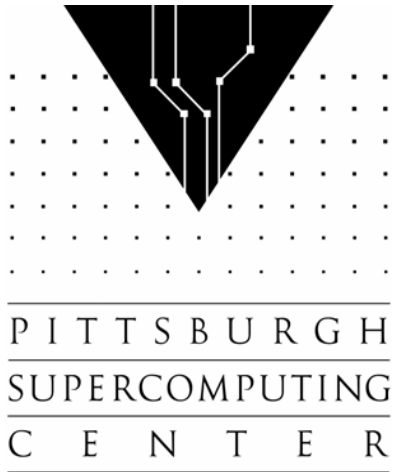


# Memory Issues and Strategies



Joel Welling

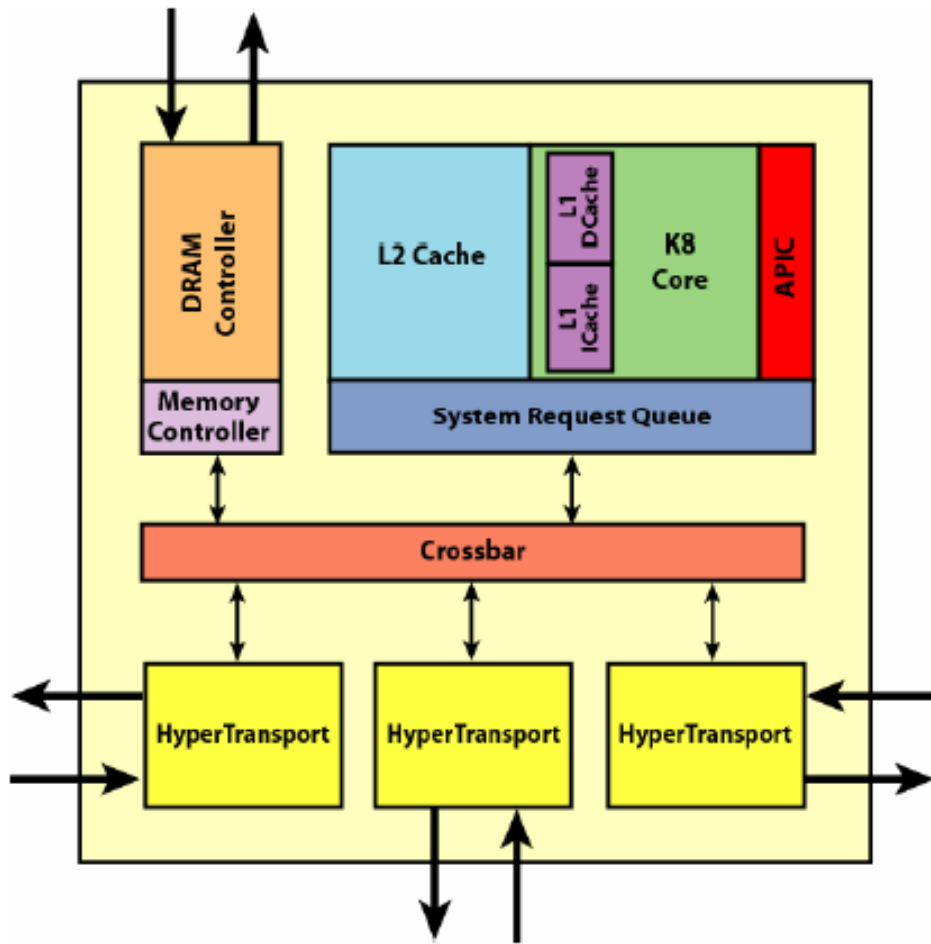
# Talk Outline

- 1) Cache architecture changes as core count rises
- 2) Benchmark examples of bandwidth contention
- 3) Bandwidth-limited rate for specific code
- 4) Techniques to minimize problems

# Core $\leftrightarrow$ L1 $\leftrightarrow$ L2 $\leftrightarrow$ DRAM

- Computing is done in the core, but data starts life in DRAM
- Data must move to core, passing through multiple layers of cache.
- Let's consider what that implies.

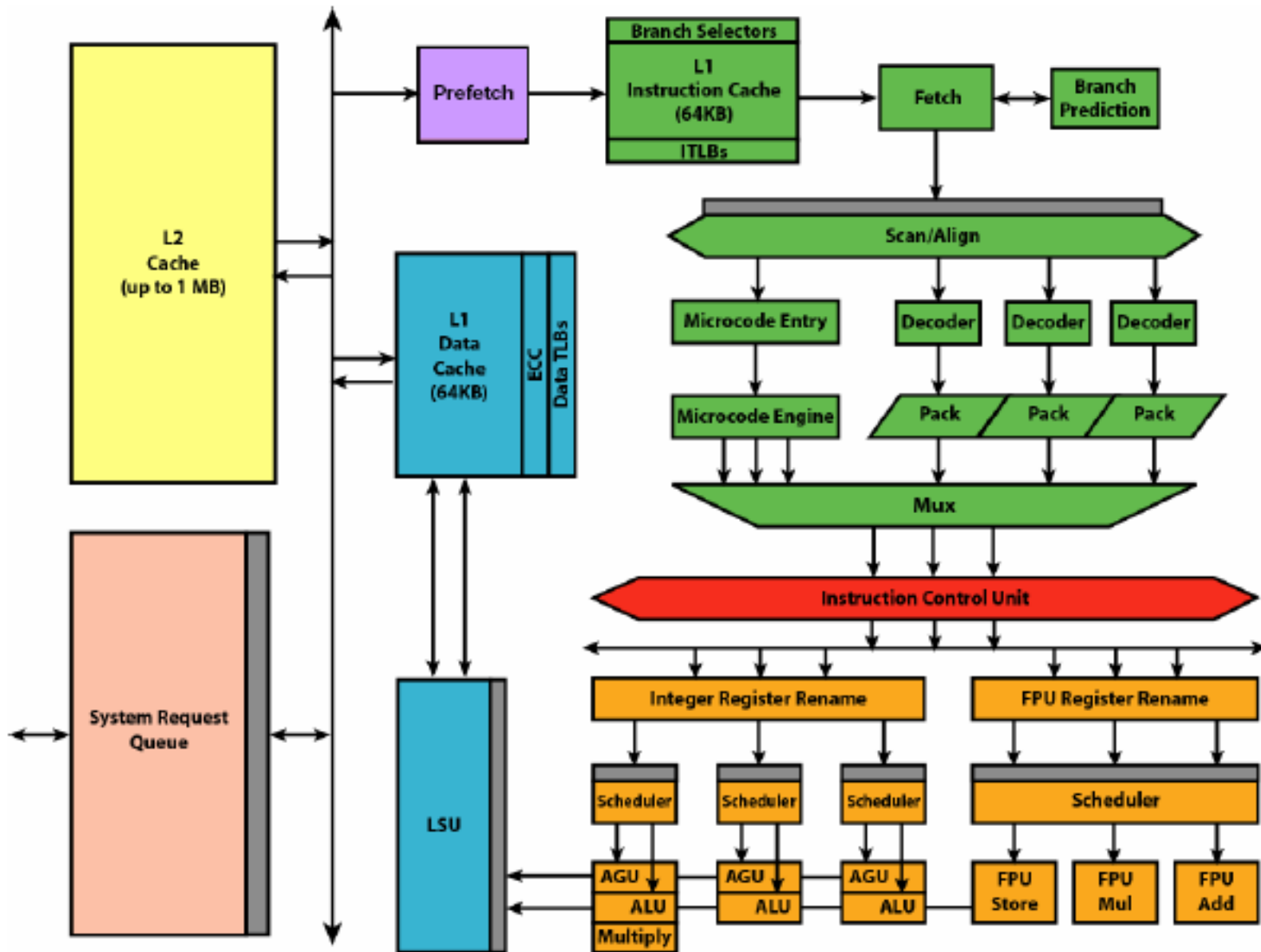
# Example: the AMD Opteron



- This is the AMD K8 architecture
- Most Semperons, Athlon64s, Opterons
- Original processor type on our XT3

# Architecture of the core

Pittsbu



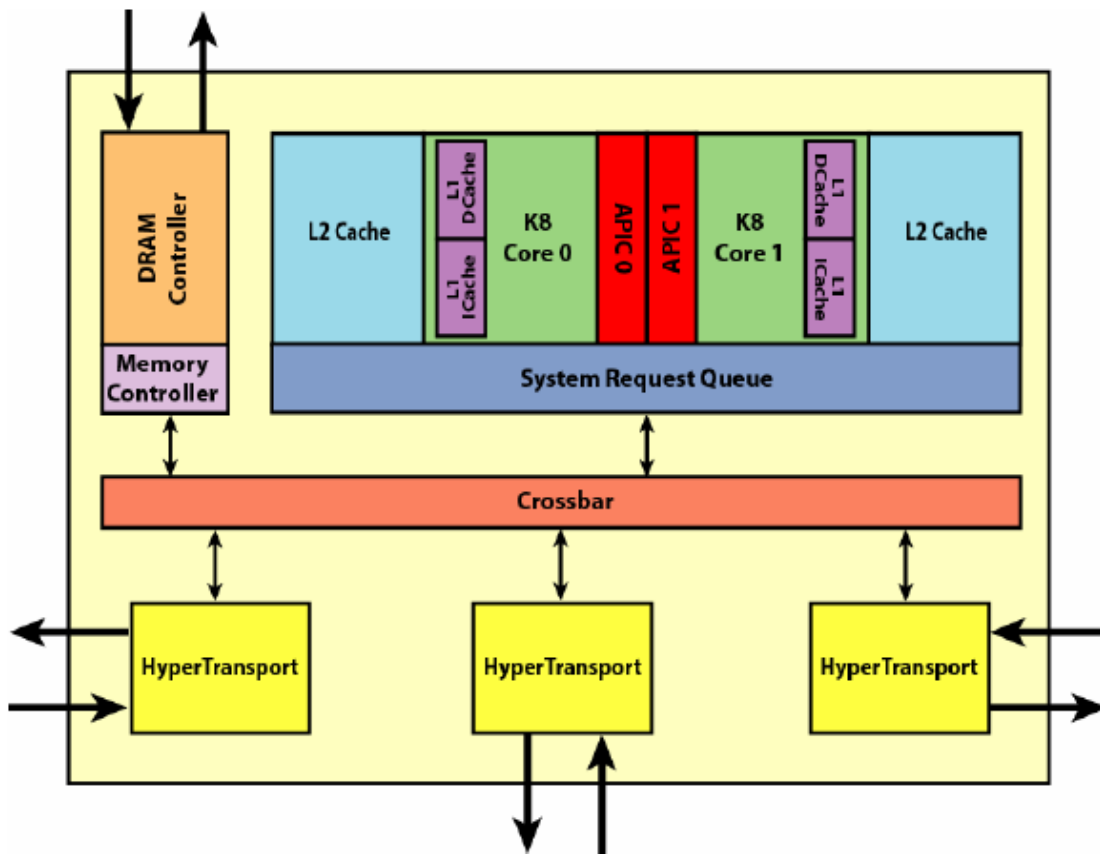
Copyright Mindshare, Inc. 2004



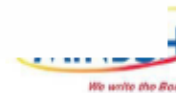
# Cores run multiple threads, but...

- Cores these days have multiple register sets and can run 2 or more 'hyperthreads'
- But supercomputing OS's don't usually let you access them.
- We'll treat each core as running one thread.

# Dual core Opteron

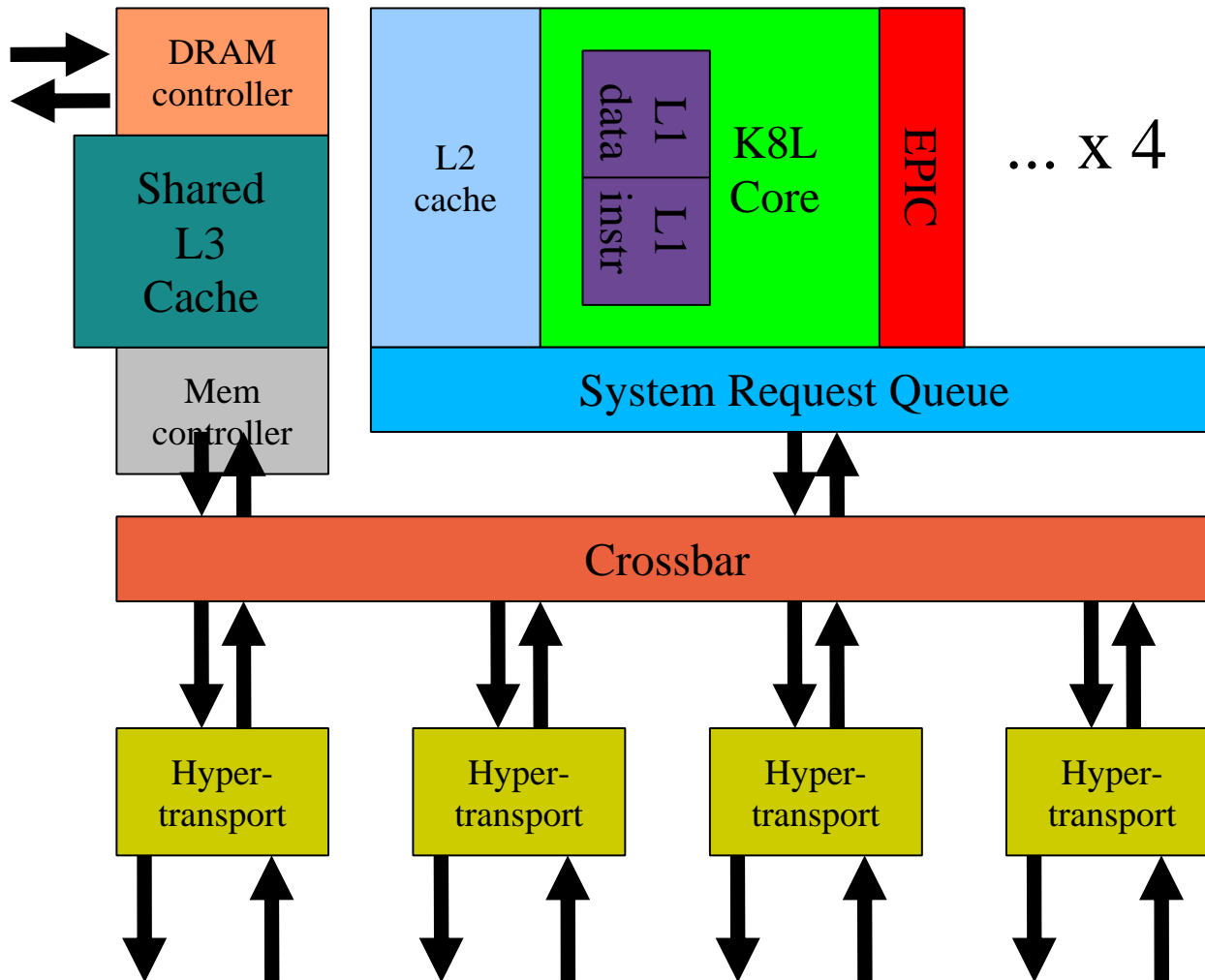


Copyright Mindshare, Inc. 2004



- Each core has its own L1 and L2 cache
- But they share memory controller- a bottleneck we've seen in timings

# Barcelona Quad Opteron



- My understanding
- Note L3 cache to reduce memory contention
- Core-to-memory data path width roughly doubles

# Quad core

- This is AMD Barcelona
- Added an L3 cache.
- DRAM interface still shared.
- Implies DRAM bandwidth is reduced as number of cores increases.

# Barcelona cache architecture

## Balanced, Highly Efficient Cache Structure

### Dedicated L1

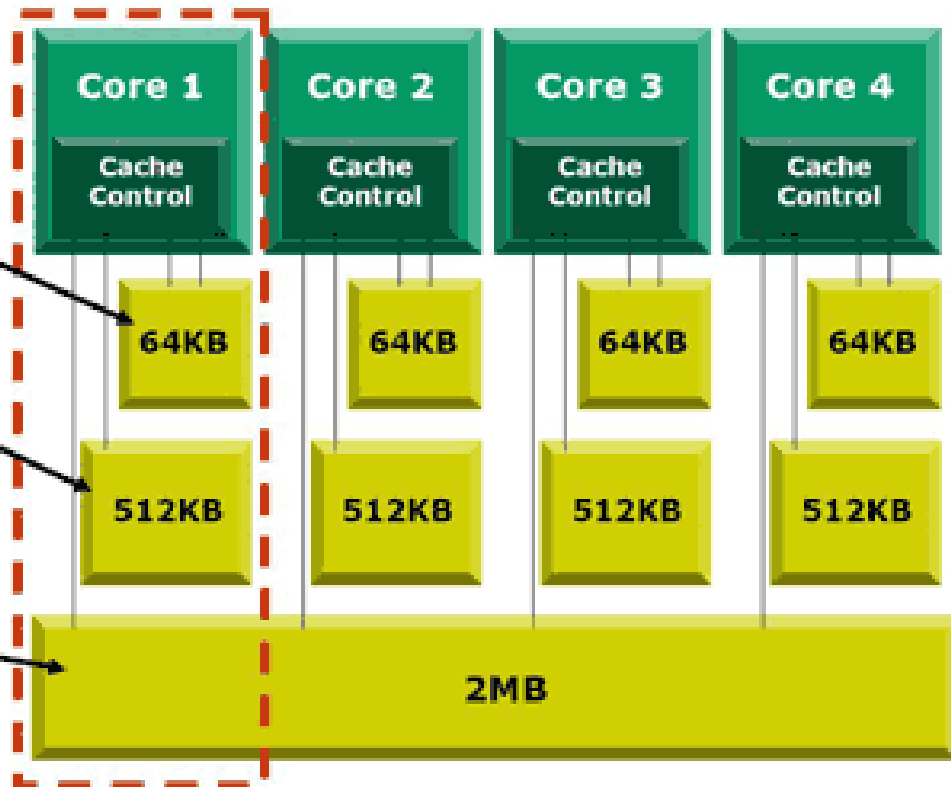
- Locality keeps most critical data in the L1 cache
- Lowest latency
- 2 loads per cycle

### Dedicated L2

- Sized to accommodate the majority of working sets today
- Dedicated to eliminate conflicts common in shared caches
  - Better for Virtualization

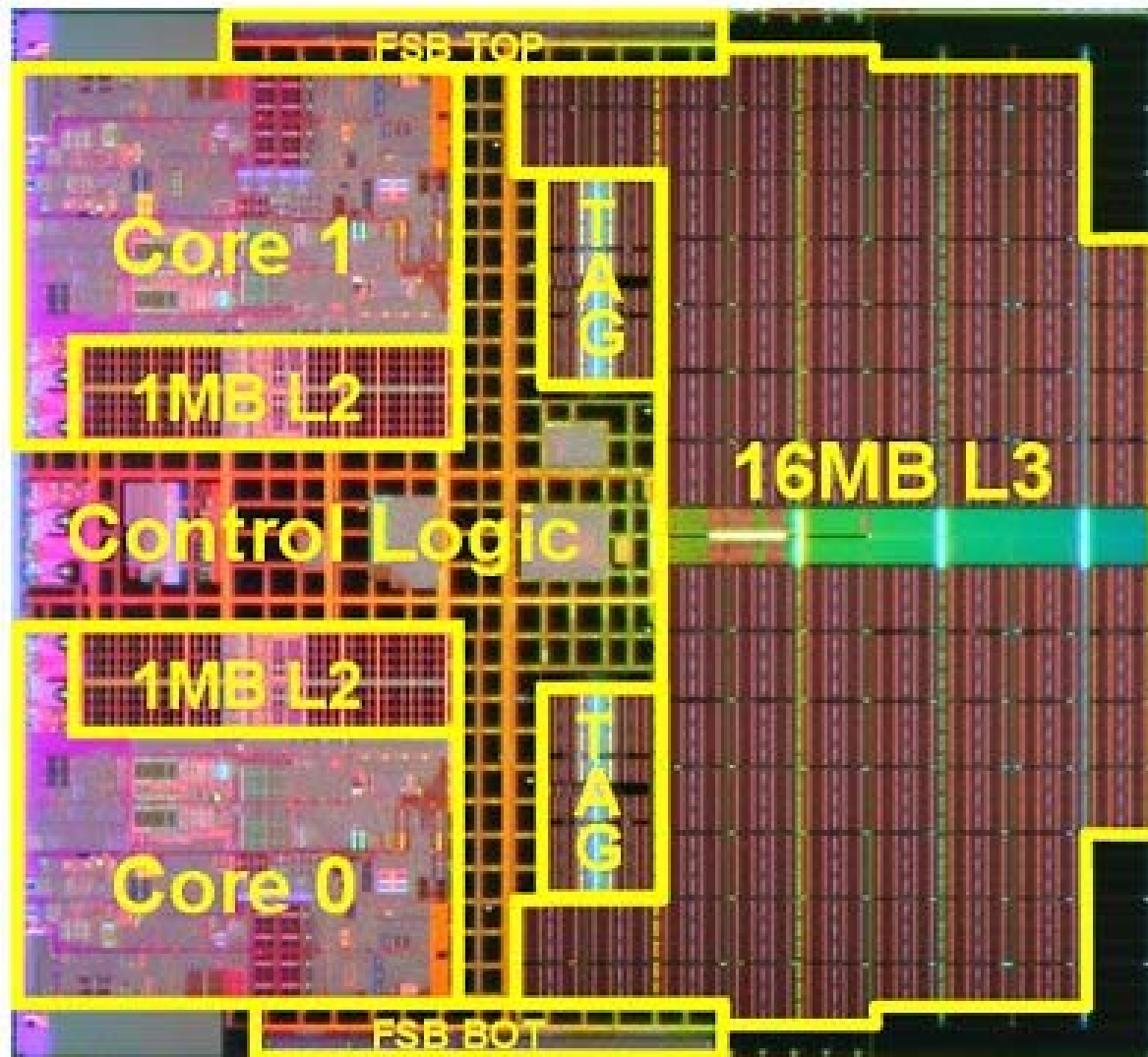
### Shared L3 - NEW

- Victim-cache architecture maximizes efficiency of cache hierarchy
- Fills from L3 leave likely shared lines in the L3
- Sharing-aware replacement policy
- Ready for expansion at the right time for customers



# Intel Xeon 'Tulsa' Dual Core

Pittsburgh Supercomputing Center



- 2 cores, each with L1 and L2
- Shared 16MB L3
  - (It's nice to have a 65nm fab!)

# Single vs. Dual Timings

- Compare an N-node dual-core system with a 2N-node single core
- 2N cores in each case, but:
  - DRAM accesses doubled
  - For memory intensive code, can lose 25% or more performance
  - Communication bandwidth halved
  - Communication ‘traffic jams’ increase

# Talk Outline

- 1) Cache architecture changes as core count rises
- 2) Benchmark examples of bandwidth contention
- 3) Bandwidth-limited rate for specific code
- 4) Techniques to minimize problems

# Measurement Methods

- Believe the vendor
  - Cray XT3 poster says 5.7GB/sec sustained through memory interface for our original single-core XT3
- Believe the spec sheets
- Measure for yourself

# Spec sheets say...

- L1 data cache loads: 2x64 bits/cycle
  - That's 35.2 GB/sec at 2.2GHz
- L1 instruction cache fetch: 16 bytes/cycle
- L2/memory controller interface: 64 bits/cycle
  - That would allow 17.6 GB/sec
  - But the memory controller is slower
- Barcelona doubles all of these

# A Word About Latency

- That's a whole class in itself.
- We're going to assume data streaming, such that cache line request queues keep the interfaces busy.
- This implies very predictable data access patterns
  - Avoid jumping around
  - Minimize branches

# Measurement Tools

- Ramsmp
  - we'll use it to measure read and write independently
- HPC Streams
  - COPY operation counts bytes twice, giving higher numbers
- Memcpy, but it has to be a **good** memcpy
  - FPU is wider, faster datapath than ALU

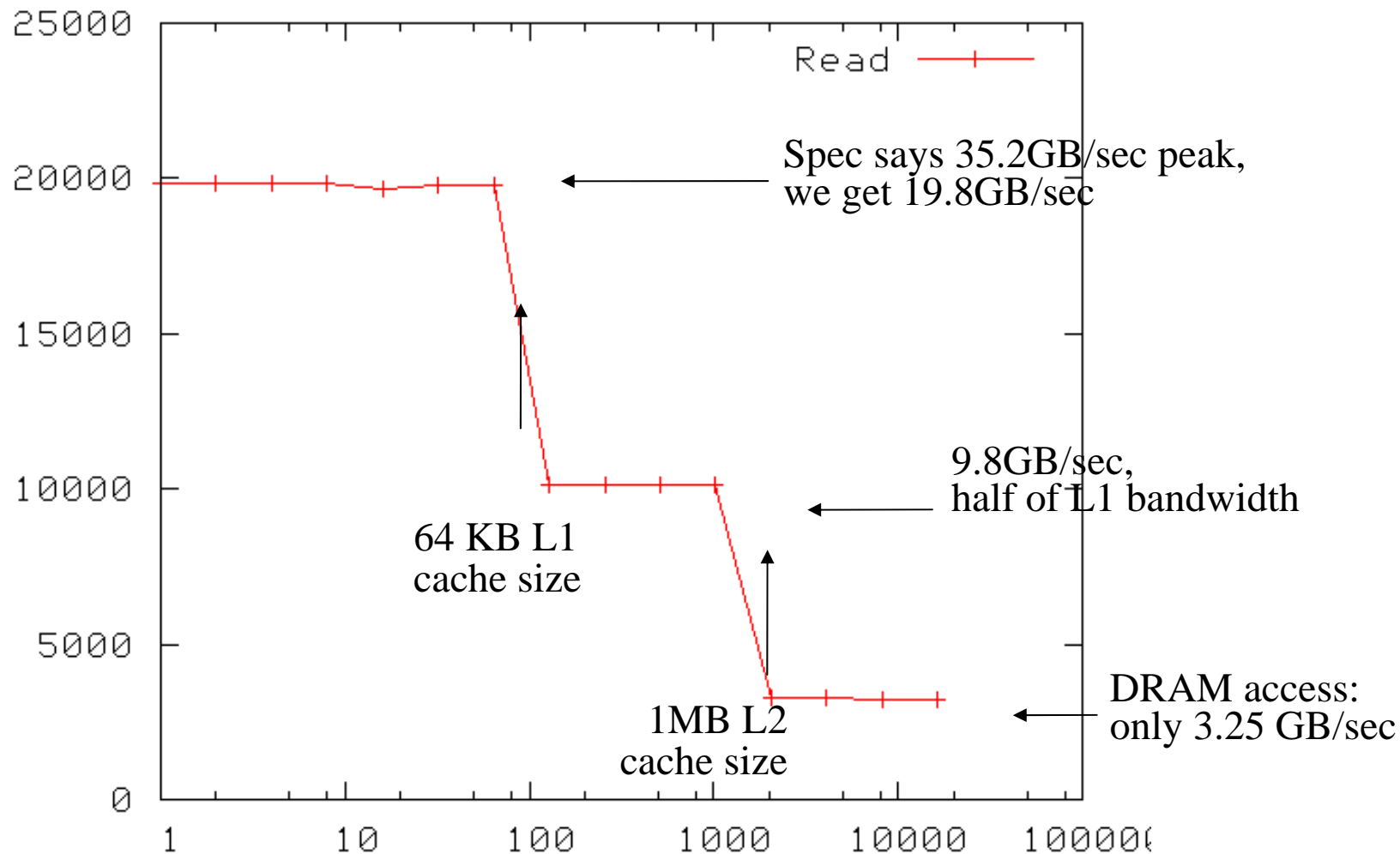
# Measurement Issues

- If you do a copy, do you count each byte twice?
  - STREAM benchmark does
- Non-cached writes (non-temporal) are faster because you can skip 'write-allocate'. (To make sure things are OK if you only write half a line).
  - Potential big win under very special circumstances

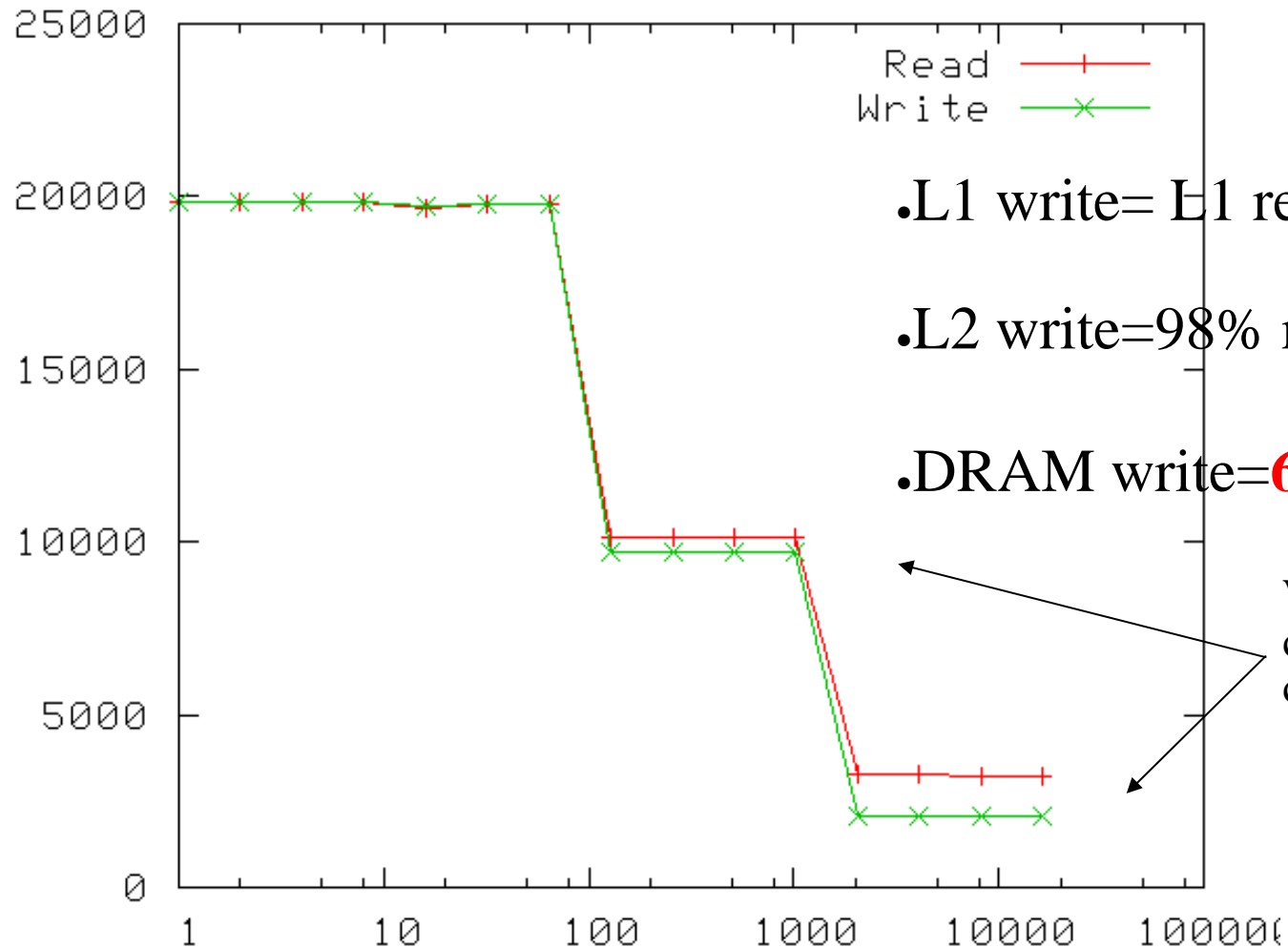
# Single- and Dual Core Performance

- This is actually one proc of our XT3
- Measured with an MPI-ified ramsmp
- Goal is to look at cache size boundaries and bandwidth bottlenecks

# Opteron Single Core Read (GB/sec vs. block size in KB)



# Single Core Read vs. Write (GB/sec vs. block size in KB)



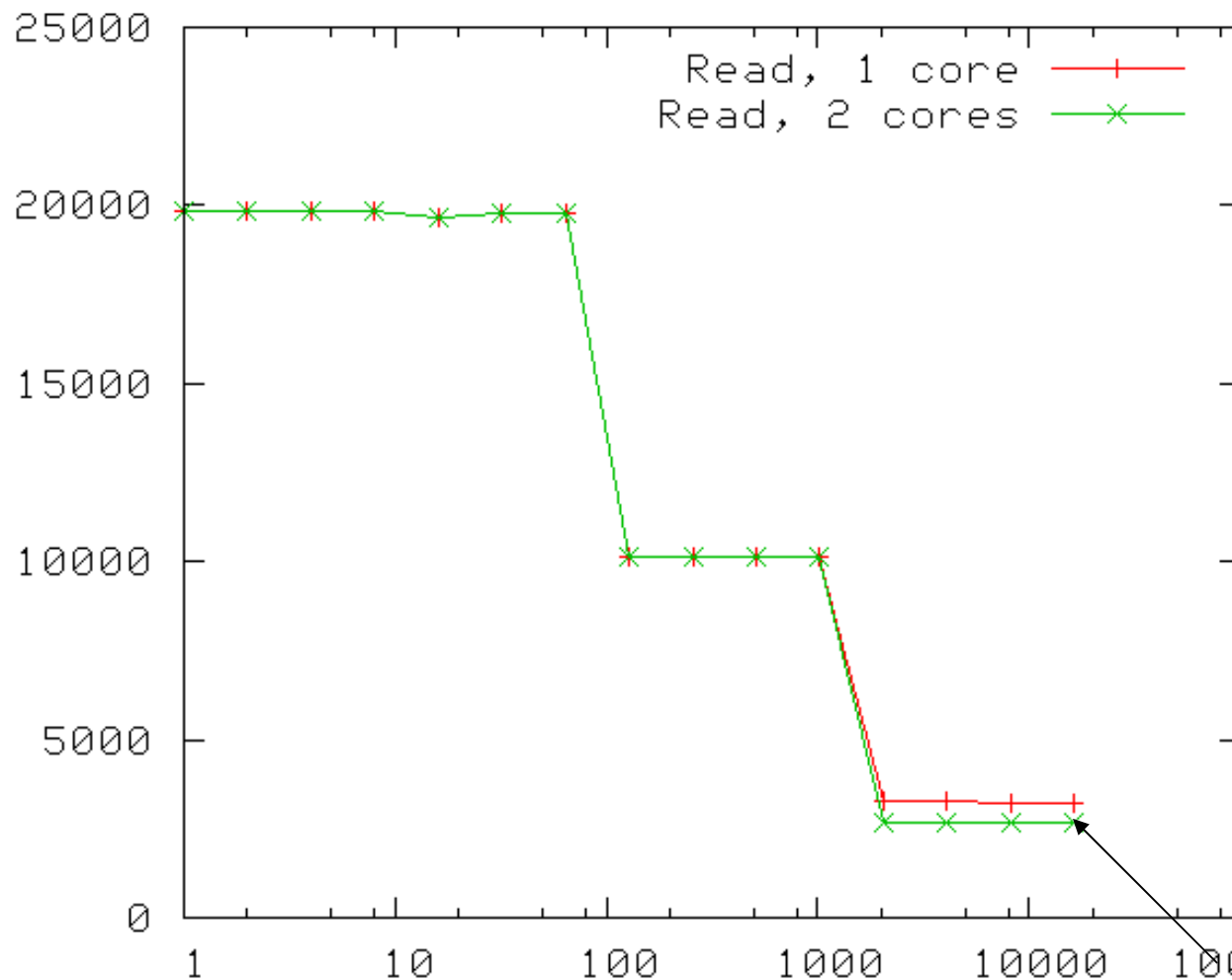
•L1 write= L1 read

•L2 write=98% read

•DRAM write=**63%** read

Writing has some cost at L2, and a big cost at DRAM

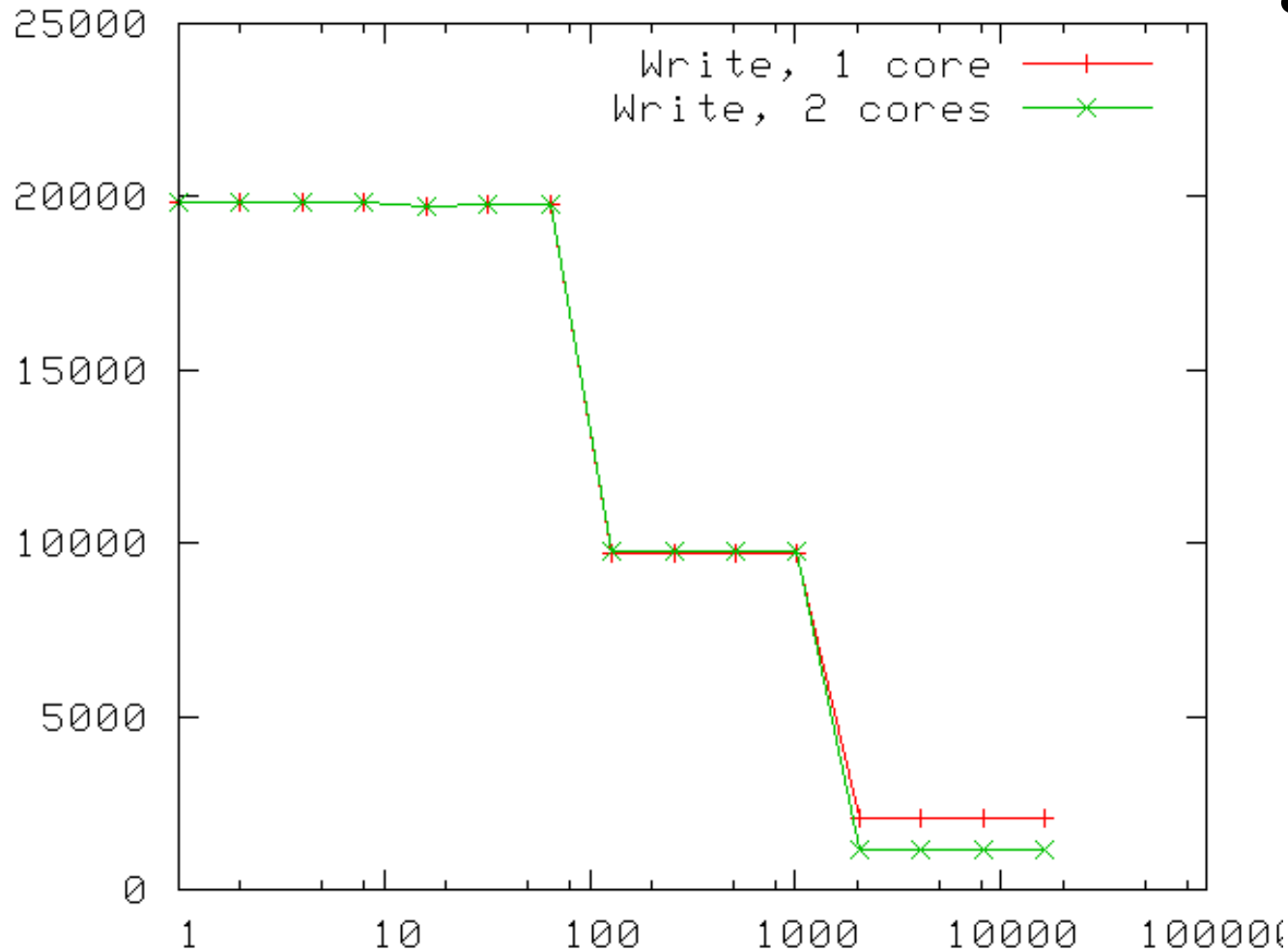
# Single vs. Dual Core Opteron



- See the DRAM interface contention?
- Note that cache pathways are independent

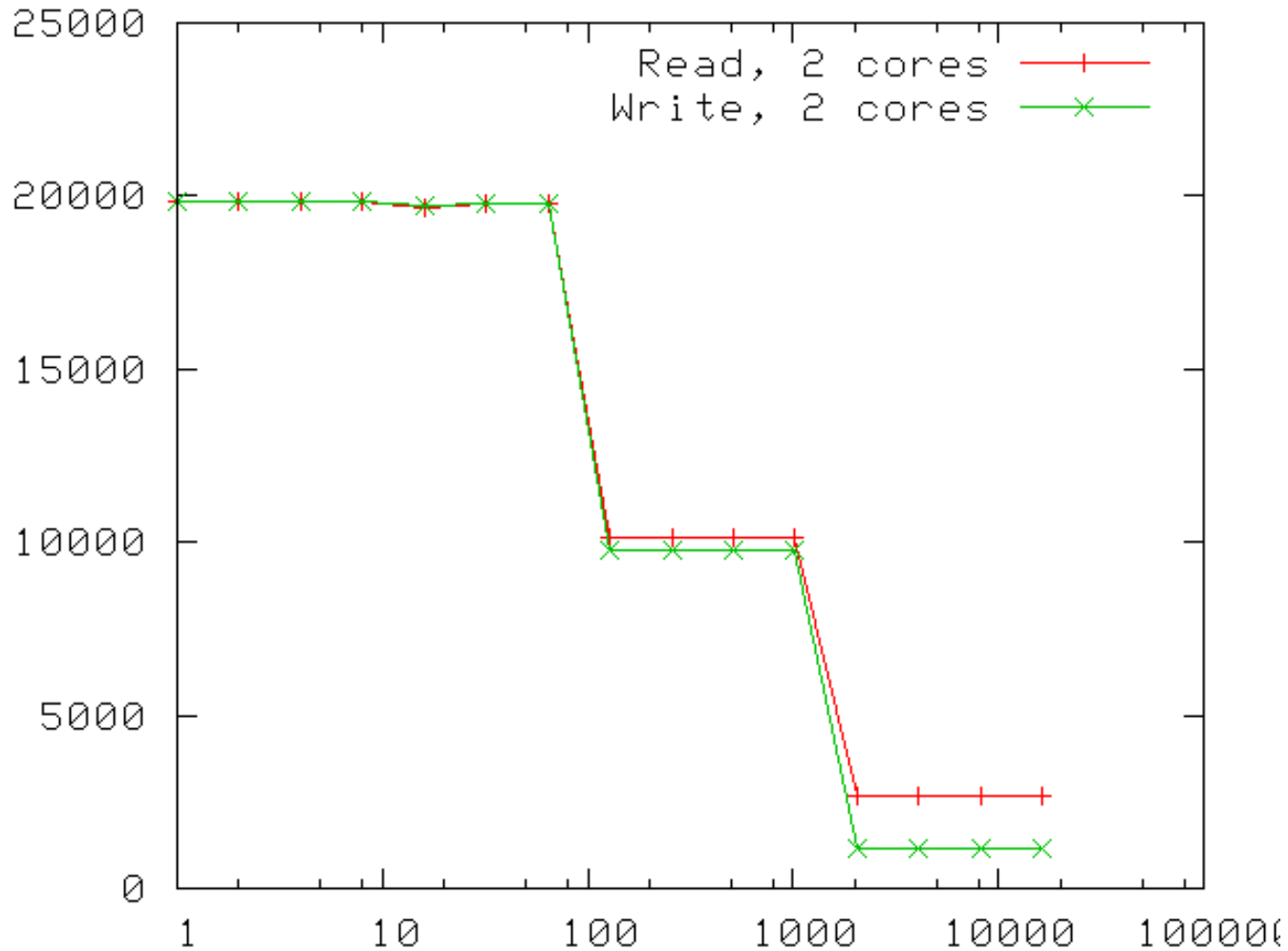
Gap size varies with test program

# Single vs. Dual Core Write



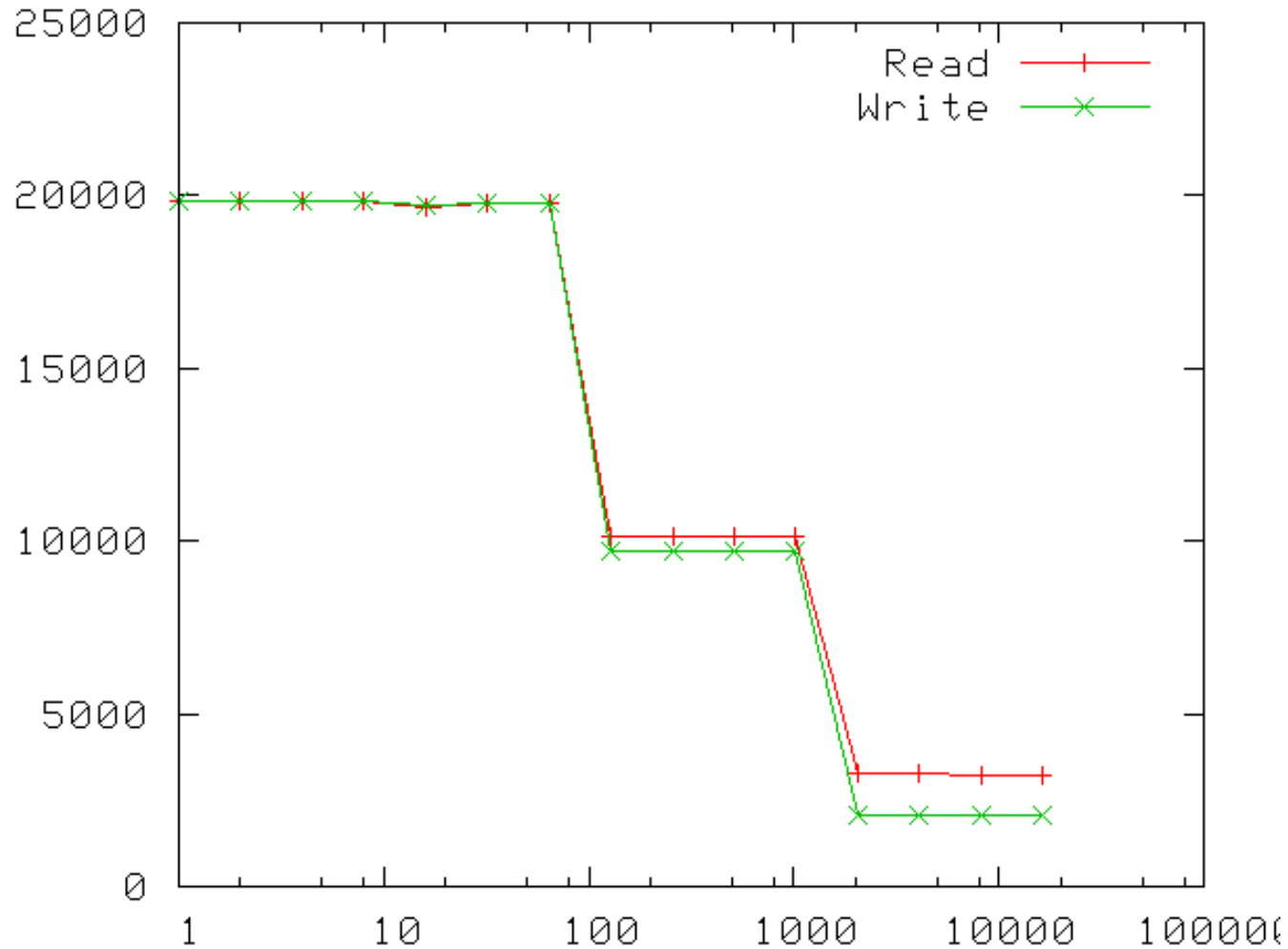
- You pay a higher penalty when writing

# Dual core Opteron read vs. write

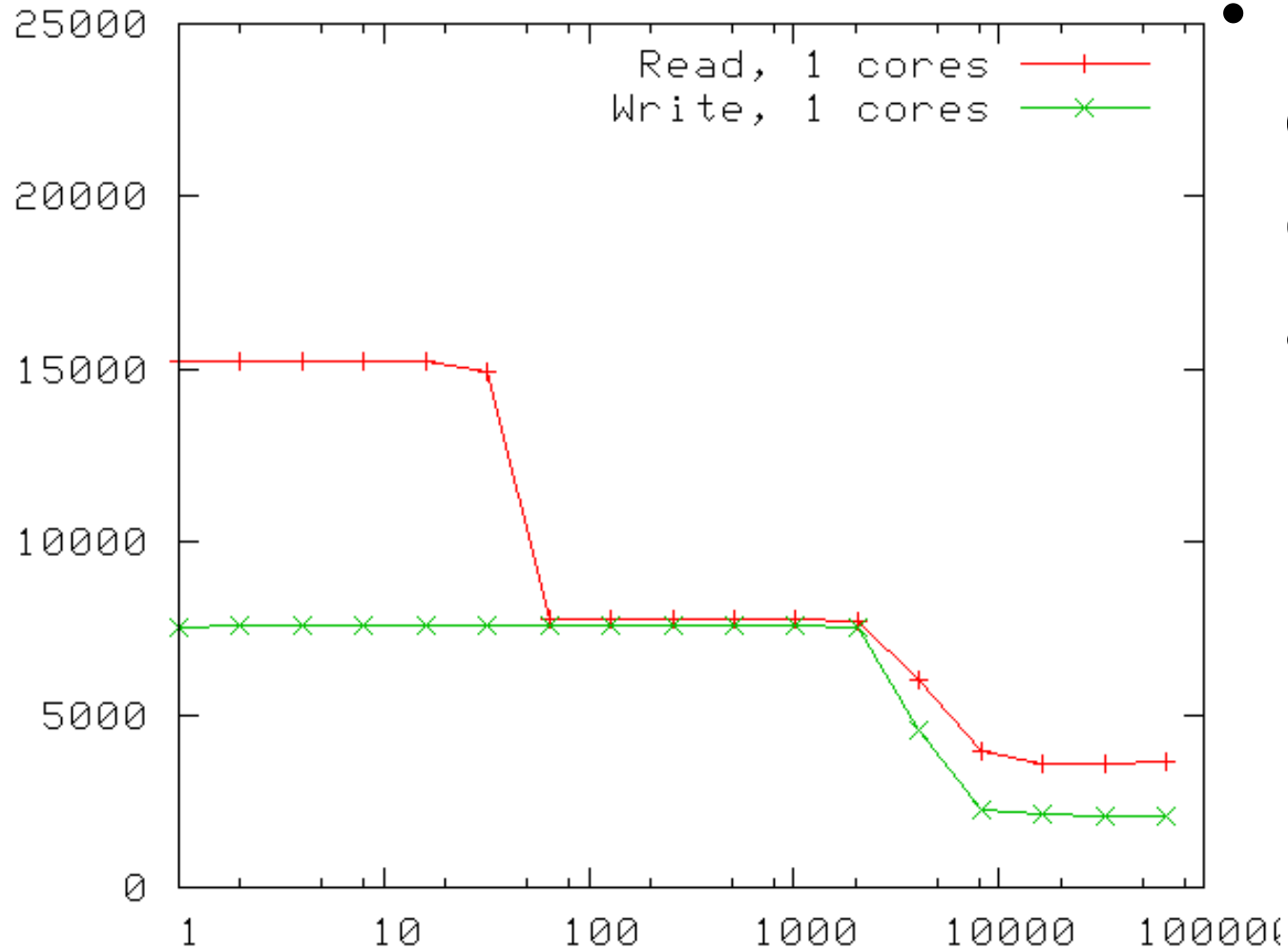


- Write bandwidth is compromised by the other core.

# Single Core Opteron Read vs. Write

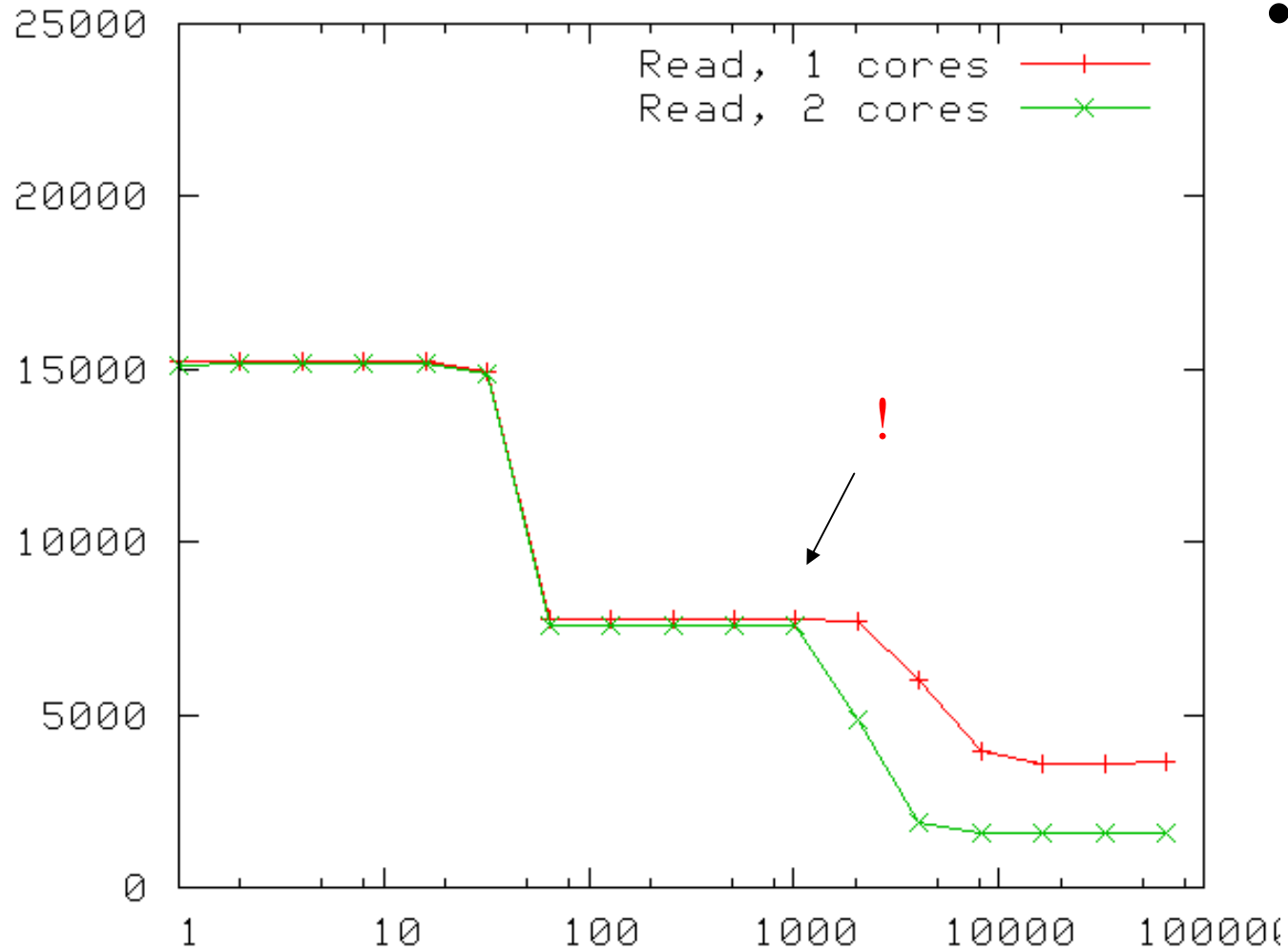


# Intel 5130 Woodcrest, 1 core



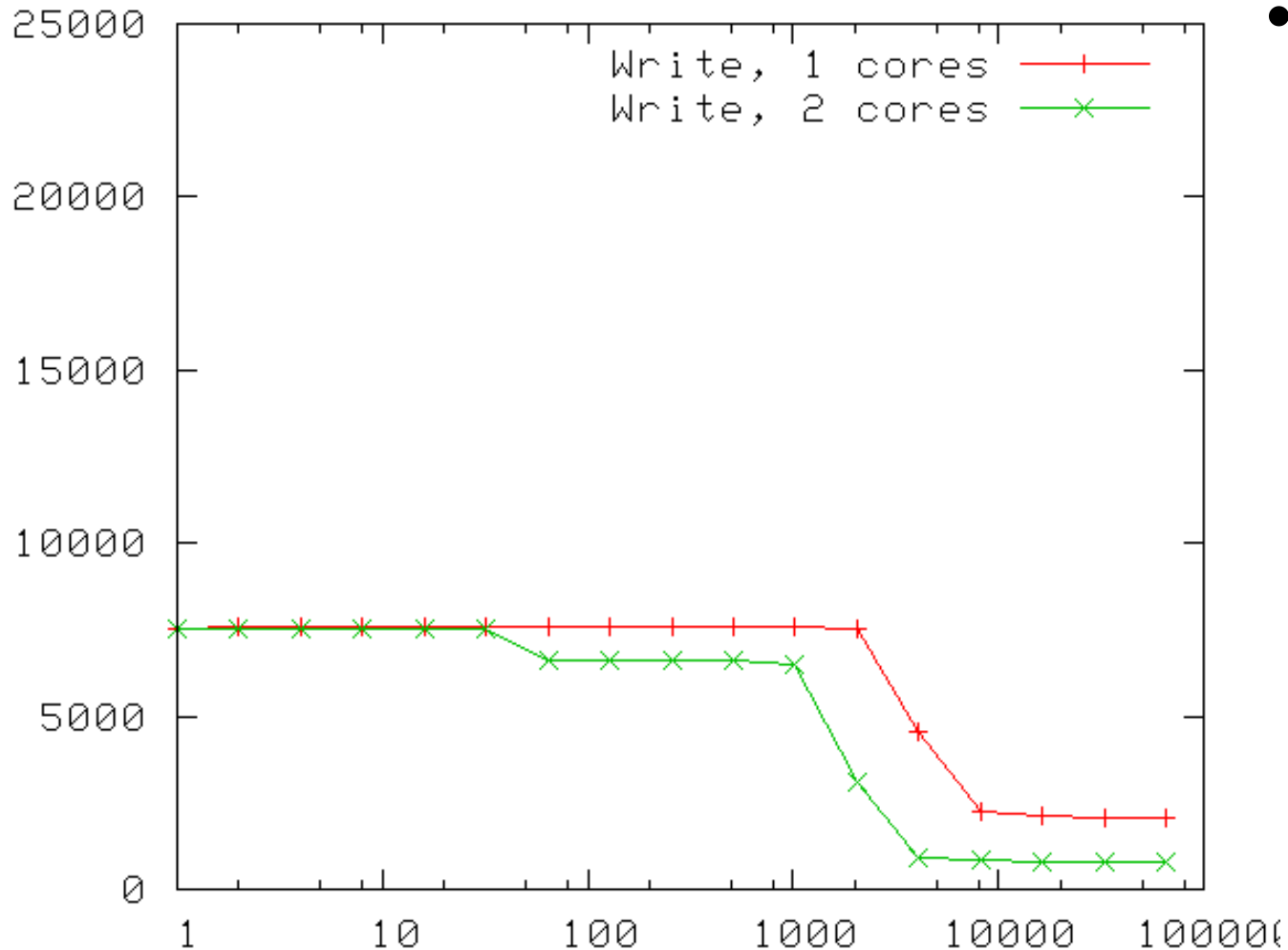
- Very different cache architecture

# Woodcrest read, 1 vs. 2 Cores



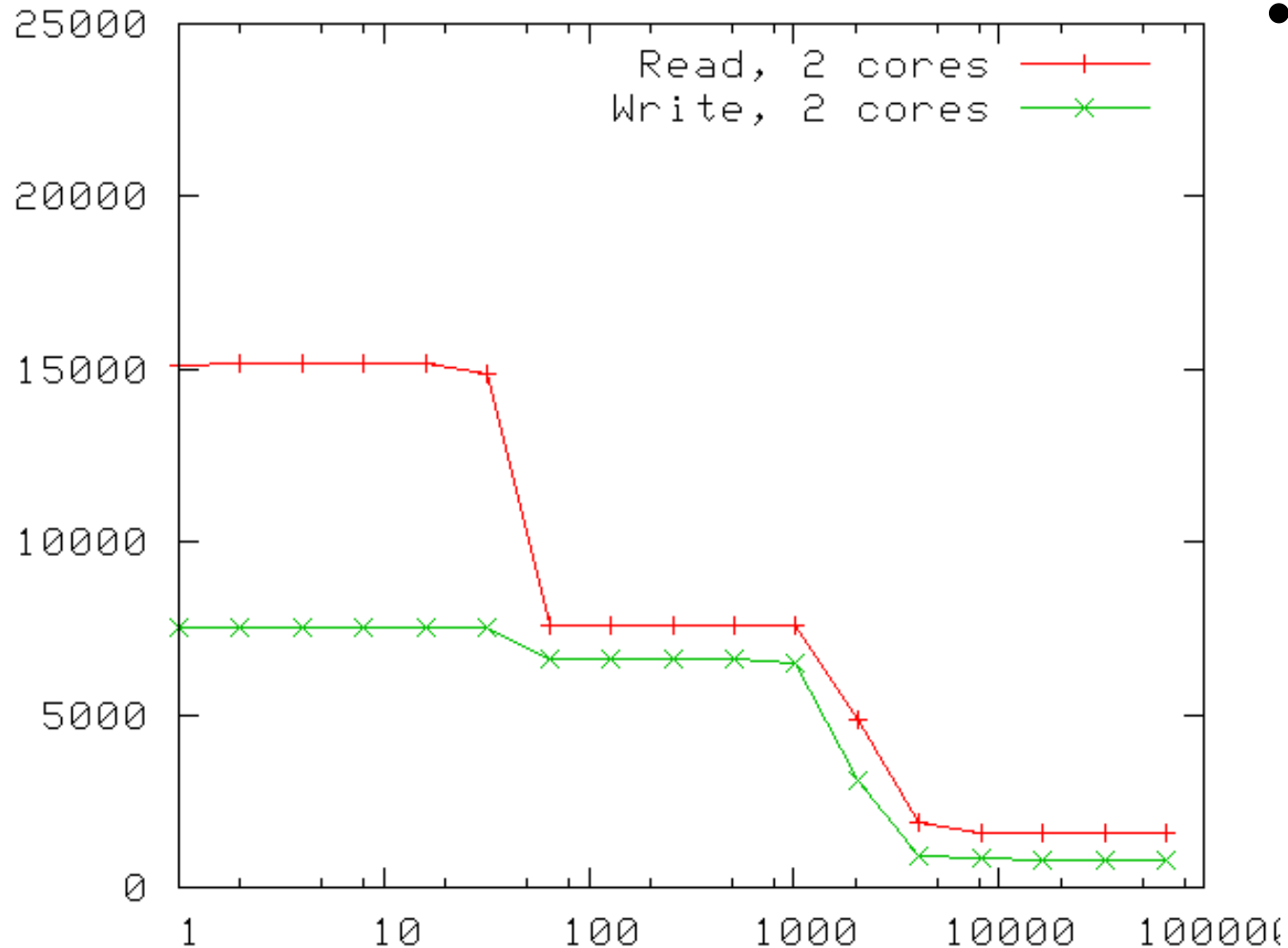
- Woodcrest has a 4MB shared L2!

# Woodcrest write 1vs 2 cores



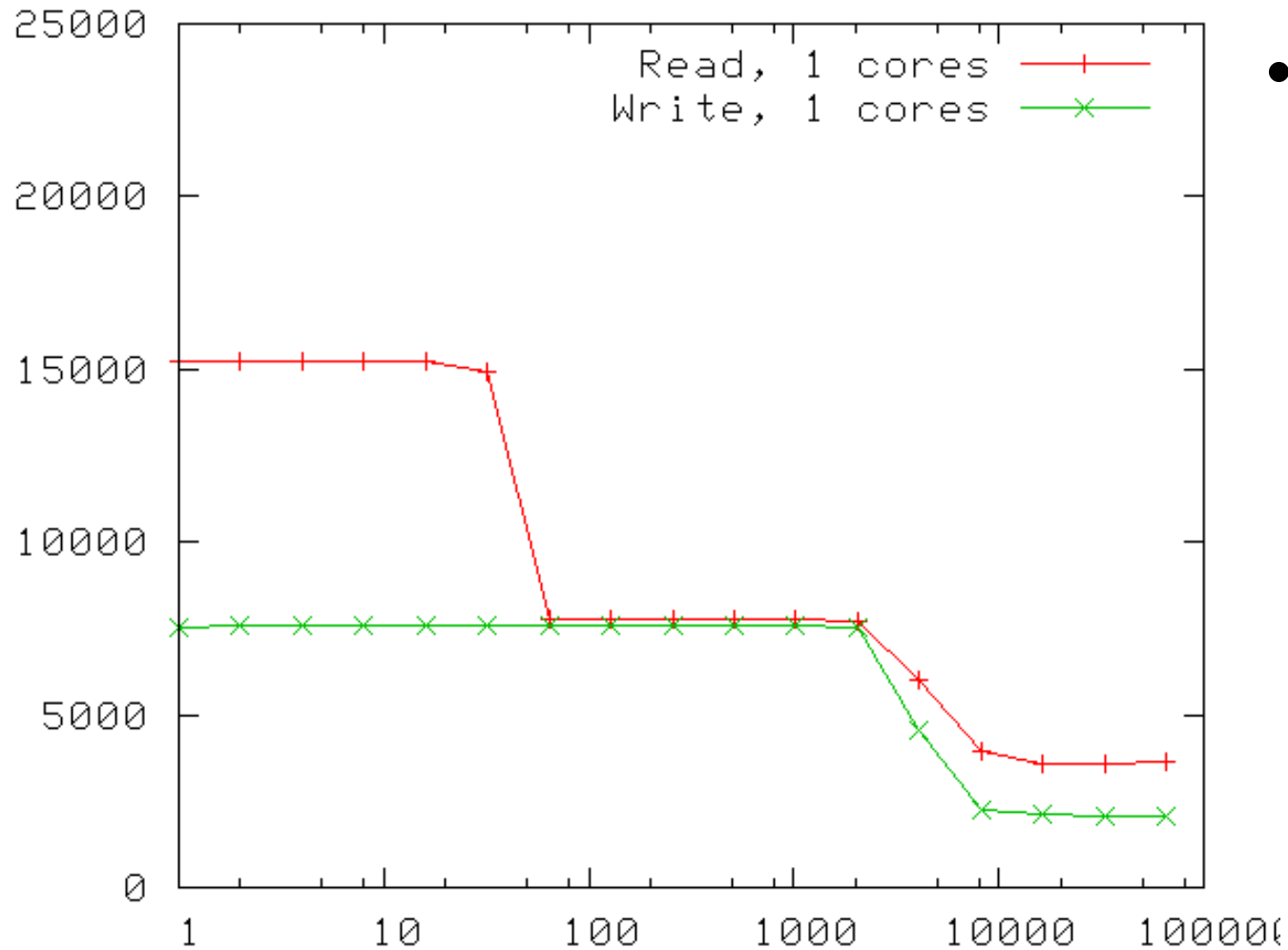
- Combines effects of slower L2, shared L2

# Woodcrest at 2 cores



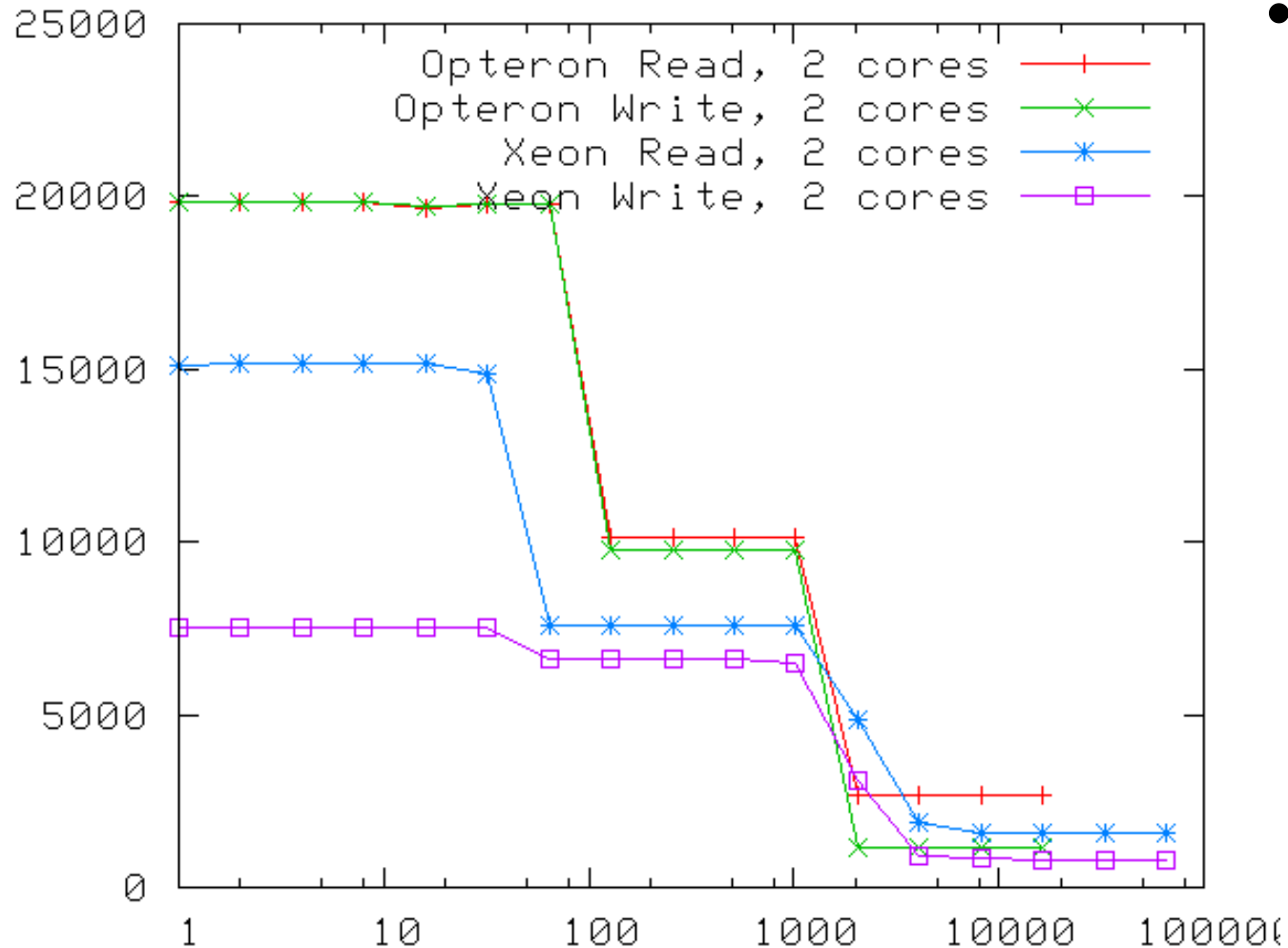
- Very different relative behavior from same job at 1 core.

# Woodcrest at 1 core



- See the change?

# For those who care...



- Opteron had better dual-core bandwidth a year ago

# Take-home Lessons

- Shared caches have very different scaling behaviour from private caches. (Of course, they tend to be larger)
- DRAM bandwidth is shared among cores, and becomes precious.
- Details of cache line request handling can have a big impact on memory interface efficiency.

# Talk Outline

- 1) Cache architecture changes as core count rises
- 2) Benchmark examples of bandwidth contention
- 3) Bandwidth-limited rate for specific code
- 4) Techniques to minimize problems

# Implications for a Code

- Consider a particular code
- WRF, a weather code with lots of SAXPY-like loops
- We'll make up a spreadsheet that examines our predictions

# One loop of WRF

- This is loop 14 of the routine `advect_scalar()`, one of about 720 such routines in the code.
- This code fragment contains:
  - 3 reads, 1 write, 2 mults, 1 add

```
DO i= i_start, i_end
    fgy(i,k,jp1)= 0.5*rv(i,k,j)*(field(i,k,j)+field(i,k,j-1))
ENDDO
```

# Doing a cache line's worth...

- Data is read and written by the cache line (64 bytes)
- That includes 16 operands (WRF is single-precision)
- Memory time to deliver the cache line tends to dominate flops

# WRF Advect\_scalar loop 14, 1 core

- Read 3 cache lines ( $3 \times 64$  bytes): (ignore latency!)
  - From L1:  $\sim 20\text{GB/sec} \rightarrow \sim 3 \times 7 = 21$  cycles
  - From L2:  $\sim 10\text{GB/sec} \rightarrow \sim 3 \times 14 = 42$  cycles
  - From DRAM:  $3.2\text{GB/sec} \rightarrow \sim 3 \times 44 = 132$  cycles
- Write 1 cache line (64 bytes)
  - To L1:  $\sim 1 \times 7 = 7$  cycles
  - To L2:  $\sim 1 \times 14 = 14$  cycles
  - To DRAM:  $\sim 2\text{GB/sec} \rightarrow \sim 1 \times 70 = 70$  cycles
  - Data paths bidirectional- overlaps with read

# WRF advect\_scalar loop 14 cont.

- 2 mults, 1 add
- Do 1 add and 1 mult simultaneously, then do the second add.
- Every 2 cycles we do 6 flops and get 2 results
- 16 results per cache line  $\rightarrow 2 \cdot (16/2) = 16$  cycles to handle a cache line's worth of inputs
- We do 48 flops per cache line

# Predictions for the WRF loop

- If the data resides in L1, we may finish our in as little as 21 cycles, with efficiency as high as (48/21) flops/cycle
  - In practice, instruction scheduling keeps us lower
  - (peak theoretical rate is 4 flops/cycle)
- Data in L2: 48 flops in 42 cycles
- Data in DRAM: 48 flops in 132 cycles
  - But other cores will compete for DRAM!

# Spreadsheet

- Let's look at a spreadsheet that implements this calculation for a few loops, and makes a very rough prediction for WRF.

# Take-home Lessons

- If your code doesn't fit in cache, ignore floating point efficiency. Memory bandwidth is much more important!
- If you distribute a fixed-size problem over enough cores, it may condense into cache, giving an unexpected speed-up.
  - (Of course, network delays may prevent that...)

# Talk Outline

- 1) Cache architecture changes as core count rises
- 2) Benchmark examples of bandwidth contention
- 3) Bandwidth-limited rate for specific code
- 4) Techniques to minimize problems

# Coping Strategies

- Cache reuse through loop mergers
- Blocked algorithms
- Linear memory access patterns
  - Activates hardware prefetch and minimizes latency
- Run on enough cores that your problem condenses into cache!

# Mixed-Mode Programming

- In pure message-passing mode, it's MPI all the way
  - e.g. Bigben
  - Shmem, portals as well, of course
- In mixed mode, run threaded on the SMP nodes if possible
  - e.g. Lemieux
  - Threading allows each core to make better use of other core's caches or shared cache

# Examples

- WRF is my example of bad behavior
- PKDGrav has an interesting linear fetch strategy that improves tree-traversal efficiency
- Paul Woodward's 'sugar cubes'

# Paul Woodward's PPM Implementation

- Colella and Woodward developed piecewise-parabolic-mesh hydro (1984).
- Woodward made it fast.
- Assumptions:
  - It's most efficient to read and write whole cache lines
  - Special support for SIMD ops of 4 operands (e.g. Xeon/Opteron SSE)
  - Swizzling: reordering those groups of 4

# 'Sugar Cubes'

- Data grid is divided into 4x4x4 'sugar cubes'.
- 3 cubes' worth of all fields are in play at any time:
  - One being read into cache
  - One having **all** needed flops performed
  - One being written out
- Swizzling allows math in all 3 directions

# Results

- This approach allows hundreds of flops per sugar cube, plenty to amortize the cost of reading the cache line.
- (Data in DRAM is stored in cube order- kind of confusing).
- Flop rate is a large fraction of peak
- And his F90 code is actually fun to read!

# Programming Tricks

- Memory locality is key!
  - Compute in registers, load and store as little as possible
  - Flops are cheap; Recalculating may be faster than reuse
  - Avoid saving temporaries to full-sized arrays
  - Write data structures to fit in a cache line
- Use tuned libs where possible
- Single precision can be twice as fast as double

# Programming Tricks 2

- Use inlining. It lets the compiler worry less about false aliasing.
- ‘ivdep’ pragma can avoid wasted write cycles
- If writing in C, align things to 16 byte boundaries
  - This is a property of the memory access hardware
  - Fortran compilers try to do this automatically



Questions?