# Go Multicore Series:

## Understanding Memory in a Multicore World, Part 2:

## Software Tools for Improving Cache Perf

**Joe Hummel, PhD** | http://www.joehummel.net/freescale.html

FTF 2014: FTF-SDS-F0099

External Use

# Agenda

- Motivation by example: matrix multiplication
- The many levels of caching
- **cachegrind**: a tool for monitoring the cache
- Other important implications of caching
- Cache coherence vs. memory consistency
- High Performance Computing = …

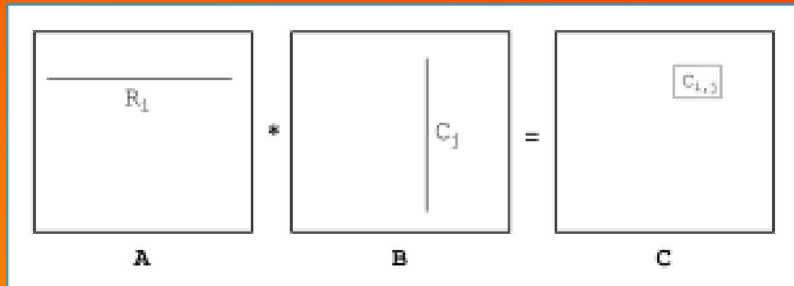
# Introductions

- **Your speaker…**
- **Joe Hummel, PhD**
  - ***PhD***: UC-Irvine, in High Performance Computing
  - ***Professor***: U. of Illinois, Chicago
  - ***Trainer***: Pluralsight, LLC

  - ***Consultant***: Joe Hummel, Inc.
  - *Microsoft MVP C++*

  - *Married, one daughter adopted from China (just turned 12!)*
  - *Avid sailor*

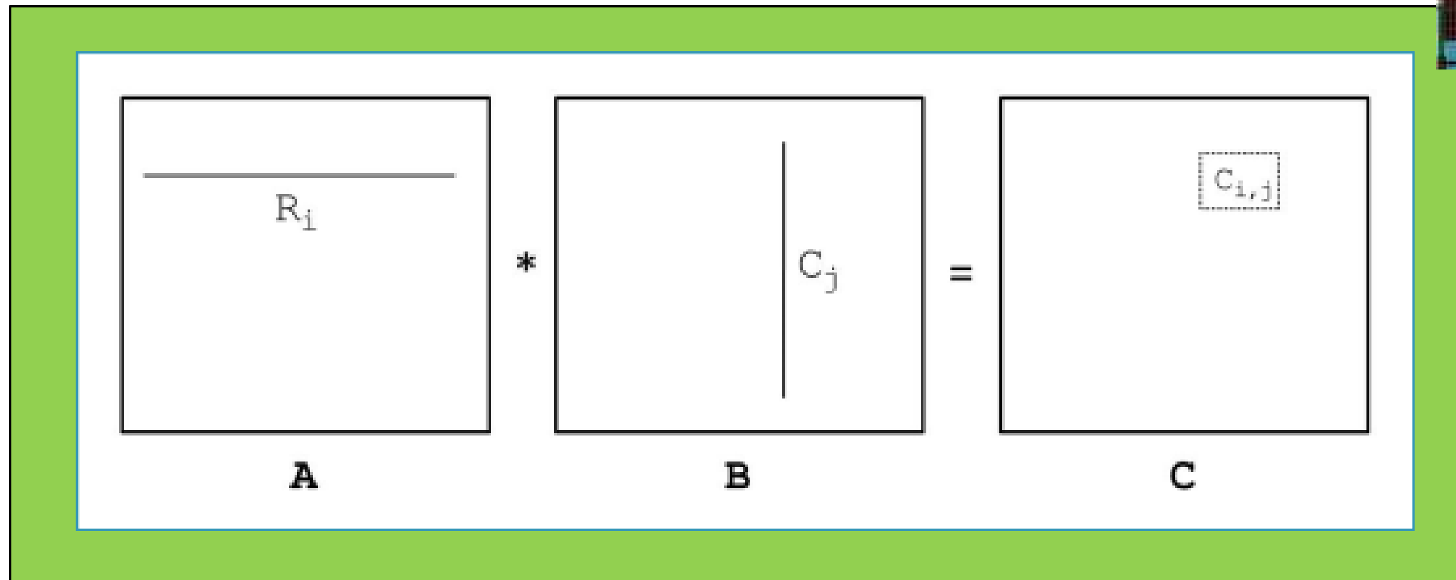
# Example

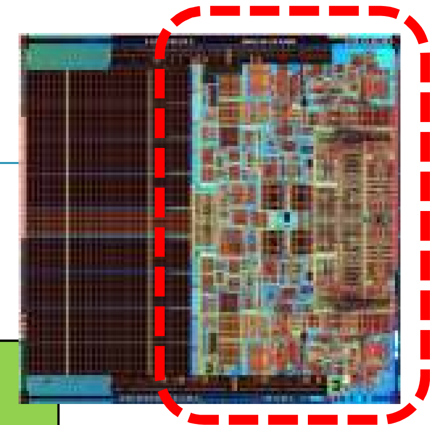


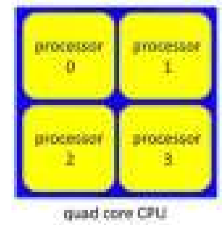
**Matrix multiplication…**

# Motivation by example

- **Matrix Multiplication**

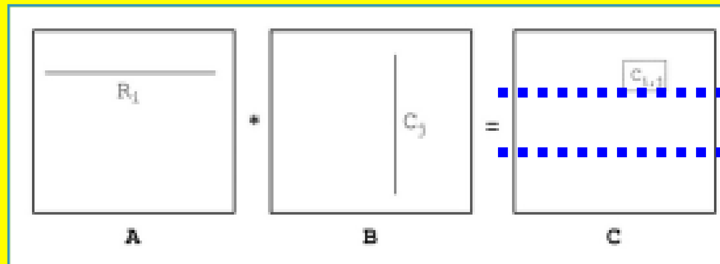


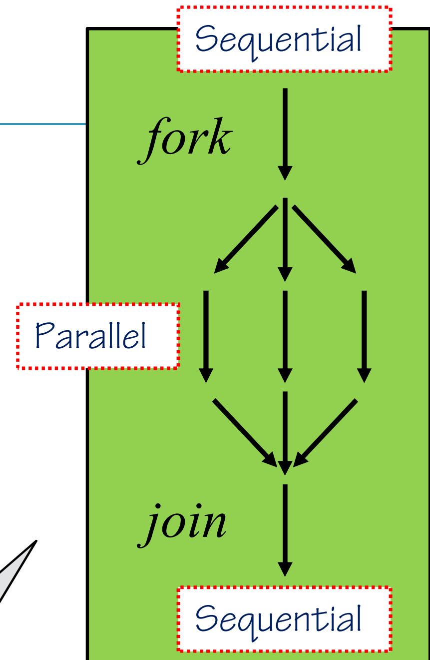





# Sequential to parallel…

```
//
// Naïve, triply-nested sequential solution:
//
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```



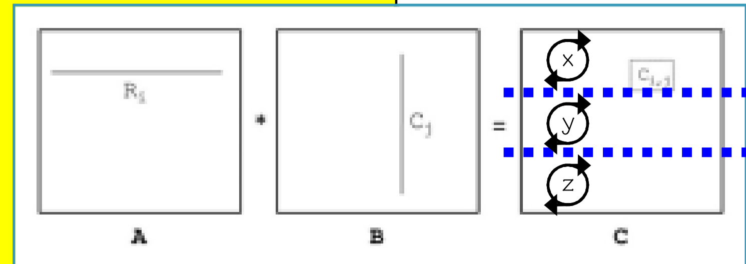



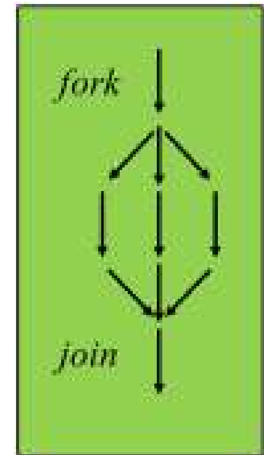

***Common pattern for parallelism**:*
*Structured (or "Fork-Join") Parallelism*

# Multithreading using OpenMP

- **OpenMP == Open Multiprocessing standard**
  – *Provide directive, and compiler multithreads for you…*

```
//
// Naïve parallel solution using OpenMP:  result is structured parallelism, with
// static division of workload by row.
//
#pragma omp parallel for   // parallelize outer loop ==> by rows:
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```

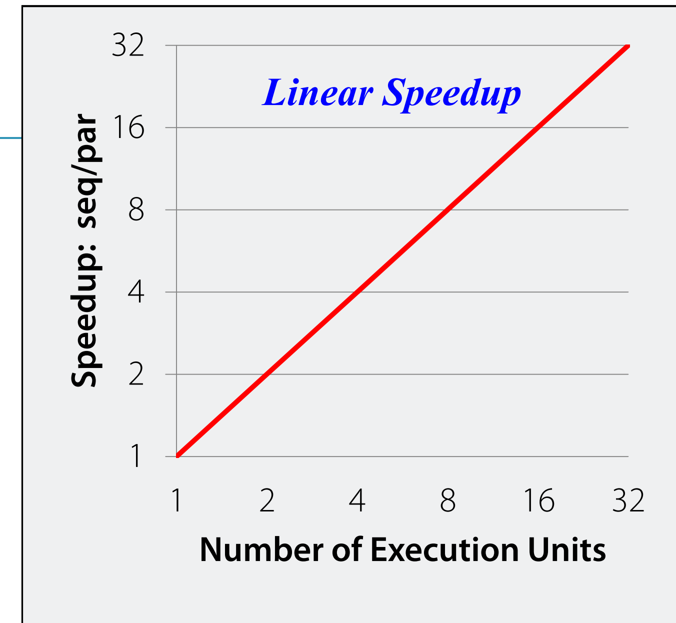


# Results?

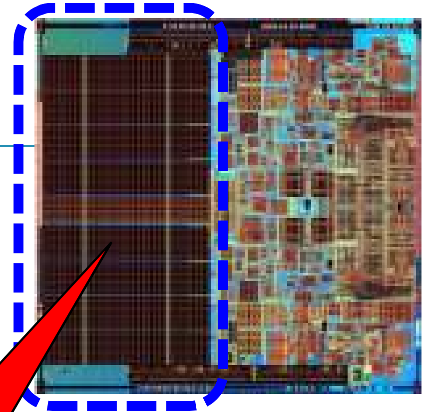
- **Very good!**
  - *matrix multiplication is "embarrassingly parallel"*
  - *linear speedup — 2x on 2 cores, 4x on 4 cores, ...*

| *Version* | Cores | Time (secs) | Speedup |
|---|---|---|---|
| Sequential | 1 | 30 | |
| OpenMP | 4 | 7.6 | **3.9** |
| | | | |



# But wait…

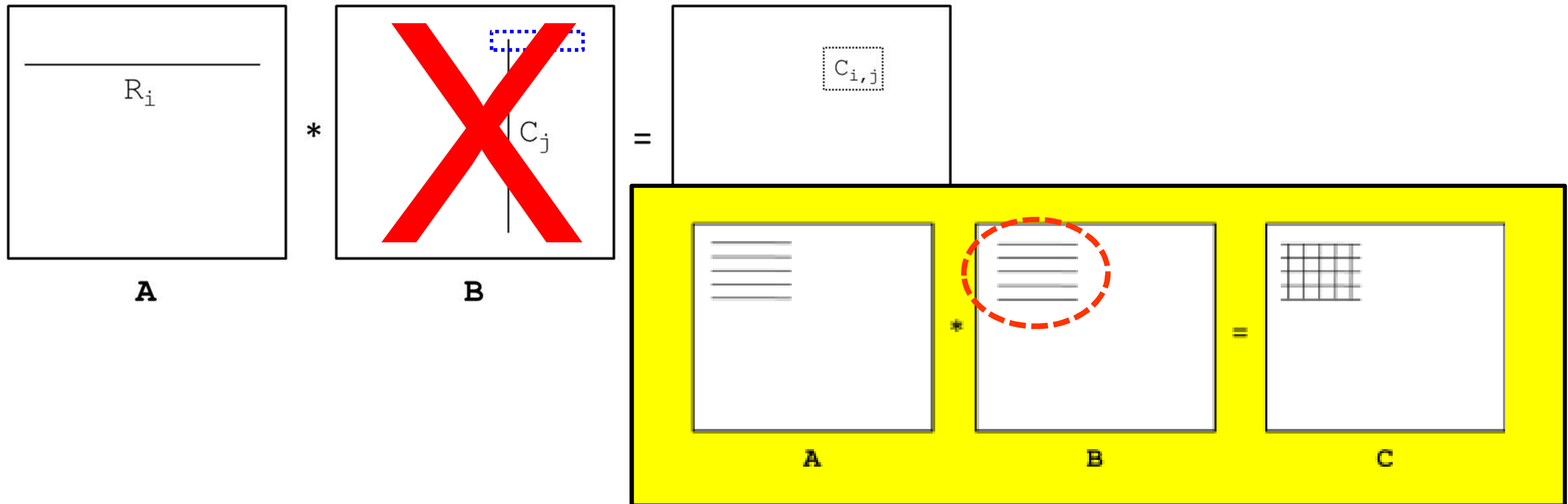
- **What's the other half of the chip?**
  - *cache!*
- **Are we using it effectively?**
  - *we are not…*



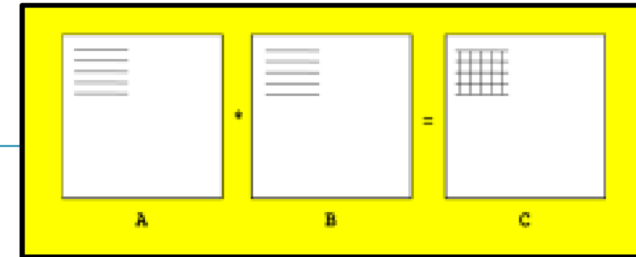
*Memory cache…*

# Cache-friendly matrix multiplication

- **No one solves MM using the naïve algorithm**
  - *horrible cache behavior*



# Step 1: loop interchange



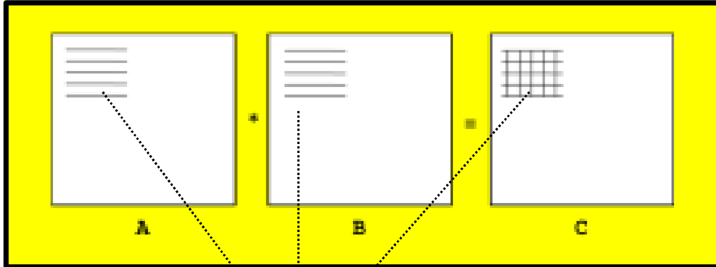

```
  for (int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
        C[i][j] = 0.0;

#pragma omp parallel for
  for (int i = 0; i < N; i++)
     for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
           C[i][j] += (A[i][k] * B[k][j]);
```

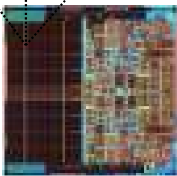
*Another factor of 2-10x improvement!*

# Step 2:  blocking

- **Block size based on size of cache closest to core — level 1 ("L1")**
  - *largest integer BS such that*

$$\frac{BS * BS * 3 * sizeof(double)}{cache\ size} < 1$$



```
#pragma omp parallel for
for (int jj=0; jj<N; jj+=BS)   // for each column block:
{
  int jjEND = Min(jj+BS, N);

  // initialize:
  for (int i=0; i<N; i++)
    for (int j=jj; j < jjEND; j++)
      C[i][j] = 0.0;

  // block multiply:
  for (int kk=0; kk<N; kk+=BS)   // for each row block:
  {
    int kkEND = Min(kk+BS, N);

     for (int i=0; i<N; i++)
      for (int k=kk; k < kkEND; k++)
         for (int j=jj; j < jjEND; j++)
           C[i][j] += (A[i][k] * B[k][j]);
  }
}
```

# Results?

- **Caching impacts all programs, sequential and parallel…**

| *Version* | Cores | Time (secs) | Speedup |
|---|---|---|---|
| **Sequential** | | | |
| Naive | 1 | 30 | |
| Blocked | 1 | 3 | **10** |
| **OpenMP** | | | |
| Naïve | 4 | 7.6 | **3.9** |
| Blocked | 4 | 0.8 | **37.5** |

# High-Performance Computing

- HPC
- Parallelism alone is not enough…

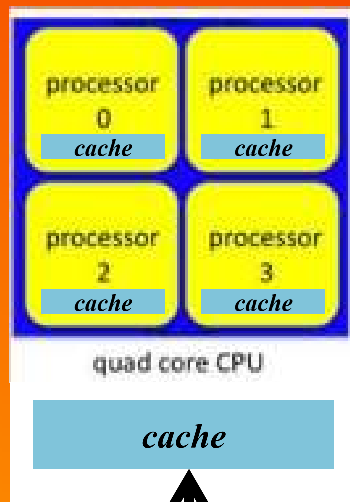
$$HPC == Parallelism + Memory\ Hierarchy - Contention$$

Expose parallelism

Maximize data locality:
- network
- disk
- RAM
- cache
- core

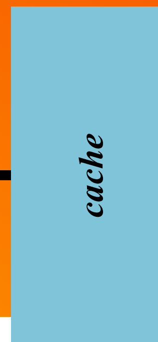
Minimize interaction:
- false sharing
- locking
- synchronization

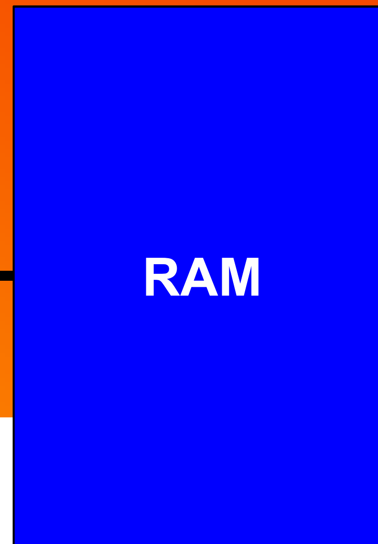
# Monitoring the Cache




cache

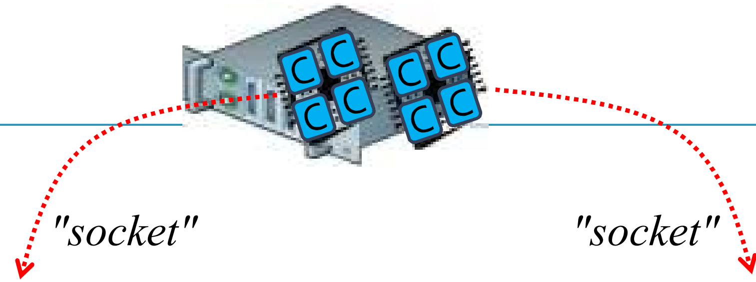
cachegrind…

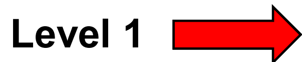


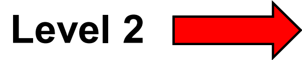



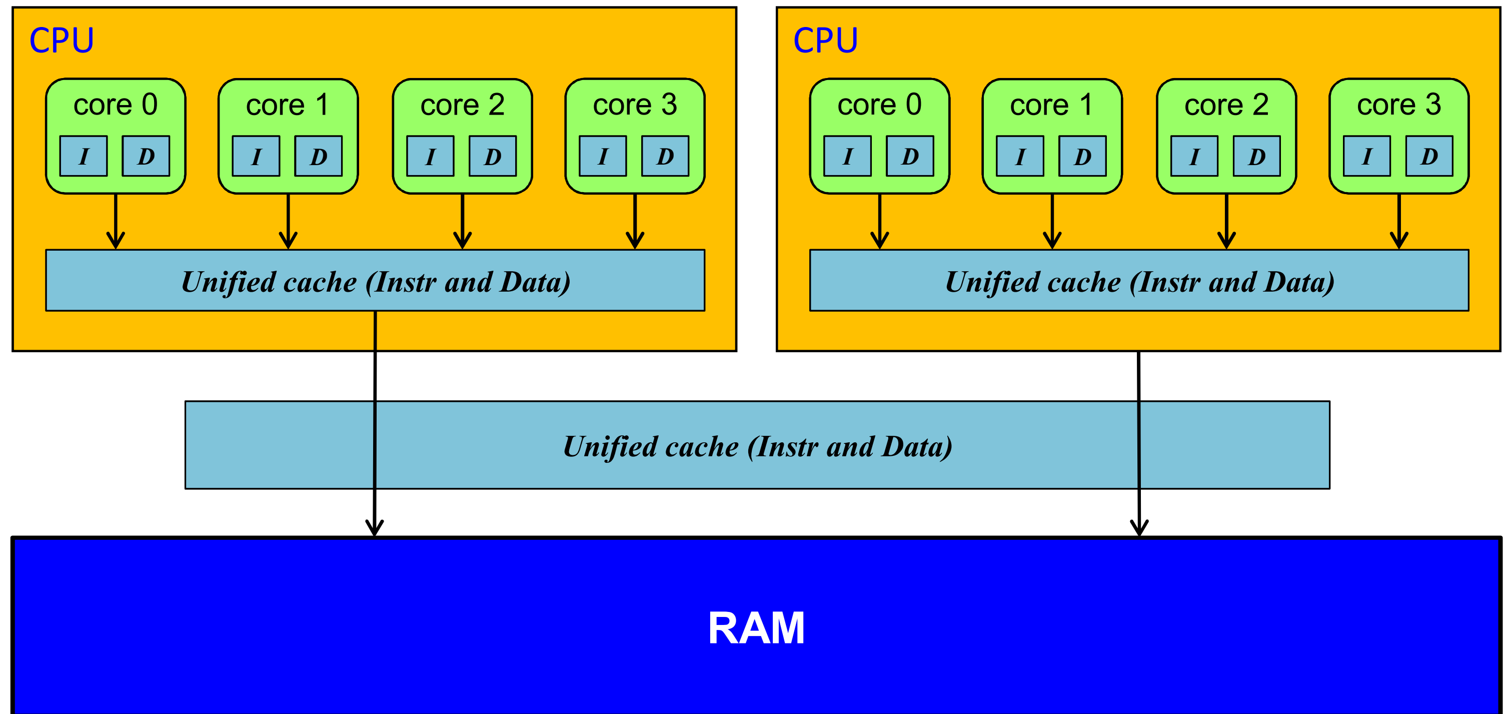
## Modern caching

- **Multi-level:**






*Typical sizes and access times (in CPU cycles):*

- ***L1***: *32KB, 1 cycle*
- ***L2***: *512KB, 10 cycles*
- ***L3***: *4MB, 100 cycles*
- ***RAM***: *GBs, 1,000 cycles*

# cachegrind

- **cachegrind is a tool for monitoring cache usage**

```
==31751==  I   refs:      27,742,716
==31751==  I1  misses:           276
==31751==  LLi misses:           275
==31751==  I1  miss rate:        0.0%
==31751==  LLi miss rate:        0.0%
==31751==
==31751==  D   refs:      15,430,290  (10,955,517 rd + 4,474,773 wr)
==31751==  D1  misses:        41,185  (    21,905 rd +    19,280 wr)
==31751==  LLd misses:        23,085  (     3,987 rd +    19,098 wr)
==31751==  D1  miss rate:        0.2% (       0.1%   +       0.4%)
==31751==  LLd miss rate:        0.1% (       0.0%   +       0.4%)
==31751==
==31751==  LL misses:         23,360  (     4,262 rd +    19,098 wr)
==31751==  LL miss rate:         0.0% (       0.0%   +       0.4%)
```

*Instructions…* (I refs through LLi miss rate)

*Data…* (D refs through LLd miss rate)

*Combined* (LL misses, LL miss rate)

***Legend:***

- **I1**: => *Instr, Level 1*
- **LLi**: => *Instr, Last Level*
- **D1**: => *Data, Level 1*
- **LLd**: => *Data, Last Level*

## cachegrind is…

- **A simulator!**
  - *it runs your program and monitors cache behavior*
  - *defaults to cache configuration of machine where simulation is run (except on Power arch)*

- **Advantages:**
  - *Can simulate different cache configurations*
  - *Can merge different execution runs to provide an average cache usage*
  - *Avoids need for hardware-specific tools*
- **Disadvantages:**
  - *Not 100% accurate --- your mileage may vary*
  - *Slow for large program runs*

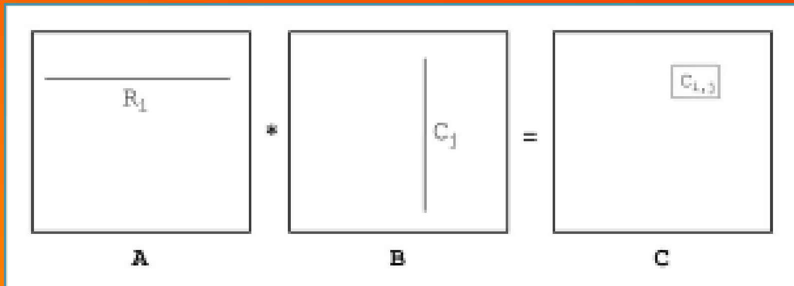
# Using cachegrind

- **Part of the valgrind tool suite:**
  - *http://valgrind.org/*
  - *Installed as part of valgrind*

- **Usage:**
  1. *Compile program with optimization*
  2. *Run valgrind with --tool option*

```
#
# makefile
#

build:
   gcc main.c –fopenmp –O4 –o myapp

simulate:
   valgrind --tool=cachegrind  myapp
```

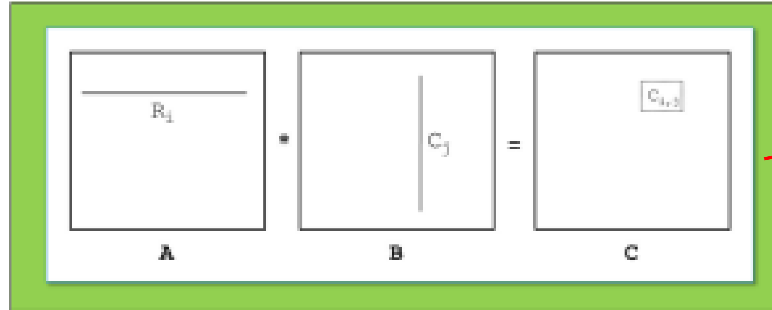
# Demo



**Matrix multiplication revisited…**

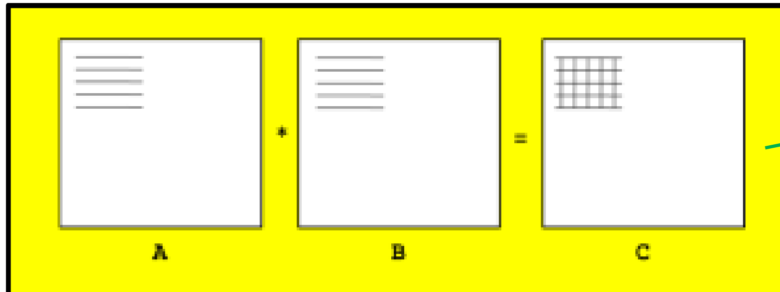
# Results

- **Naïve MM:**




30% miss rate…

- **Blocking MM:**




< 1% miss rate!

# Other features…

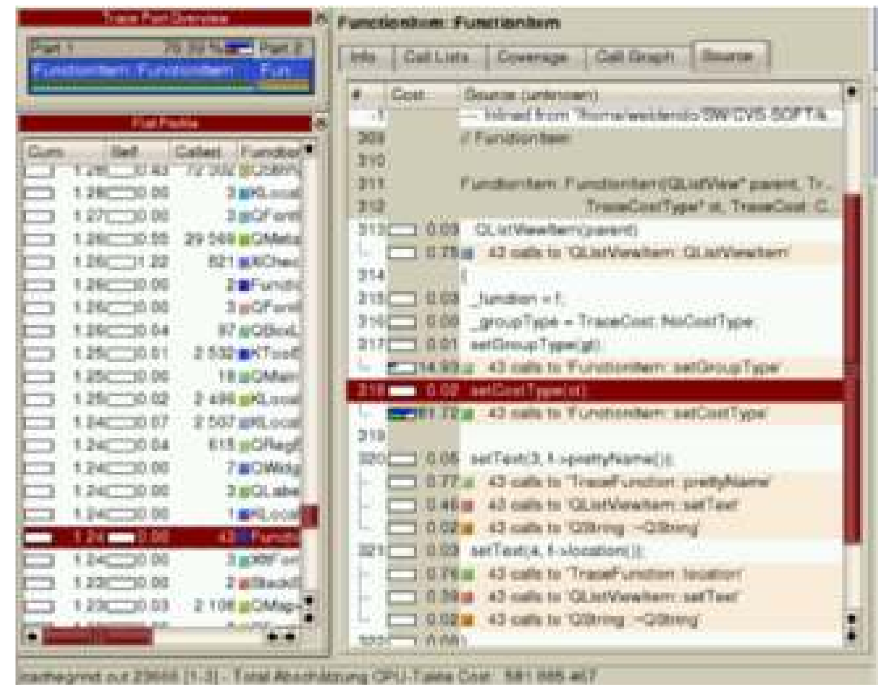
- **Run cg_annotate to get more detailed information**
  - *Cache configuration used*
  - *Function-by-function statistics*

- **Run cg_merge to merge different simulation results**

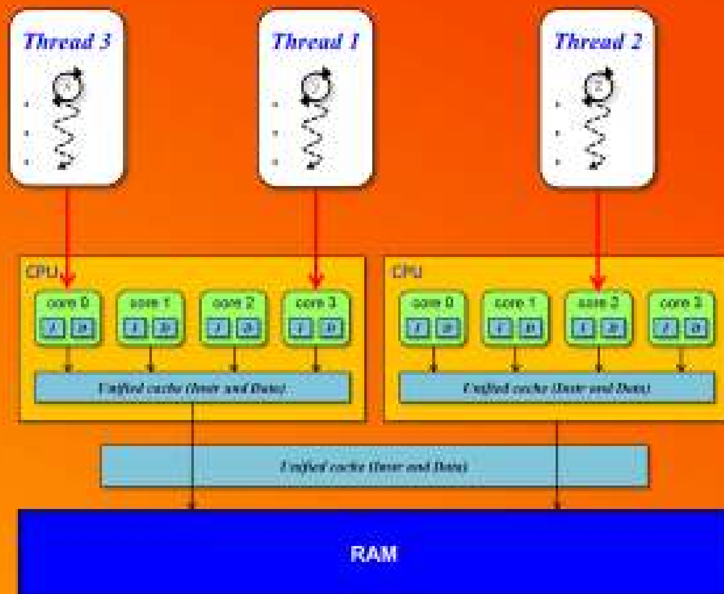
- **Online manual:**
  - *http://valgrind.org/docs/manual/cg-manual.html*

# kcachegrind

- **kcachegrind is a utility for viewing cachegrind results**
  - [*http://kcachegrind.sourceforge.net/html/Home.html*](http://kcachegrind.sourceforge.net/html/Home.html)
  - *Recompile with debugging…*



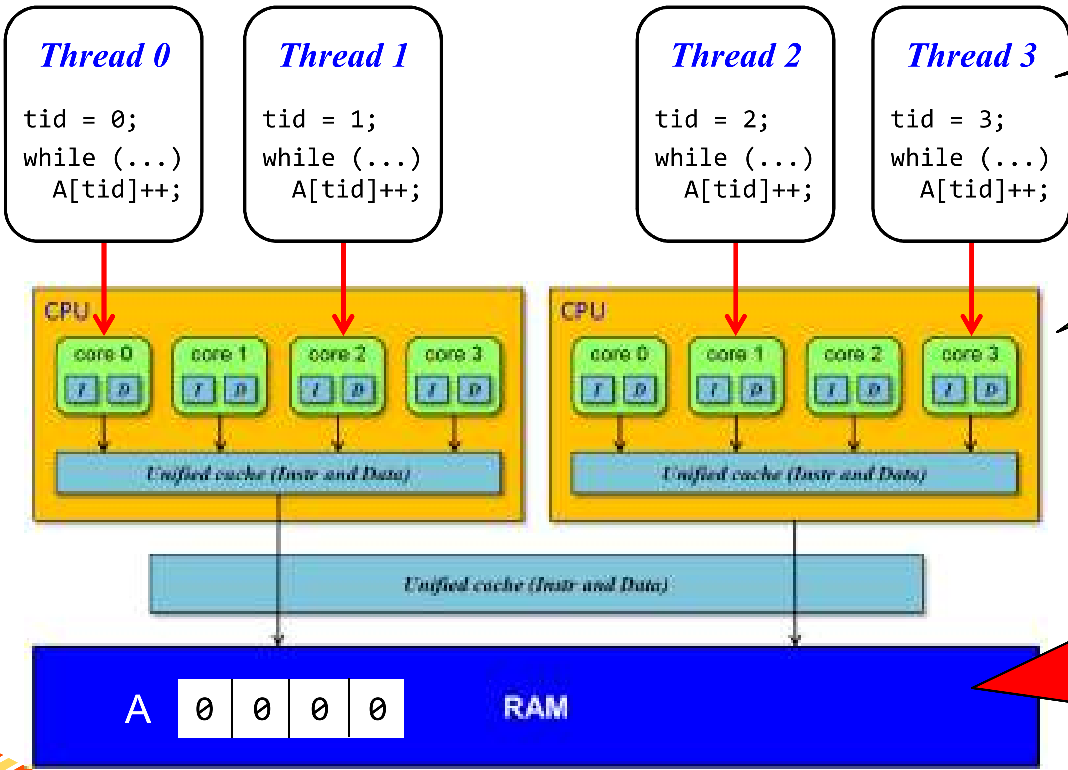
# Other implications…



**Here are 3 of the most important implications for software developers…**

# (1) False sharing

- Not a correctness problem but a performance one…




*Threads are counting results in parallel… For correctness, each thread operates on a different element of the array…*

*Each thread is going to cache its element of the array… But recall, how wide is cache entry? And what happens on a write?*

***Width of cache entry?*** *64 BYTES.*
***What happens on write?*** *Other entries invalidated.*

*All 4 threads cache the SAME 64-byte chunk of the array: A[0]-A[15] assuming 4-byte ints. So as each thread writes, it invalidates others, forcing reload and causing 1,000x penalty.* ***Known as "false sharing".***

# Solution?

- **Padding**
  - *Add unused space between elements --- width of cache line (64 bytes!)*

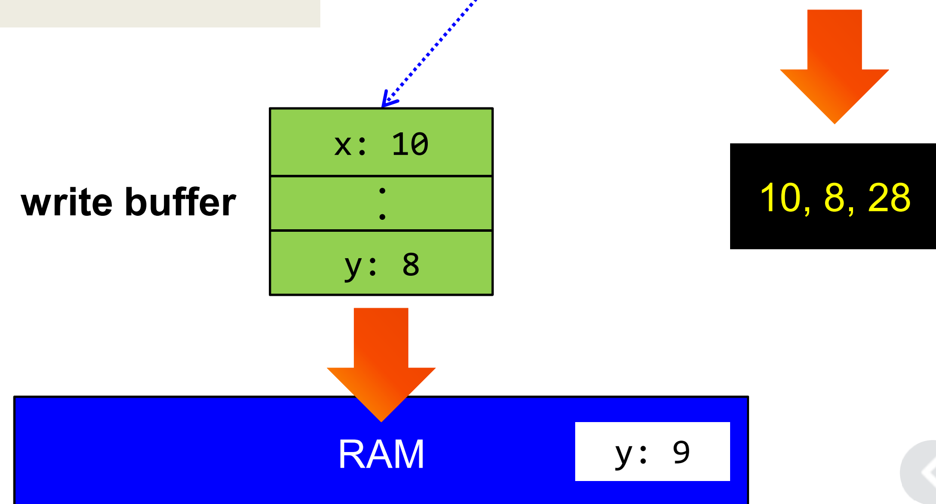
- **Redesign**
  - *e.g. "block" the data so threads work on blocks, not neighboring elements*

# (2) Sequential consistency

- **Multicore HW typically reorders writes for efficiency**
  - *E.g. write-back policy of buffering may cause out-of-order RAM updates*
- **Allowed as long as HW maintains "sequential consistency"**

> *Def: **sequential consistency** is a memory model where the results are indistinguishable from that of sequential execution on a single core.*

```
y = 9;
c = 1;
  .
  .
  .
x = y + c;
y = y - 1;
z = 2*x + y;

printf("%d,%d,%d",
         x, y, z);
```



# Is there a problem?

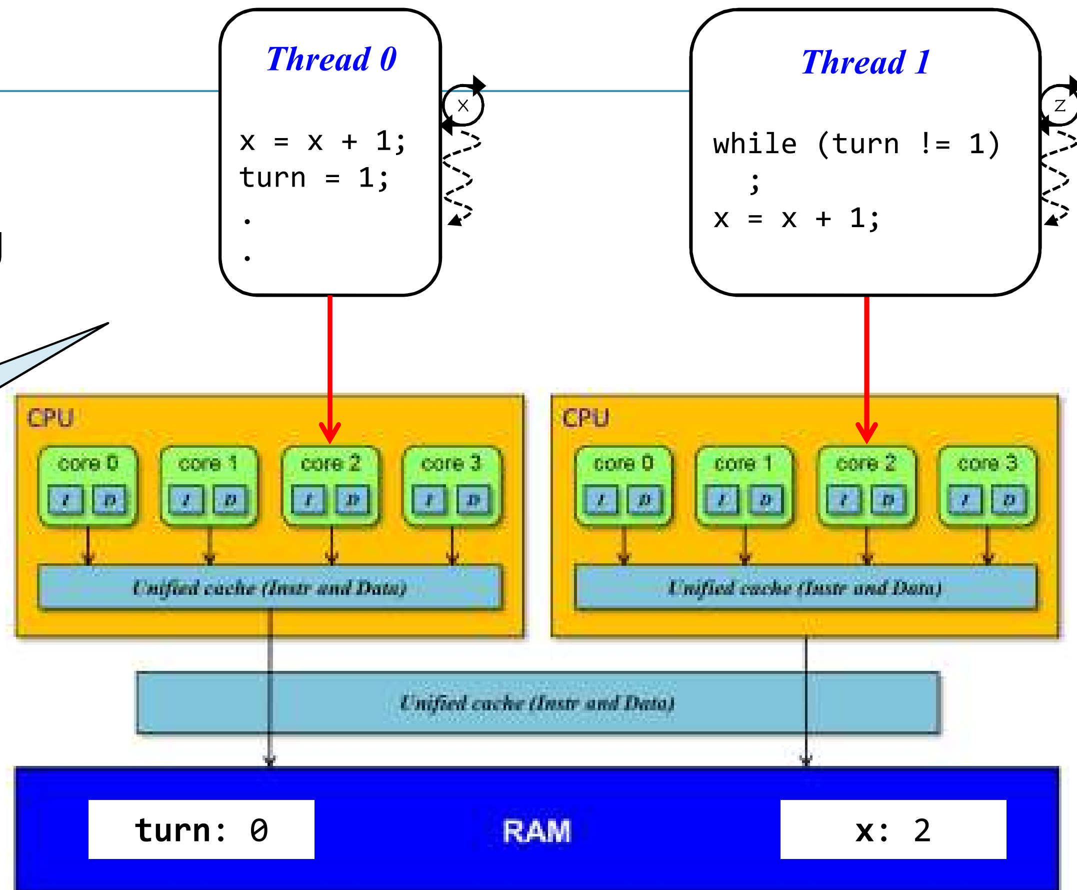
- **Potentially!**
- **You have to be careful when accessing memory from <u>other</u> cores…**

*Suppose **turn** is 0 and **x** is 2. Thread 0 increments x, and then changes turn so thread 1 can go. Thread 1 spins until its turn, and then increments x. After both threads finish, x should be 4.*

*This code is correct on a single core CPU.*

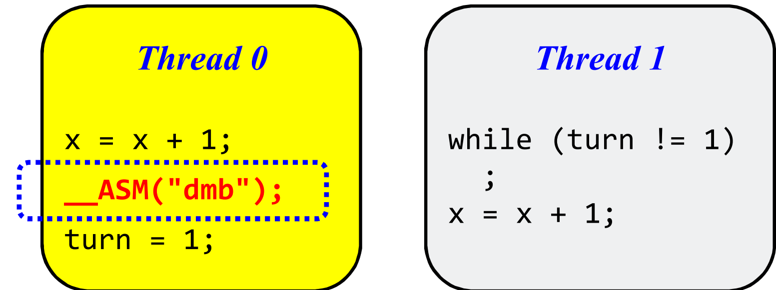
***This code is *incorrect* on a multicore CPU.***

*Thread 0 increments x and then updates turn, but turn could be written <u>before</u> x. So thread 1 could exit the loop, and fetch the old value of x. The result will be 3 when both threads finish, not 4.*



# Solution?

- **Use "memory fence" instruction**
  - *Forces write to memory*
  - *Example: "data memory barrier" instr on ARM*



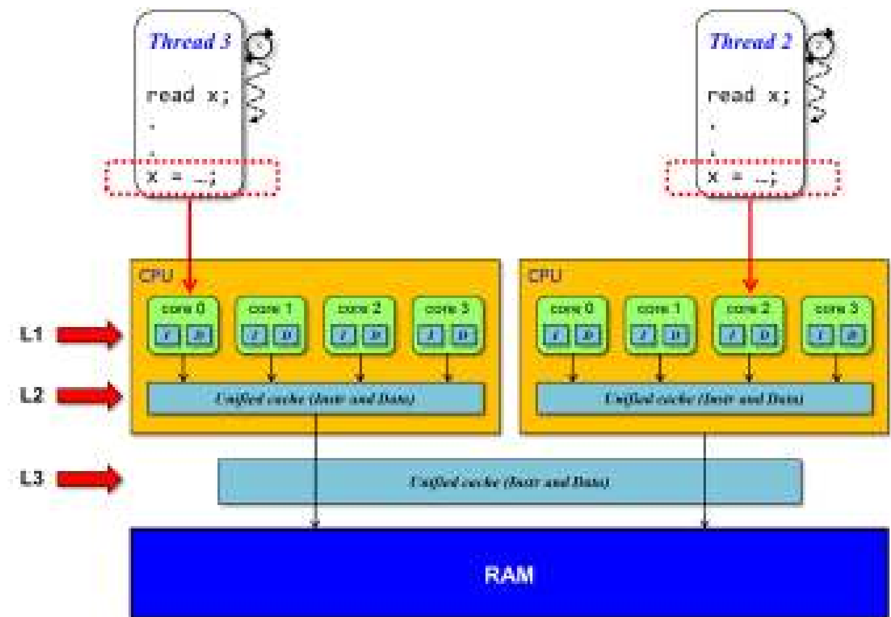

- **Use synchronization construct**
  - *Condition variable or semaphore*

# (3) Race conditions

- **What happens if 2 cores read and write the same variable?**

- **Much like 2 threads on a single-core:**
  - *"race condition"*
  - *no guarantee which will write first*

∴ *Values read & written are unpredictable.*
*This is almost always a **logic error**.*



# Solution?

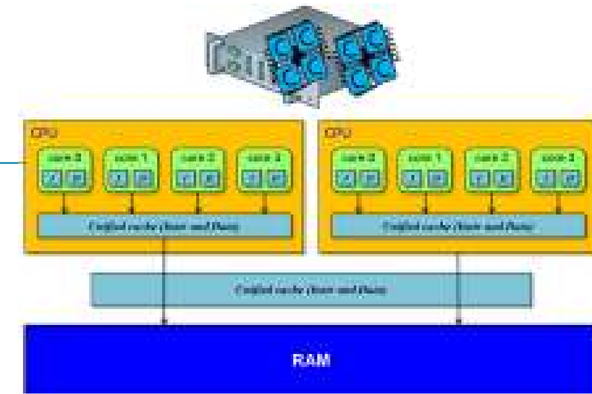
- **Locking**
  - *mutex around critical section*
- **Redesign**
  - *to eliminate shared resource*
  - *e.g. shared sum variable? Use reduction pattern…*

# Coherence vs. Consistency

- **Coherence:**
  - *Guarantees that (1) a write will eventually be seen by other cores, and (2) all cores see writes to the same location in the same order.*

- **Consistency:**
  - *Defines the ordering of writes and reads to different memory locations*
  - *Hardware (and programming languages) guarantee a particular consistency model, e.g. "sequential consistency"*

# Summary




# Summary

- **Multicore is the path forward for better performance**
- **But good performance is not easy:**

$$HPC == \boxed{Parallelism} + \boxed{Memory\ Hierarchy} - \boxed{Contention}$$

- **Thank you for attending!**
  - *Email:* *joe@joehummel.net*
  - *Materials:* *http://www.joehummel.net/freescale.html*

www.Freescale.com

© 2014 Freescale Semiconductor, Inc. | ***External Use***