2^{f+n} = 2^f \cdot 2^n$$
- Get 2^n from setting IEEE-754 exponent: zero mantissa bits (=1), exponent is *n* + 1023
- 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 log2(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*)
- Precomputed spline table at aligned address

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

- Example from the “HPC Helpdesk” @ TempleU
- User requests access to HPC resource because his self-written program needs to much memory and runs too slow
- Next the users asks for parallel programming courses 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

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 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 empty

Optimization 1: Reduce Memory

- The by far most time consuming step is the calculation of a “connection matrix” of the selected nodes
- The matrix elements are either 1 if two nodes are connected or 0, if not.
- 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. no 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

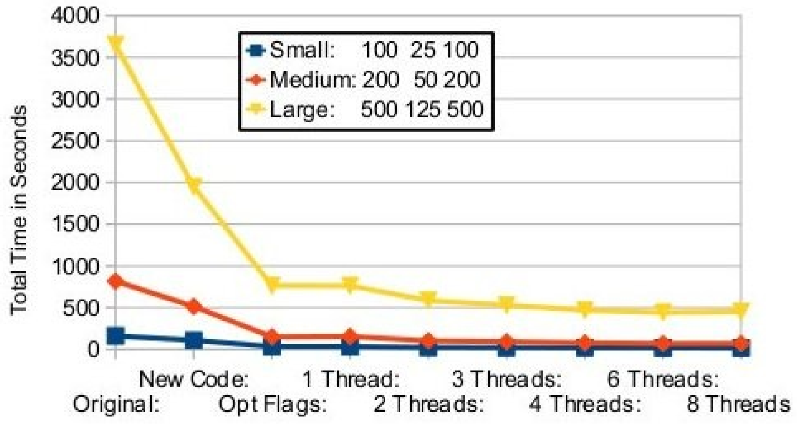
Optimization 3: Parallelization

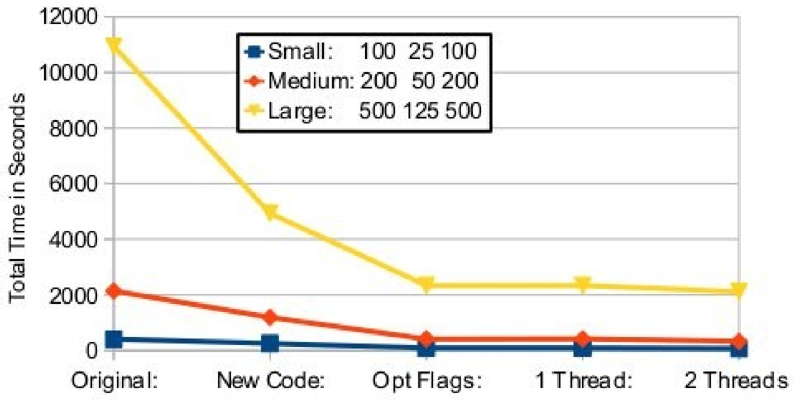
- The construction of the connection matrix has no data dependencies => multi-threading
- Using OpenMP requires only adding one directive and a little bit of code reorganization
- Speedup going from serial to 2 threads: 1.5x
- Speedup levels out at 6-8 threads: 2.5x total

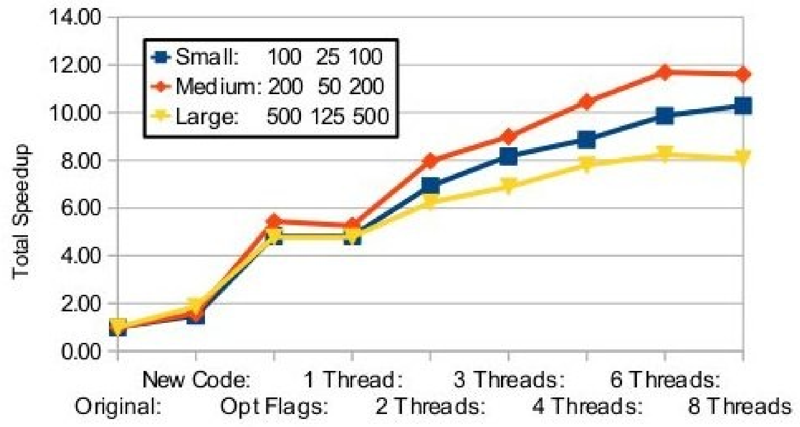
  => very little computation, mostly data access
  => performance limited by memory contention
- Total improvement: 8x-12x with 8 threads

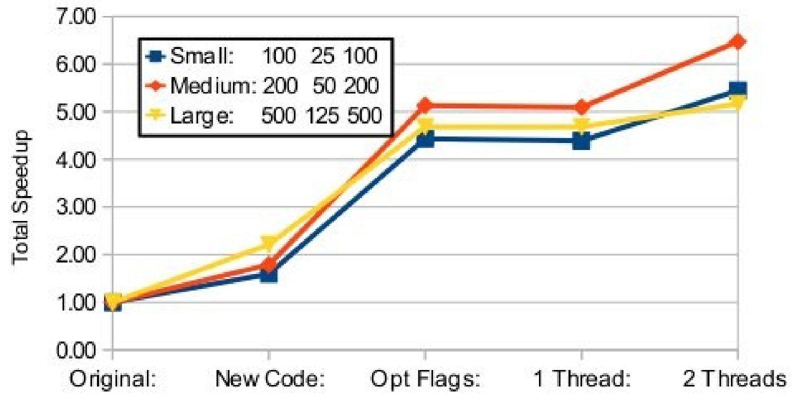
2x Intel Xeon X5677, 3.5GHz
96 GB 1333MHz DDR3 RAM

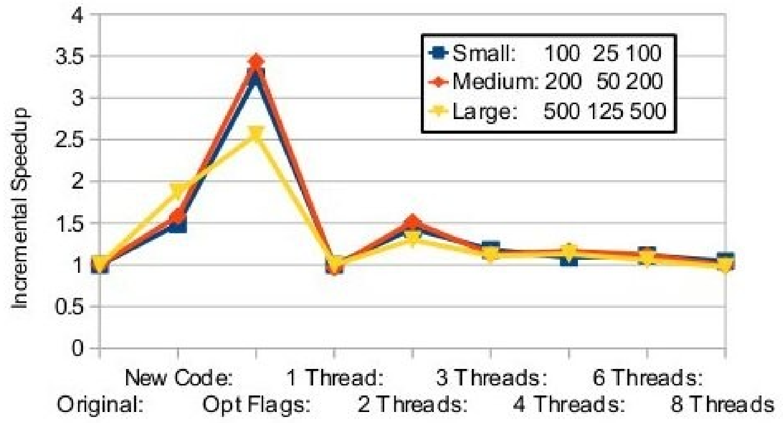
1x Intel Core2 Duo, 1.4GHz
4 GB 800MHz DDR2 RAM

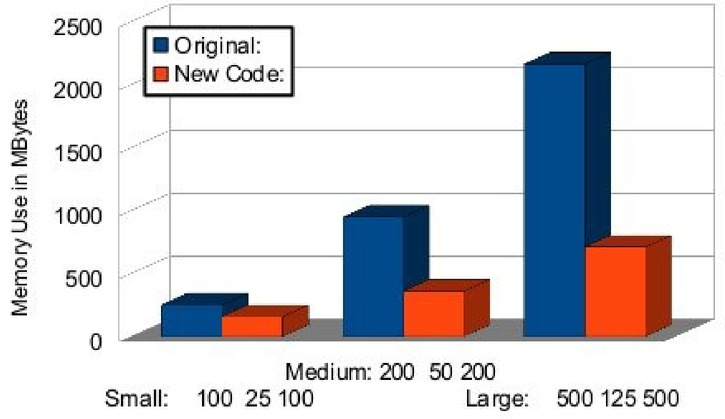












d

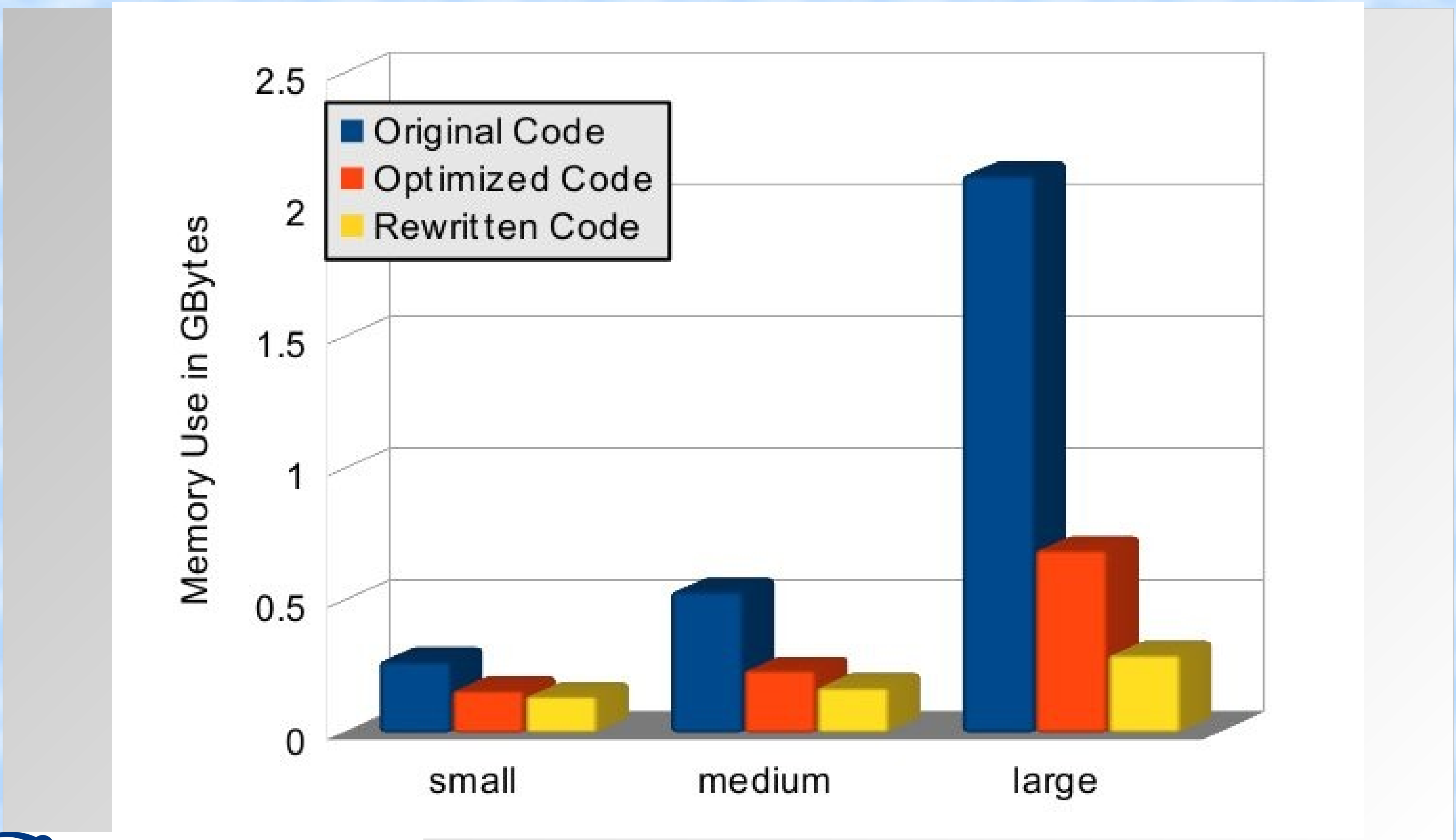
Proper Optimization or: The Power of the Rewrite

- Quick'n'dirty optimizations of T-CLAP resulted in significant improvements in a short time
- More optimization potential with a full rewrite:
  - Connection matrix information requires only 1 bit => reduce storage need by factor of 8 (vs. `char`)
  - Network represented by structs and lists of pointers => multiple pointers refer to the same data => pointers require more storage in 64-bit mode
  - Pruning implementation uses memmove() to compact matrix rows => performance bottleneck for large data ($O(N^2)$)

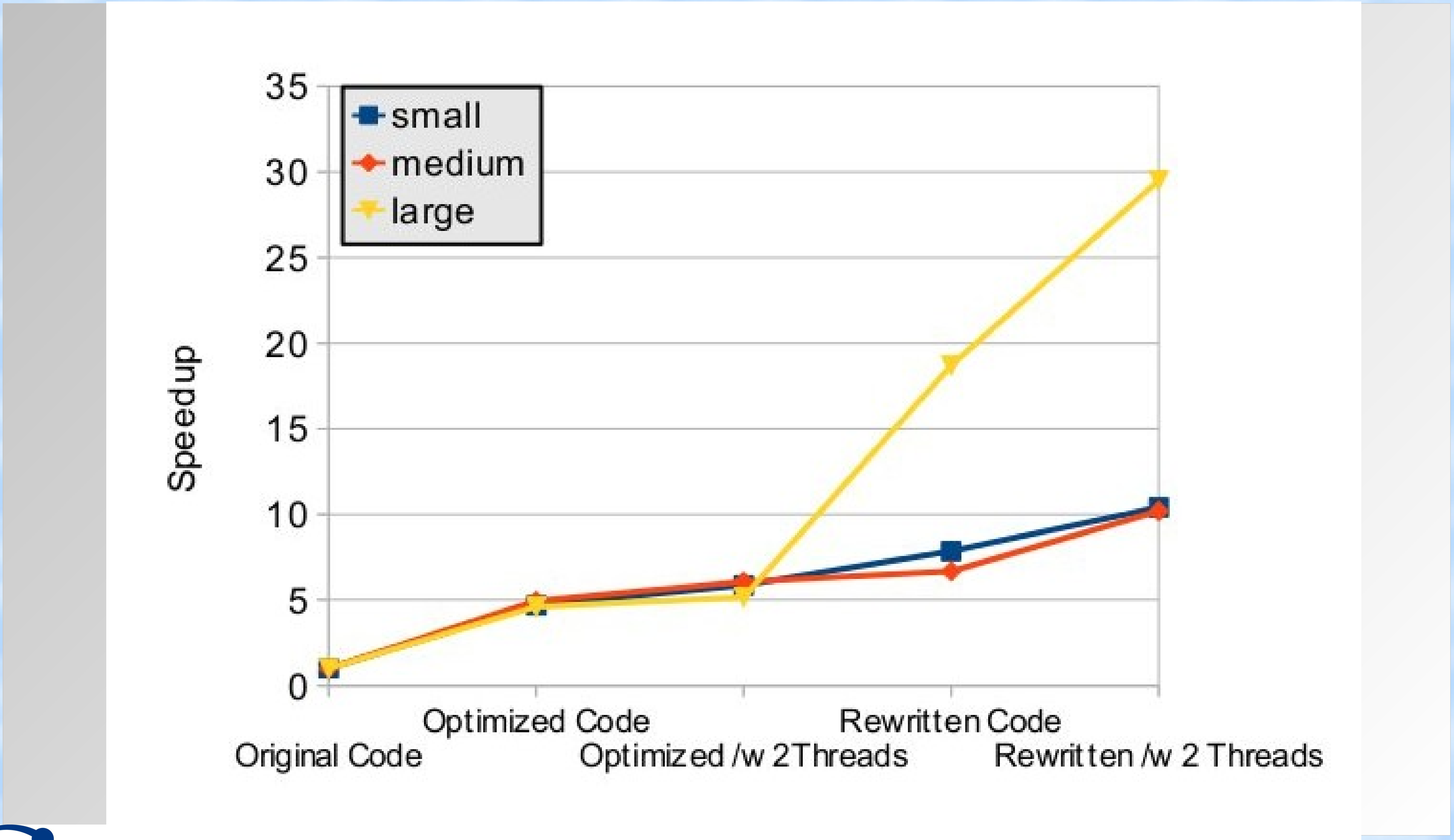
The Rewrite

- 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>)
- Leave data in place during pruning, maintain lists of valid rows and columns instead
- Avoid some redundant operations
- Rewrite piece-by-piece to reproduce original

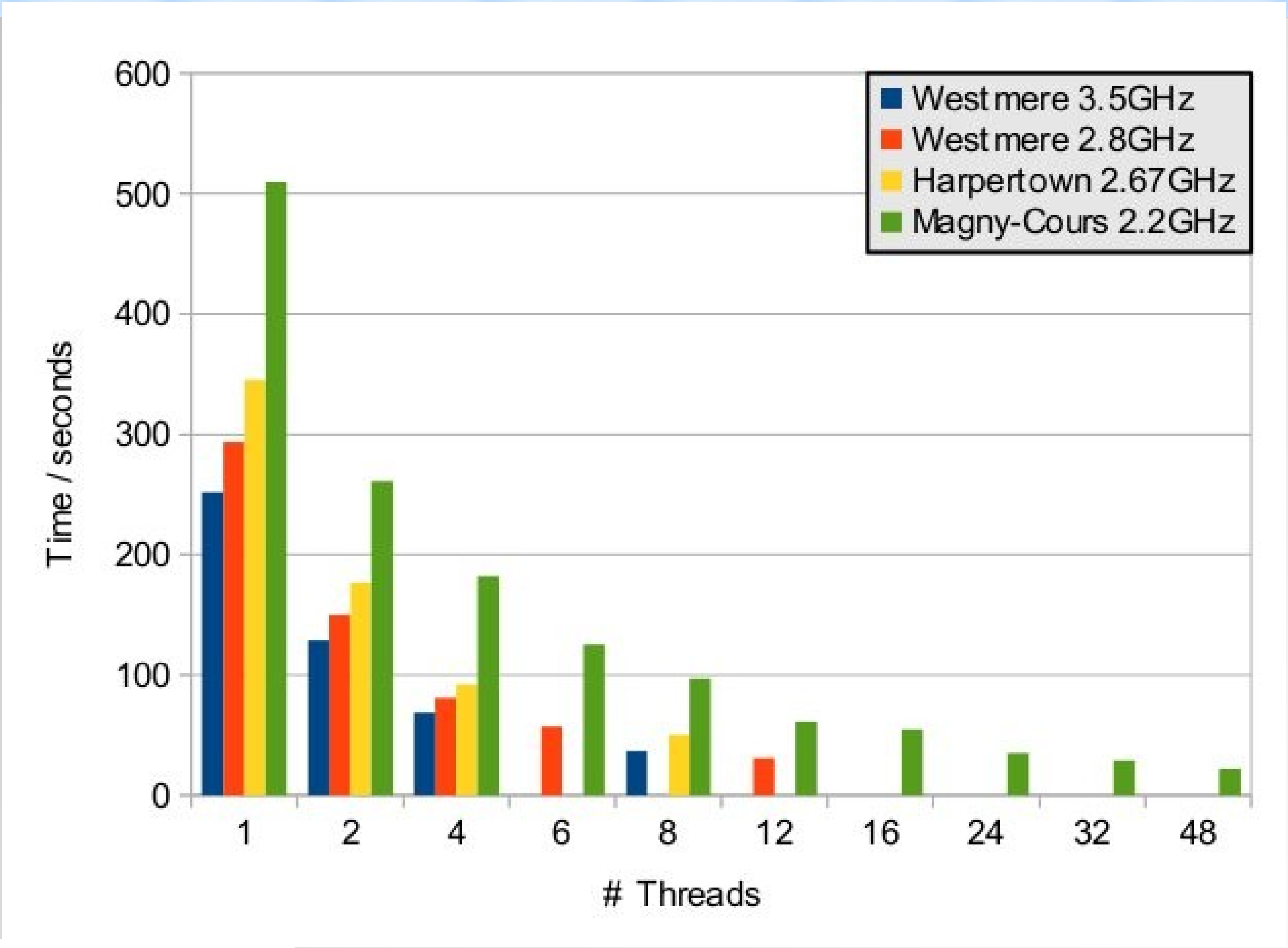
Memory Usage After Rewrite



Performance After Rewrite



Parallel Performance After Rewrite



6) Conclusions

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

  => Performance increase now mostly through better algorithms, optimization, vectorization, and parallelization
- Bottleneck has transitioned from CPU speed to memory access and efficient data structures
- Optimization potential may be found in unusual places and small changes can go far

Hardware Specific Optimizations

or

What to do if there is no bigger hammer?

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu