Cache Algorithms

Motivation

- We've seen basics of how cache works
- Now we'd like to know how to structure our code to take advantage of it

Example

- In C, arrays are stored in *row major order*
- This means that for this declaration: int A[5][3]; Data is stored in RAM as: A00, A01, A02, A10, A11, A12, A20, A21, A22, A30, ... , A42

Iteration

- This means if code is iterating over 2D array (or 3D or 4D or...) it's best to iterate over *inner* dimension inside innermost loop.

Example

- Consider this code:

```
for(int i=0;i<SIZE;++i){
    for(int j=0;j<SIZE;++j){
        A[i][j] += value;
    }
}
```

Execute

- When I run this (SIZE=4000, 64MB total), cachegrind says this:

```
D   refs:      176,707,065  (160,544,242 rd   + 16,162,823 wr)
D1  misses:      1,016,175  (  1,013,882 rd   +      2,293 wr)
LLd misses:      1,009,612  (  1,008,102 rd   +      1,510 wr)
D1  miss rate:         0.6% (        0.6%     +        0.0%  )
LLd miss rate:         0.6% (        0.6%     +        0.0%  )
```

Restructure

- Suppose we restructure the code:

```
for(int j=0;j<SIZE;++j){
    for(int i=0;i<SIZE;++i){
        A[i][j] += value;
    }
}
```

Results

- I get this:

```
D   refs:      176,707,065  (160,544,242 rd   + 16,162,823 wr)
D1  misses:     16,016,175  ( 16,013,882 rd   +      2,293 wr)
LLd misses:      1,009,612  (  1,008,102 rd   +      1,510 wr)
D1  miss rate:         9.1% (       10.0%     +        0.0%  )
LLd miss rate:         0.6% (        0.6%     +        0.0%  )
```

- Notice level 1 cache misses go up by 16×
- But since whole thing fits in last level of cache, last level misses don't change

Moral

- If you're iterating over an array, it's best to iterate over the innermost dimension in the innermost for-loop
  - And work out dimension-wise as you go out nesting-wise
- Same idea applies for 3D arrays: int A[8][5][3]
  - In RAM: A000, A001, A002, A010, A011, A012, A020, A021, ... A042, A100, A101, ... , A742
- Likewise for larger arrays

However...

- The above results were obtained with compiler optimization turned off
- When optimization turned on, I get same results both way:

```
D   refs:       8,691,042  (4,532,232 rd   + 4,158,810 wr)
D1  misses:     1,016,176  (1,013,881 rd   +     2,295 wr)
LLd misses:     1,009,613  (1,008,102 rd   +     1,511 wr)
D1  miss rate:       11.7% (     22.4%     +       0.1%  )
LLd miss rate:       11.6% (     22.2%     +       0.0%  )
```

- Why?

Compiler

- Compilers have been tuned to recognize this sort of access pattern
- They then invert the loop iteration if needed
- Sometimes, compiler may not be able to detect that this transform is possible
  - So then it's better if *you* do it

CPU

- If you intend to write data but not use it in near future, can tell CPU about this
- Can also use CPU instructions that access memory more efficiently (SIMD)
- Example: With CPU-specific optimizations turned on:

```
D   refs:       4,691,046  (2,532,234 rd   + 2,158,812 wr)
D1  misses:     1,016,173  (1,013,886 rd   +     2,287 wr)
LLd misses:     1,009,615  (1,008,104 rd   +     1,511 wr)
D1  miss rate:       21.7% (     40.0%     +       0.1%  )
LLd miss rate:       21.5% (     39.8%     +       0.1%  )
```

- The miss rate is doubled, but the total number of references is halved
- The total number of misses remains almost constant

Example

- Suppose we're doing a physics simulation
- We have several spheres we're processing:

```
struct vec4{
    double x,y,z,w;
};
struct Sphere{
    vec4 center;
    double radius;
    double elasticity;
    vec4 velocity;
    vec4 color;
    vec4 acceleration;
};
```

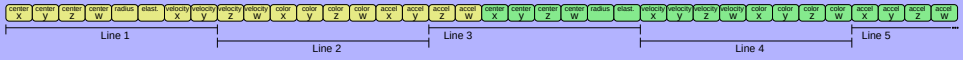
Example

- Suppose we're going to update each object's position...

```
void update(vector<Sphere>& S){
    for(unsigned i=0;i<S.size();++i){
        s[i].center = s[i].center + t * S[i].velocity;
    }
}
```

Cache

- Suppose our cache line size is 64 bytes
- This would accommodate 2 vec4's
- So one cache line could fit center + radius + velocity x,y,z
- Next cache line would store velocity w, color r,g,b,a, acceleration x,y,z



Analysis

- We'll see two cache misses: One for line 1, one for line 2
- On the next sphere, same thing: Miss for line 3, miss for line 4
- The auto-prefetcher will probably cover most of these on its own

However

- What if we had arranged the sphere like so:

```
struct Sphere{
    vec4 center;
    double radius;
    double elasticity;
    vec4 color;
    vec4 acceleration;
    vec4 velocity;
};
```

Cache

- Notice that now we need to fetch three lines for the first sphere
- That helps us with the second sphere: Center is already in cache
- But: We don't need line 4 at all!
  - The prefetcher will get it for us, but that's wasted effort
  - We'll likely see a miss on line 5

Numbers

- Let's get some hard data
- Here's example code:

```
vector<Sphere> S;
S.resize(100000);
double t = 0.05;
for(unsigned i=0;i<S.size();++i){
    S[i].center = S[i].center + t * S[i].velocity;
}
```

Output

- First, consider the initial layout for Sphere
  - center, radius, elasticity, velocity, color, acceleration
- I used cachegrind to profile the code compiled with and without optimization
- Here's what I got...

-O0

- No optimization:

```
==7903== Cachegrind, a cache and branch-prediction profiler
==7903== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==7903== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7903== Command: ./spheretest1-1-O0
==7903==
--7903-- warning: L3 cache found, using its data for the LL simulation.
==7903==
==7903== I   refs:      34,280,395
==7903== I1  misses:         1,556
==7903== LLi misses:         1,497
==7903== I1  miss rate:       0.00%
==7903== LLi miss rate:       0.00%
==7903==
==7903== D   refs:      21,689,508  (12,031,277 rd   + 9,658,231 wr)
==7903== D1  misses:       441,107  (   213,830 rd   +   227,277 wr)
==7903== LLd misses:       434,535  (   208,031 rd   +   226,504 wr)
==7903== D1  miss rate:        2.0% (       1.8%     +       2.4%  )
==7903== LLd miss rate:        2.0% (       1.7%     +       2.3%  )
==7903==
==7903== LL refs:          442,663  (   215,386 rd   +   227,277 wr)
==7903== LL misses:        436,032  (   209,528 rd   +   226,504 wr)
==7903== LL miss rate:         0.8% (       0.5%     +       2.3%  )
```

-O3

- With optimization:

```
==7904== Cachegrind, a cache and branch-prediction profiler
==7904== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==7904== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7904== Command: ./spheretest1-1-O3
==7904==
--7904-- warning: L3 cache found, using its data for the LL simulation.
==7904==
==7904== I   refs:      6,480,917
==7904== I1  misses:        1,507
==7904== LLi misses:        1,452
==7904== I1  miss rate:      0.02%
==7904== LLi miss rate:      0.02%
==7904==
==7904== D   refs:      3,088,915  (930,943 rd   + 2,157,972 wr)
==7904== D1  misses:      441,100  (213,821 rd   +   227,279 wr)
==7904== LLd misses:      434,530  (208,023 rd   +   226,507 wr)
==7904== D1  miss rate:      14.3% (   23.0%     +      10.5%  )
==7904== LLd miss rate:      14.1% (   22.3%     +      10.5%  )
==7904==
==7904== LL refs:         442,607  (215,328 rd   +   227,279 wr)
==7904== LL misses:       435,982  (209,475 rd   +   226,507 wr)
==7904== LL miss rate:        4.6% (    2.8%     +      10.5%  )
```

Wow!

- Take a look at the cache misses:
  - No optimization: D1 miss rate: 2.0%
  - Full optimization: D1 miss rate: 14.3%
- It seems like optimization seriously hurts our hit rate!

But...

- Take a closer look...
  - No optimization:
    - D refs: 21,689,508
    - D1 misses: 441,107
  - Optimization:
    - D refs: 3,088,915
    - D1 misses: 441,100
- The total number of misses remains nearly constant
- But optimized code has fewer overall memory accesses, so the miss *rate* goes up while the miss *count* does not
- Moral: Beware of looking at only *part* of your data!

Compare

- We'll consider only the optimized executions
- Comparing the two sphere layouts...
  - Number of memory references equal, so OK to look at percents here
- First one: D1 miss rate: 14.3%
- Second one: D1 miss rate: 13.5%
- So: It makes some difference, but not much

Layout

- One problem: Data is spread out in memory
- Caches work best when we plow through sequentially
  - Prefetcher can help, but it's not perfect
- Organization of data we have: Called *array of structures* (AoS)

SoA

- Another way: Structure of arrays
- Suppose we organize our spheres like so:

```
struct SphereArray{
    vector<vec4> centers;
    vector<vec4> velocities;
    vector<double> radii;
    vector<double> elasticity;
    vector<vec4> colors;
    vector<vec4> acceleration;
};
```

Code

- Then we write code like so:

```
SphereArray S;
S.centers.resize(100000);
S.velocities.resize(S.centers.size());
S.radii.resize(S.centers.size());
S.elasticity.resize(S.centers.size());
S.colors.resize(S.centers.size());
S.acceleration.resize(S.centers.size());
...fill with data...
double t = ...;
vec4* C = S.centers.data();
vec4* V = S.velocities.data();
for(unsigned i=S.centers.size();i>0;--i,C++,V++){
    *C = *C + t * *V;
}
```

Results

- Here's what I get:
  - D1 misses: 141,181 (113,886 rd + 27,295 wr)
  - D1 miss rate: 4.9% (12.2% + 1.4%)
- Notice: Misses dropped from 416,000 to 141,000
- SoA layout is much more cache friendly
  - And: Each cache line is exact multiple of item size
  - No "half-loaded" pieces in cache

No Free Lunch

- There's no free lunch of course!
- SoA format tends to be less intuitive to work with
- Single object is spread out over memory
- Harder to pass an object to a function that uses it

Enough!

- Enough theory. Let's see some real results
- Recall the raytracer
- I ran it on a mesh model (scene2.txt): 256x256 image: 3.46 seconds
  - The system I ran these tests on has 32KB of L1 cache (per-core), 256KB of L2 cache (per-core), and 8MB L3 cache (shared)

Analysis

- Observe the Triangle class has a lot of information in it:

```
class Triangle{
public:
    vec3 p[3];      //vertices
    vec3 e[3];      //edges: e[i] = p[i+1]-p[i]
    vec3 N;         //normal, unit length
    float D;            //plane equation D
    float oneOverTwiceArea;     //1.0 / twice the triangle's area
    ...
}
```

- Each float is 4 bytes; each vec3 is 3*4 = 12 bytes
- Total size of one triangle: 12*3 + 12*3 + 12 + 4 + 4 = 92 bytes
- scene2.txt has 1536 triangles = 141,312 bytes

Rewrite

- I rewrote the code like so:
  - The Mesh has a single list of vec3's: All the points and normals in the mesh go in here
  - The triangle holds indices instead of vec3's
  - The edge data is computed on the fly as needed rather than being stored:

```
class Triangle{
public:
    typedef uint16_t u16;
    u16 p[3];
    u16 N;
    float D;
    float oneOverTwiceArea; //1.0 / twice the triangle's area
    ...
}
```

Sources

- Michael McCool, Arch D. Robinson, James Reinders. Structured Parallel Programming. Morgan Kaufmann Publishers.
- Harald Prokop. Cache Oblivious Algorithms. http://supertech.csail.mit.edu/papers/Prokop99.pdf
- Stack Overflow. Caching - How does one write code that best utilizes the CPU cache to improve performance? https://www.akkadia.org/drepper/cpumemory.pdf
- Ulrich Drepper. What Every Programmer Should Know About Memory. https://www.akkadia.org/drepper/cpumemory.pdf
- https://pc-builds.com/cpu/Intel_Core_i7-4790/0zU/

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org