

Multi-core processors are here, but how do you resolve data bottlenecks in native code?

hint: it's all about locality

Michael Wall | October, 2008



part I of II: System memory



Session Prerequisites

- Working knowledge of C/C++
- Basic understanding of microprocessor concepts
- Interest in making software run faster



Session Objectives and Agenda

Software optimization techniques related to memory

- Data strategies to help maximize memory performance
- NUMA
- AMD CodeAnalyst profiler



The big trend: a lot more cores...

...but memory performance does not increase as much

Moore's Law is alive and well, but working in a new way

- We still get 2x more transistors every 18-24 months...
- But instead of higher single-thread performance, we get more cores
- For developers, this can change almost everything



The big trend: a lot more cores...

...but memory performance does not increase as much

Multi-threading or multi-processing is essential for performance

- Threading must become routine, like using printf or “for” loops
- OpenMP, ConcRT and threaded library code help make it easier
- But threading is not really the focus of this talk

Many cores + limited memory bandwidth/latency =
data bottlenecks

- This talk is about how to help reduce the bottlenecks



What does this all mean for you?

Now we talk about software!



A useful cross-platform approximation

cache architecture and memory latency

Every CPU may have different specs for cache

Different computers have different memory systems

But we can still make a useful, general approximation:

- Fastest data access time is L1 cache, let's call it 1x
- Higher cache levels are 10x
- Main memory is 100x

The point is: accessing main memory is very expensive!



#1 Optimize data layout to hide latency

Latency to access main memory is the most common S_L_O_W operation performed by software

Idea: help reduce the effect of latency by issuing memory requests sooner

Problem: the CPU cannot issue a memory request until the address is known

How can the CPU generate an address sooner?



#1 Optimize data layout to hide latency

Arrays vs. linked lists

Array

Easy random access
Hard to insert/remove data
Sequential addresses

Address can be *calculated*

Linked list

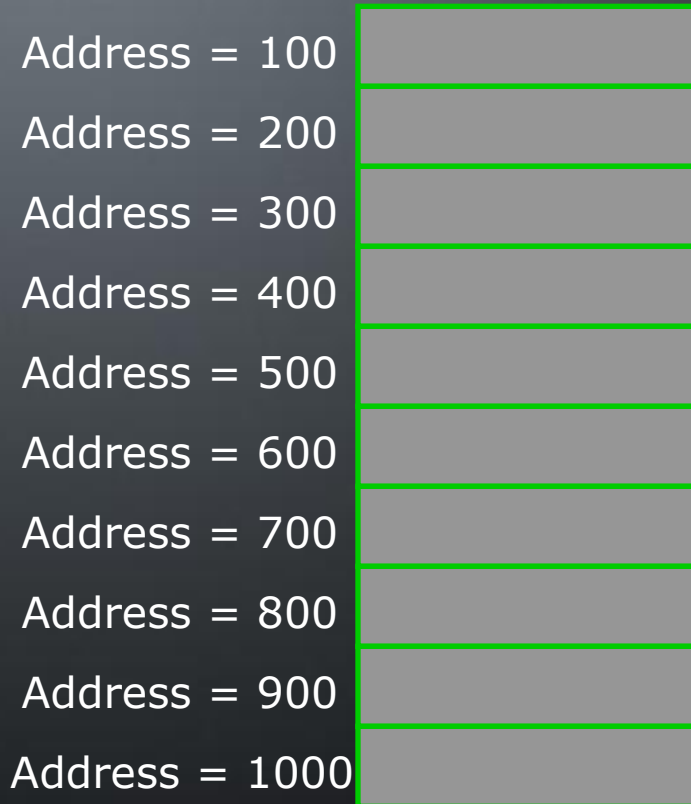
No random access
Easy to insert/remove data ☺
Scattered addresses

Address *depends on data*

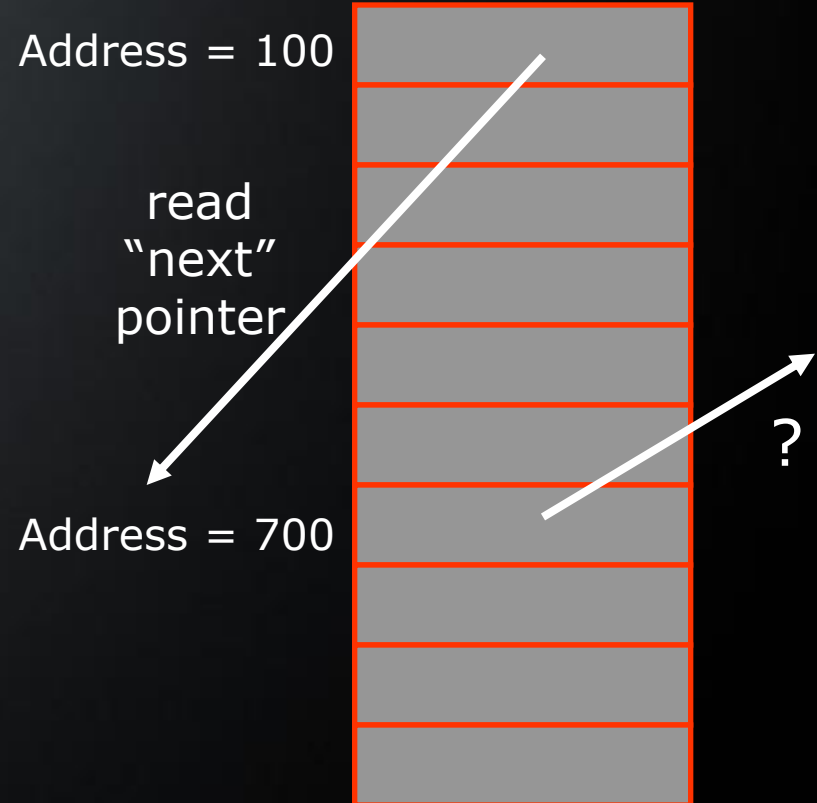


#1 Optimize data layout to hide latency

traversing an array



traversing a linked list

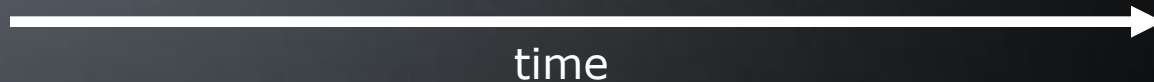


#1 Optimize data layout to hide latency

traversing an array:

Fetch address 100
Fetch address 200
Fetch address 300
Fetch address 400
Fetch address 500

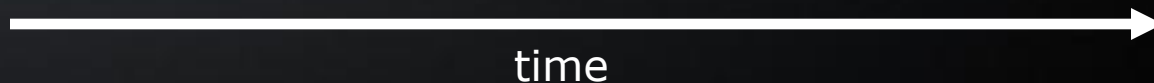
*Memory requests overlap!
This is a form of ILP
(Instruction-Level Parallelism)*



traversing a linked list:

Memory requests cannot overlap!

Fetch address 100... wait for "next" data from memory... Fetch address 700...



Demo

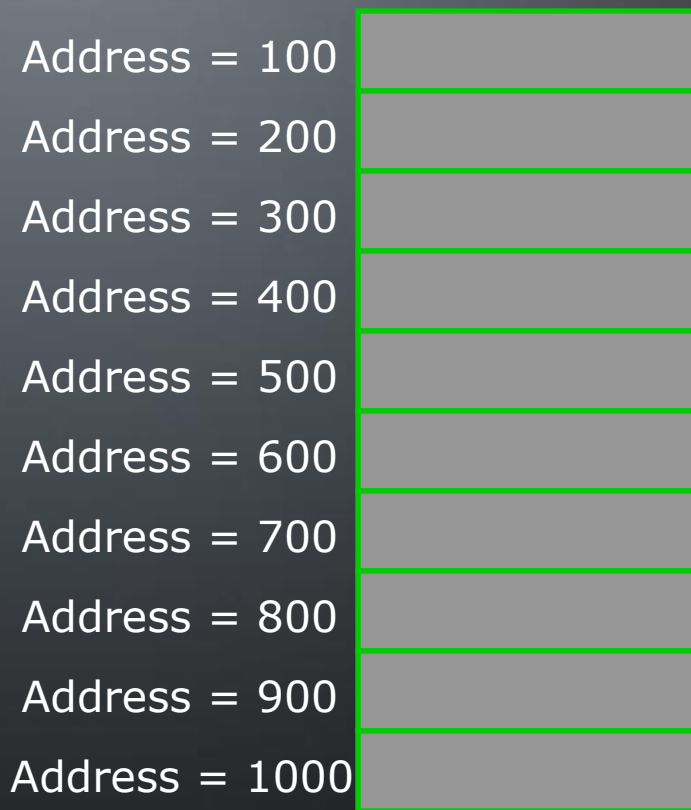
Array and linked list example in Visual Studio



#1 Optimize data layout to hide latency

get higher performance from linked lists

traversing an array



traversing a *sorted* linked list



Locality helps enable prefetch to be effective

Both hardware and software prefetch can help

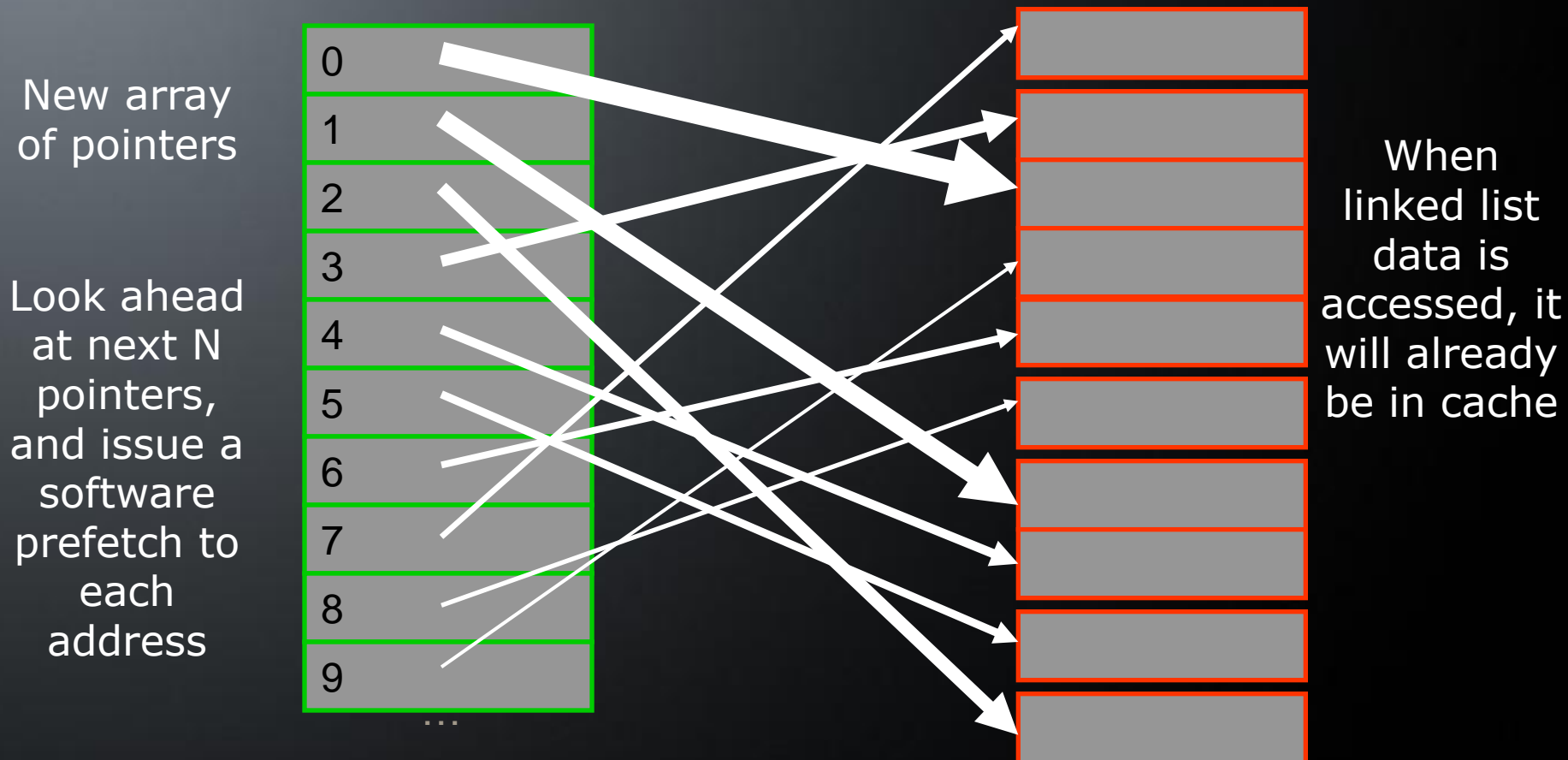
Sometimes it's beneficial to look at the actual *value* of an address pointer !



#1 Optimize data layout to hide latency

get higher performance from linked lists

Another trick: build a dynamic index array to guide software prefetch



This method requires relatively small changes to your code 😊



#1a A word about the TLB

TLB is the Translation Lookaside Buffer

Windows® uses Virtual Memory

Every address must be translated from virtual to physical

This translation mechanism is optimized: TLB

TLB is just like a cache, but it stores memory page addresses

- Typical page size is 4K

TLB has a finite size

- On latest AMD processors, L1 TLB has 48 entries, L2 has 512

Just remember this:

Locality of data helps the TLB work effectively

- Fewer TLB misses = high performance
- More important with larger data sets



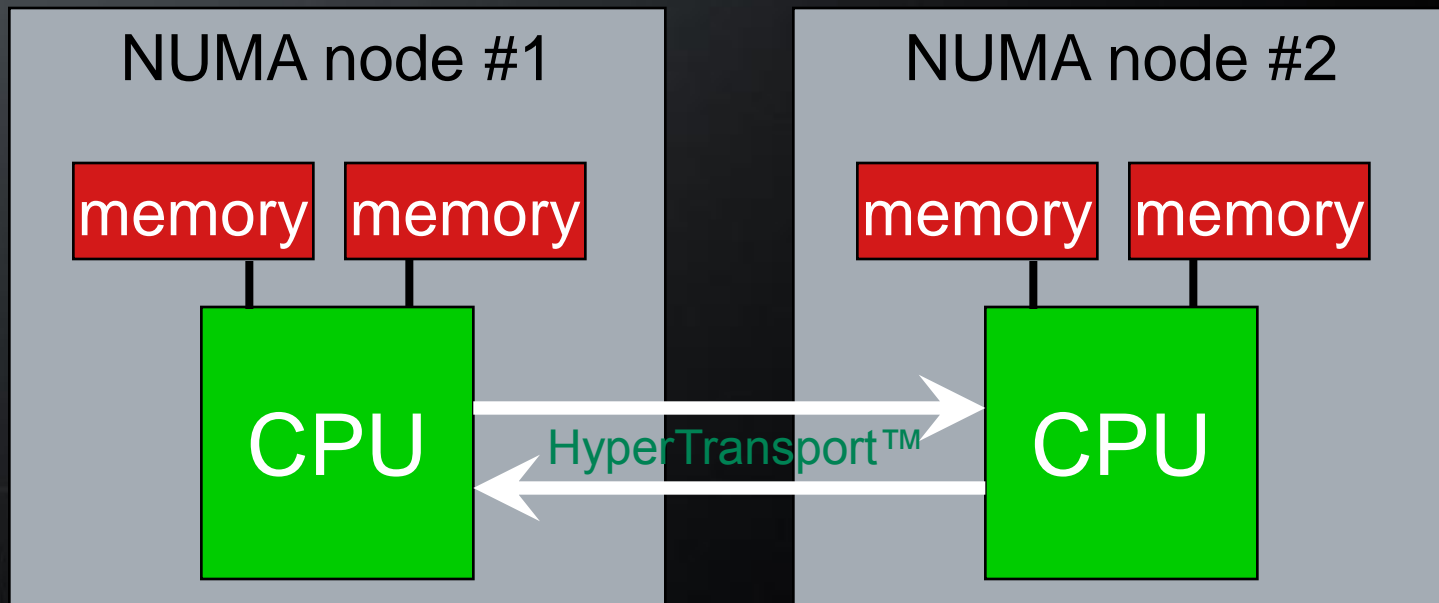
#2 NUMA and memory affinity

NUMA = Non Uniform Memory Access

Multi-socket AMD Opteron™ processor-based machines are NUMA architectures, which is currently the industry trend

- Multiple memory controllers = high total system bandwidth
- Local memory bank has somewhat low *latency*
- If threads/processes uses local bank, more *bandwidth* is available

OS can automatically keep a process on a NUMA node



#2 NUMA and memory affinity

NUMA = Non Uniform Memory Access

Multi-threaded applications should be NUMA-aware!

GetLogicalProcessorInformation API in Windows®

- Identify which logical processors are in which NUMA node
- Also reveals cache sharing and other details

SetThreadAffinityMask can *restrict a thread* to certain NUMA node(s) or logical processors

Allocate memory with affinity by calling allocation function from the desired NUMA node

Also GetNumaHighestNodeNumber and other functions

See MSDN for all the details on NUMA API functions



#3 AMD CodeAnalyst Profiler

Optimization requires real, measured performance data

See what your code is actually doing

- No modification to your code is necessary
- View performance data for *all processes* that are running

Timer based sampling

- The classic “find the hot spots” analysis

Event based sampling

- L1 cache miss, L2 cache miss, I-cache miss, branch mispredict, misaligned memory access, TLB miss, etc.

Download AMD CodeAnalyst from developer.amd.com

(you need to register, if you haven't already, and then login)



Homework assignment! 😊

Profile your code using the AMD CodeAnalyst tool

See what are the “hot spots”

- Are you surprised by what you see?

If your code uses linked lists:

- Identify which data member(s) get accessed in hot spots
- Be sure to look for ‘cache miss’ and ‘TLB miss’ events
- Re-order struct to place “hot” members near “next” link

For extra credit:
sort your linked list
by address order,
and try using
prefetchNTA 😊

```
struct foo {
```

```
    - foo* next;
    - char name[48];
    - BOOL hot_variable;
```



Bad! The “hot”
members are separated

```
struct foo {
```

```
    - foo* next;
    - BOOL hot_variable;
    - char name[48];
```



Good! The “hot” members
are together, and can ride
on the same cache line...
faster!



Related Content

Visit <http://developer.amd.com>

- Software Optimization Guide for “Barcelona“ Family 10h*
Bios and Kernel Developer’s Guide for “Barcelona” Family 10h
- Optimization white paper in Windows® section:
 - “Performance Optimization of Windows Applications on AMD Processors, Part I and II”
 - Many optimization techniques, relevant to 32-bit and 64-bit
 - Other papers on NUMA, compilers, virtualization, etc.
- Ben Sander’s “Barcelona” slides from Microprocessor Forum on developer.amd.com

*Family 10h = AMD Barcelona CPUs



Multi-core processors are here, but how do you resolve data bottlenecks in native code?

hint: it's all about locality

Michael Wall | October, 2008



part II of II: The Cache



Session Prerequisites

- Working knowledge of C/C++
- Basic understanding of microprocessor concepts
- Interest in making software run faster



Session Objectives and Agenda

General cache optimization techniques

- Data layout to help maximize cache efficiency
- Explicit cache control
- AMD CodeAnalyst profiler



#1 Optimize data layout for cache

A cache line is the “atomic unit” of storage

- Read one byte from memory, and you get 64 bytes
- *Make sure the other 63 bytes are useful!*



Maximize data locality



#1 Optimize data layout for cache

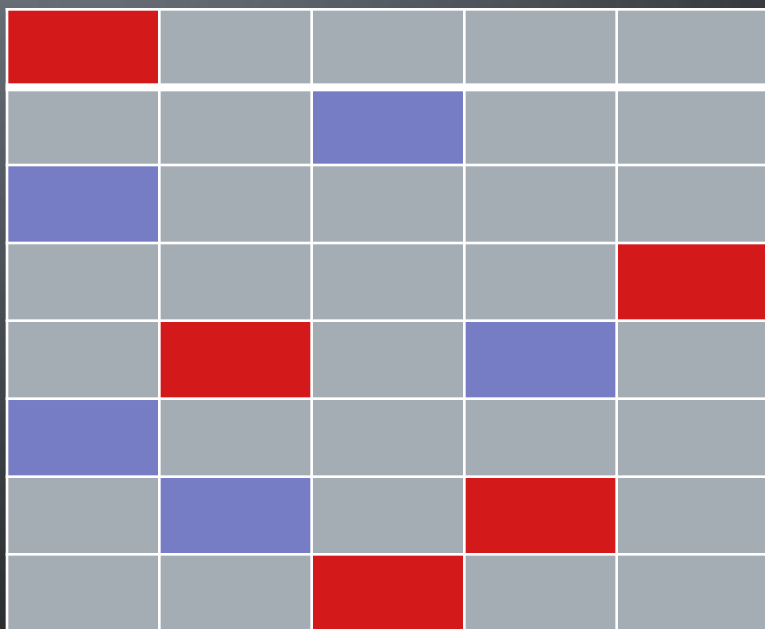
Dense packing of data in struct and class members

- Use the smallest appropriate data type
 - Use byte instead of int to store a small integer
 - Use a small index value (byte/short/int) instead of a pointer when possible
 - Beware: the compiler may use padding to align elements!
- Consider splitting struct into “hot” and “cold” parts
 - Hot part contains all the frequently used elements
 - Cold part stores elements that are used less often
 - Based on dynamic behavior with real workloads
 - You really should profile your code, to learn this

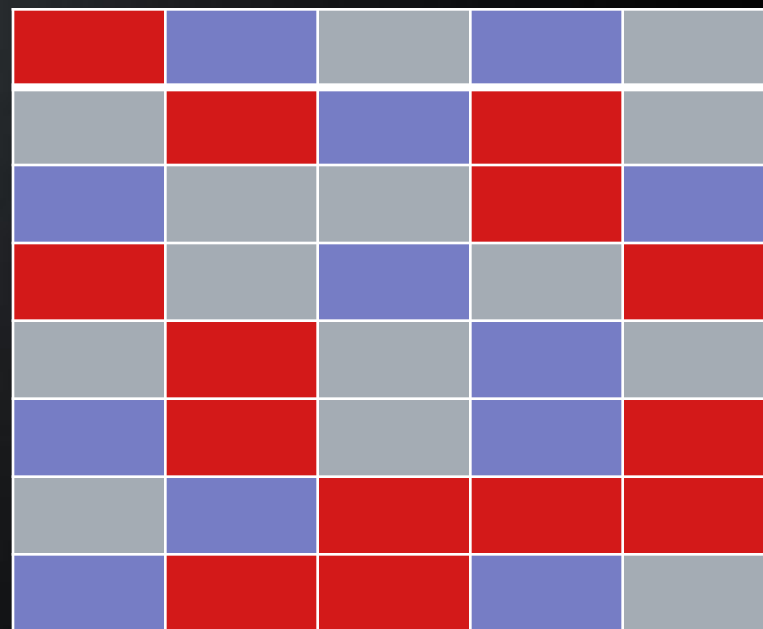


#1 Optimize data layout for cache

Low utilization of cache



Dense packing of useful data



More useful data per cache line = few memory requests = high performance 😊



#1 Optimize data layout for cache

considerations for the new level 3 cache

Share what's in Level 3 cache, between threads

- Symmetric multi-core access to “Barcelona’s” L3 cache
- *Shared data stays in L3 longer than non-shared data*
- L3 is ideal for producer/consumer multi-threading models
- L3 will tend to store code, if you have a large code set
 - Run the same code on multiple threads (cores)
 - Data-parallel threading, e.g. OpenMP loops



#2 Cache management: help avoid re-loading

cache blocking of data: increase locality

Simple non-block way: probably not optimal for large data sets

It uses more memory bandwidth than necessary, re-loading data

```
For each data item, do process A; // read memory
```

```
For each data item, do process B; // read memory
```

```
For each data item, do process C; // read memory
```

Process *blocks* of data in-cache: only read memory once!

```
Divide data into cache-friendly blocks of N items
```

```
For each block of N data items do
```

```
{
```

```
  Do process A; // read memory
```

```
  Do process B; // data still in cache!
```

```
  Do process C; // data still in cache!
```

```
}
```

Fast!



#3 Cache management: non-temporal data

Avoid filling the cache with data you don't need again

- Non-temporal data means “don't need it again soon”
- Only keep cache data that has valuable *temporal locality*

Loading non-temporal data: prefetchNTA instruction

- Non-temporal SW prefetch reads data into L1 cache, of course
- Like any prefetch, it can help hide memory latency
- ▪ Will *not evict* to L2 later; data just gets over-written in L1
- Execute one prefetch per 64-byte cache line, ideally
- Implemented as a compiler intrinsic function in Visual Studio
 - Easy to use!
 - `_mm_prefetch(char* ptr, _MM_HINT_NTA);`



#3 Cache management: non-temporal data

The other way data gets into cache is by executing store instructions, e.g. when your code writes to “memory”

You are actually writing to the cache, not to memory

And nearby memory data must be *read* into cache first

- Remember: a cache line is an atomic 64-byte unit
- It gets filled from memory when allocated

Storing non-temporal data: `movNTxx` instructions

- Does not allocate a cache line, uses a 64-byte *write-combine buffer*
- Store a complete SSE register, packed int or fp data
- `_mm_stream_ps(float* ptr, __m128 a); // 16 bytes`

Scalar streaming store: new in AMD processors and Visual Studio 2008!

No awkward data packing required

- `_mm_stream_ss(float* ptr, __m128 a); // 4 bytes`
- `_mm_stream_sd(double* ptr, __m128 a); // 8 bytes`



#3a A word about instruction cache

A large working set of code can stress I-cache

- Compiling for small code may give superior performance!
- Try compiling with /O1
- Sometimes /O2 or /Ox are not the fastest

Also try Whole Program Optimization

- /GL and /LTCG
- And try profile guided optimization

Good D-cache (data cache) management can also help reduce I-cache pressure, because L2 and L3 store both instructions and data



#4 AMD CodeAnalyst Profiler

Optimization requires real, measured performance data

See what your code is actually doing

- No modification to your code is necessary
- View performance data for *all processes* that are running

Timer based sampling

- The classic “find the hot spots” analysis

Event based sampling

- L1 cache miss, L2 cache miss, I-cache miss, branch mispredict, misaligned memory access, TLB miss, etc.

Download AMD CodeAnalyst from developer.amd.com

(you need to register, if you haven't already, and then login)



Homework assignment! 😊

Profile your code using the AMD CodeAnalyst tool

See what are the “hot spots”

- Are you surprised by what you see?

If your code uses linked lists:

- Identify which data member(s) get accessed in hot spots
- Be sure to look for ‘cache miss’ and ‘TLB miss’ events
- Re-order struct to place “hot” members near “next” link

For extra credit:
sort your linked list
by address order,
and try using
prefetchNTA 😊

```
struct foo {
```

```
    - foo* next;
    - char name[48];
    - BOOL hot_variable;
```



Bad! The “hot”
members are separated

```
struct foo {
```

```
    - foo* next;
    - BOOL hot_variable;
    - char name[48];
```



Good! The “hot” members
are together, and can ride
on the same cache line...
faster!



Related Content

Visit <http://developer.amd.com>

- Software Optimization Guide for “Barcelona“ Family 10h
- Bios and Kernel Developer’s Guide for “Barcelona” Family 10h
- Optimization white paper in Windows® section:
 - “Performance Optimization of Windows Applications on AMD Processors, Part I and II”
 - Many optimization techniques, relevant to 32-bit and 64-bit
 - Other papers on NUMA, compilers, virtualization, etc.
- Ben Sander’s “Barcelona” slides from Microprocessor Forum on developer.amd.com

*Family 10h = AMD Barcelona CPUs



DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

AMD, the AMD Arrow logo, ATI, the ATI logo, AMD Opteron and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Windows is a registered trademark of Microsoft Corporation in the United States and/or other jurisdictions. HyperTransport is a licensed trademark of the HyperTransport Technology Consortium. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2008 Advanced Micro Devices, Inc. All rights reserved.

