A Tale of Two Traversals

Two ways to traverse a matrix:

- Each touches exactly the same memory.

Row Major

Column Major
A Tale of Two Traversals

Code very similar:

```c
void sumMatrix(const Matrix<int>& m,
               long long& sum, TraversalOrder order)
{
    sum = 0;
    if (order == RowMajor) {
        for (unsigned r = 0; r < m.rows(); ++r) {
            for (unsigned c = 0; c < m.columns(); ++c) {
                sum += m[r][c];
            }
        }
    } else {
        for (unsigned c = 0; c < m.columns(); ++c) {
            for (unsigned r = 0; r < m.rows(); ++r) {
                sum += m[r][c];
            }
        }
    }
}
```

A Tale of Two Traversals

Performance isn’t:
A Tale of Two Traversals

Traversal order matters.

Why?

A Scalability Story

Herb Sutter’s scalability issue in counting odd matrix elements.

- Square matrix of side DIM with memory in array **matrix**.
- Sequential pseudocode:

```c
int odds = 0;
for( int i = 0; i < DIM; ++i )
    for( int j = 0; j < DIM; ++j )
        if( matrix[*DIM + j] % 2 != 0 )
            ++odds;
```
A Scalability Story

- Parallel pseudocode, take 1:

```java
int result[P];
// Each of P parallel workers processes 1/P-th of the data;
// the p-th worker records its partial count in result[p]
for (int p = 0; p < P; ++p )
    pool.run( 
        [&,p] { 
            result[p] = 0;
            int chunkSize = DIM/P + 1;
            int myStart = p * chunkSize;
            int myEnd = min( myStart+chunkSize, DIM );
            for( int i = myStart; i < myEnd; ++i )
                for( int j = 0; j < DIM; ++j )
                    if( matrix[*DIM + j] % 2 != 0 )
                        ++result[p]; } );

pool.join(); // Wait for all tasks to complete
odds = 0; // combine the results
for( int p = 0; p < P; ++p )
    odds += result[p];
```

A Scalability Story

Scalability unimpressive:

![Graph showing speedup over 1 thread for Example 1](attachment://graph.png)

- Faster than 1 core
- Slower than 1 core
A Scalability Story

- Parallel pseudocode, take 2:

```cpp
int result[P];
for (int p = 0; p < P; ++p )
  pool.run( [&,&] { int count = 0; // instead of result[p]
                  int chunkSize = DIM/P + 1;
                  int myStart = p * chunkSize;
                  int myEnd = min( myStart+chunkSize, DIM );
                  for( int i = myStart; i < myEnd; ++i )
                    for( int j = 0; j < DIM; ++j )
                      if( matrix[i*DIM + j] % 2 != 0 )
                        ++count; // instead of result[p]
                  result[p] = count; } ); // new statement
... // nothing else changes
```

Scalability now perfect!

![Speedup for Example 2](image-url)
A Scalability Story

Thread memory access matters.

Why?

CPU Caches

Small amounts of unusually fast memory.

- Generally hold contents of recently accessed memory locations.
- Access latency much smaller than for main memory.
CPU Caches

Three common types:

- **Data** (D-cache, D$)
- **Instruction** (I-cache, I$)
- Translation lookaside buffer (TLB)
  - Caches virtual → real address translations

Voices of Experience

Sergey Solyanik (from Microsoft):

Linux was routing packets at ~30Mbps [wired], and wireless at ~20. Windows CE was crawling at barely 12Mbps wired and 6Mbps wireless. ...

We found out Windows CE had a LOT more instruction cache misses than Linux. ...

After we changed the routing algorithm to be more cache-local, we started doing 35MBps [wired], and 25MBps wireless - 20% better than Linux.
Voices of Experience

Jan Gray (from the MS CLR Performance Team):
If you are passionate about the speed of your code, it is imperative that you consider ... the cache/memory hierarchy as you design and implement your algorithms and data structures.

Dmitriy Vyukov (developer of Relacy Race Detector):
Cache-lines are the key! Undoubtedly! If you will make even single error in data layout, you will get 100x slower solution! No jokes!

Cache Hierarchies

Cache hierarchies (multi-level caches) are common.

E.g., Intel Core i7-9xx processor:
- 32KB L1 I-cache, 32KB L1 D-cache per core
  - Shared by 2 HW threads
- 256 KB L2 cache per core
  - Holds both instructions and data
  - Shared by 2 HW threads
- 8MB L3 cache
  - Holds both instructions and data
  - Shared by 4 cores (8 HW threads)
CPU Caches and Why You Care

CPU Cache Characteristics

Caches are small.
- Assume 100MB program at runtime (code + data).
  - 8% fits in core-i79xx’s L3 cache.
  - L3 cache shared by every running process (incl. OS).
  - 0.25% fits in each L2 cache.
  - 0.03% fits in each L1 cache.

Caches much faster than main memory.
- For Core i7-9xx:
  - L1 latency is 4 cycles.
  - L2 latency is 11 cycles.
  - L3 latency is 39 cycles.
  - Main memory latency is 107 cycles.
  - 27 times slower than L1!
  - 100% CPU utilization ⇒ >99% CPU idle time!
Effective Memory = CPU Cache Memory

From speed perspective, total memory = total cache.
- Core i7-9xx has 8MB fast memory for *everything*.
  - Everything in L1 and L2 caches also in L3 cache.
- Non-cache access can slow things by orders of magnitude.

Small = fast.
- No time/space tradeoff at hardware level.
- Compact, well-localized code that fits in cache is fastest.
- Compact data structures that fit in cache are fastest.
- Data structure traversals touching only cached data are fastest.

Cache Lines

Caches consist of lines, each holding multiple adjacent words.
- On Core i7, cache lines hold 64 bytes.
  - 64-byte lines common for Intel/AMD processors.
  - 64 bytes = 16 32-bit values, 8 64-bit values, etc.
    - E.g., 16 32-bit array elements.

Main memory read/written in terms of cache lines.
- Read byte not in cache ⇒ read full cache line from main memory.
- Write byte ⇒ write full cache line to main memory (eventually).
Cache Lines

Explains why row-major matrix traversal better than column-major:

Cache Line Prefetching

Hardware speculatively prefetches cache lines:

- Forward traversal through cache line \( n \Rightarrow \text{prefetch line } n+1 \)
- Reverse traversal through cache line \( n \Rightarrow \text{prefetch line } n-1 \)
Implications

- **Locality counts.**
  - Reads/writes at address $A \Rightarrow$ contents near $A$ already cached.
    - E.g., on the same cache line.
    - E.g., on nearby cache line that was prefetched.
- **Predictable access patterns count.**
  - “Predictable” = forward or backwards traversals.
- **Linear array traversals very cache-friendly.**
  - Excellent locality, predictable traversal pattern.
  - Linear array search can beat $\log_2 n$ searches of heap-based BSTs.
  - $\log_2 n$ binary search of sorted array can beat $O(1)$ searches of heap-based hash tables.
  - Big-Oh wins for large $n$, but hardware caching takes early lead.

---

Cache Coherency

From core i7’s architecture:

Assume both cores have cached the value at (virtual) address $A$.

- Whether in L1 or L2 makes no difference.

Consider:

- Core 0 writes to $A$.
- Core 1 reads $A$.

What value does Core 1 read?
**Cache Coherency**

Caches a latency-reducing optimization:
- There’s only one virtual memory location with address A.
- It has only one value.

Hardware invalidates Core 1’s cached value when Core 0 writes to A.
- It then puts the new value in Core 1’s cache(s).

Happens automatically.
- You need not worry about it.
  - Provided you synchronize access to shared data...
- But it takes time.

---

**False Sharing**

Suppose Core 0 accesses A and Core 1 accesses A+1.
- *Independent* pieces of memory; concurrent access is safe.
- But A and A+1 probably map to the same cache line.
  - If so, Core 0’s writes to A invalidates A+1’s cache line in Core 1.
    - And vice versa.
    - This is *false sharing*. 
False Sharing

It explains Herb Sutter’s issue:

```
int result[P];  // many elements on 1 cache line
for (int p = 0; p < P; ++p )
    pool.run( [&,&] { // run P threads concurrently
        result[p] = 0;
        int chunkSize = DIM/P + 1;
        int myStart = p * chunkSize;
        int myEnd = min( myStart+chunkSize, DIM );
        for( int i = myStart; i < myEnd; ++i )
            for( int j = 0; j < DIM; ++j )
                if( matrix[i*DIM + j] % 2 != 0 )
                    ++result[p]; } ); // each repeatedly accesses the
                        // same array (albeit different
                        // elements)
```

False Sharing

And his solution:

```
int result[P];  // still multiple elements per
                // cache line
for (int p = 0; p < P; ++p )
    pool.run( [&,&] { // use local var for counting
        int count = 0;
        int chunkSize = DIM/P + 1;
        int myStart = p * chunkSize;
        int myEnd = min( myStart+chunkSize, DIM );
        for( int i = myStart; i < myEnd; ++i )
            for( int j = 0; j < DIM; ++j )
                if( matrix[i*DIM + j] % 2 != 0 )
                    ++count; // update local var
        result[p] = count; } ); // access shared cache line
                        // only once
```
False Sharing

His scalability results are worth repeating:

Problems arise only when all are true:

- Independent values/variables fall on one cache line.
- Different cores concurrently access that line.
- Frequently.
- At least one is a writer.

All types of data are susceptible:

- Statically allocated (e.g., globals, statics).
- Heap allocated.
- Automatics and thread-locals (if pointers/references handed out).
Voice of Experience

Joe Duffy at Microsoft:

_During our Beta1 performance milestone in Parallel Extensions, most of our performance problems came down to stamping out false sharing in numerous places._
Guidance

For data:

- **Where practical, employ linear array traversals.**
  - “I don’t know [data structure], but I know an array will beat it.”

- **Use as much of a cache line as possible.**
  - Bruce Dawson’s antipattern (from reviews of video games):
    ```cpp
    struct Object { // assume sizeof(Object) ≥ 64
        bool isLive; // possibly a bit field
    };
    std::vector<Object> objects; // or an array
    for (std::size_t i = 0; i < objects.size(); ++i) { // pathological if
        if (objects[i].isLive) // most objects
            doSomething(); // not alive
        } // or an array
    }
    ```

- **Be alert for false sharing in MT systems.**

Guidance

For code:

- **Fit working set in cache.**
  - Avoid iteration over heterogeneous sequences with virtual calls.
    - E.g., sort sequences by type.

- **Make “fast paths” branch-free sequences.**
  - Use up-front conditionals to screen out “slow” cases.

- **Inline cautiously:**
  - The good:
    - Reduces branching.
    - Facilitates code-reducing optimizations.
  - The bad:
    - Code duplication reduces effective cache size.

- **Take advantage of PGO and WPO.**
  - Can automate some of above.
Beyond Surface-Scratching

Cache-related topics not really addressed:

- Other cache technology issues:
  - Memory banks.
  - Associativity (but wait...).
  - Inclusive vs. exclusive content.

- Latency-hiding techniques.
  - Hyperthreading.

- Cache performance evaluation:
  - Why it’s critical.
  - Why it’s hard.
  - Tools that can help.

- Cache-oblivious algorithm design.

Beyond Surface-Scratching

Overall cache behavior can be counterintuitive.

Matrix traversal redux:

- Matrix size can vary.

- For given size, shape can vary:
Beyond Surface-Scratching

Row major traversal performance unsurprising:

Beyond Surface-Scratching

Column major a different story:
Beyond Surface-Scratching

A slice through the data:

- White ⇒ fast.
- Blue ⇒ slow.

Igor Ostrovsky’s demonstration of cache-associativity effects.

Columns = 200
CPU caches:

  → Relevant section title is “Of Cache Misses, Page Faults, and Computer Architecture”
- “Memory is not free (more on Vista performance),” Sergey Solyanik, *[1-800-Magic](http://www.1-800-magic.com)* (Blog), 9 December 2007.
  → Experience report about optimizing use of I-cache.

---

CPU caches:

  → Discusses false sharing.
  → Video of a *Build 2013* presentation.
  → Note ryg’s comment about per-type operation batching.
Further Information

Data-oriented design:
  - Conference keynote presentation.
  - Video available at YouTube.

Further Information

Profile-guided optimization (PGO):
  - Still a very nice overview.
  - Much code optimization info, including PGO for gcc.
Further Information

More on PGO:
  - Using PGO to change DS layouts to improve D$ performance.
  - Video interview about PGO support in MS Visual C++.

Further Information

Whole-program optimization (WPO):
  - Enables elimination of unused global data.
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- **Commercial use**: [http://aristeia.com/Licensing/licensing.html](http://aristeia.com/Licensing/licensing.html)
- **Personal use**: [http://aristeia.com/Licensing/personalUse.html](http://aristeia.com/Licensing/personalUse.html)

Courses currently available for personal use include:

About Scott Meyers

Scott is a trainer and consultant on the design and implementation of C++ software systems. His web site, [http://www.aristeia.com/](http://www.aristeia.com/) provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog