Optimization & Vectorization

(or: “Caching (2)”)
Today's Agenda:

- Recall
- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
Fact Checking

INFOMOV – Lecture 10 – "Data-Oriented Design"

https://www.reddit.com/r/Amd/comments/jqjg8e/quick_zen3_die_shot_annotations_die_shot_from/
Previously in INFOMOV: cache architectures

Because: memory is slow.

First architecture: fully associative.

Characteristic: any memory location can be stored anywhere in the cache.

**Pro:**
- Optimal use of the cache.

**Con:**
- O(N) algorithmic complexity.
Previously in INFOMOV: **cache architectures**

Because: memory is *slow*.

Second architecture: **direct mapped**.

Characteristic: a particular address always goes to the same slot in the cache.

**Pro:**
- O(1) algorithmic complexity.

**Con:**
- Collisions.
- Cache may be under-utilized.
Recall

Previously in INFOMOV:  

**cache architectures**

Because: memory is **slow**.

Second architecture: **$N$-way set associative**.

Characteristic: a particular address always goes to the same *set of $N$ slots* in the cache.

**Pro:**
- Can balance algorithmic complexity and collision probability.

**Con:**
- Cache may still be under-utilized.
Recall

**32KB, 8-way set-associative, 64 bytes per cache line:**

**64 sets of 512 bytes**

![32-bit address diagram]

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set nr</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples:

- **0x00001234**
  - tag: 0001
  - set nr: 00100
  - offset: 110100

- **0x00008234**
  - tag: 1000
  - set nr: 00100
  - offset: 110100

- **0x00006234**
  - tag: 0110
  - set nr: 00100
  - offset: 110100

- **0x0000A234**
  - tag: 1010
  - set nr: 00100
  - offset: 110100

- **0x0000A240**
  - tag: 1010
  - set nr: 00100
  - offset: 000000

- **0x0000F234**
  - tag: 1111
  - set nr: 00100
  - offset: 110100
Recall

32KB, 8-way set-associative, 64 bytes per cache line: 64 sets of 512 bytes

![32-bit address diagram]

Examples:

- 0x00001234: 0001 001000 110100
- 0x00008234: 1000 001000 110100
- 0x00006234: 0110 001000 110100
- 0x0000A234: 1010 001000 110100
- 0x0000A240: 1010 001001 000000
- 0x0000F234: 1111 001000 110100
**Recall**

**32KB, 8-way set-associative, 64 bytes per cache line:**
64 sets of 512 bytes

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td></td>
<td>set nr</td>
<td>offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples:

- 0x00001234 0001 001000 110100
- 0x00008234 1000 001000 110100
- 0x00006234 0110 001000 110100
- 0x0000A234 1010 001000 110100
- 0x0000A240 1010 001001 000000
- 0x0000F234 1111 001000 110100

32-bit address

[Diagram showing cache set and slot]
Recall

Details:

- Cache lines:
  - Smallest unit of data transferred from RAM to cache (and between cache levels).
  - On CPU: 64 bytes.
  - On GPU: 128 bytes.
  - A cache slot tag does not store the lowest 6 bits of an address.

- Multiple cores:
  - Synchronize via L3.
  - Mind false sharing!

- Memory system:
  - To the CPU, the memory hierarchy is (mostly) just memory.
  - Exception: _mm_prefetch()
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
Object Oriented Programming:

- Objects
  - Data
  - Methods
  - Instances

“Death by a Thousand Cuts”
“Death by a Thousand Cuts”

Object Oriented Programming:

- **Objects**
  - Data
  - Methods
  - Instances

Cost of a virtual function call:

1. Virtual Function Table
2. No inlining

... 

Calling such a function:

1. Read pointer to VFT of class
2. Add function offset
3. Read function address from VFT
4. Load address in PC (jump)

But, that isn’t realistic, right?

It is, if we use OO for what it was designed for: operating on heterogeneous objects.
OOP

“Death by a Thousand Cuts”

Characteristics of OO:

- Virtual calls
- Scattered individual objects
OOP

“The problem is growing with time.”

The graph shows the performance gap between processor and memory over the years from 1980 to 2005. It illustrates how the performance of memory has lagged behind that of the processor, with a significant gap of 600 cycles at 3.2Ghz and 40 cycles at 300Mhz, indicating that the problem is growing with time.
“Death by a Thousand Cuts”

Dealing with “bandwidth starvation”:

Caching

Continuous memory access (full cache lines)

Large array continuous memory access

(caches ‘read ahead’)
“Death by a Thousand Cuts”

Code performance is typically bound by memory access.

“The ideal data is in a format that we can use with the least amount of effort.”

➔ Effort = CPU-effort.

“Most programs are made faster if we improve their memory access patterns.” (this will be more true every year)

“You cannot be fast without knowing how data is touched.”
“Death by a Thousand Cuts”

Parallel processing typically requires synchronization.

“You cannot multi-thread without knowing how data is touched.”
"Death by a Thousand Cuts"

Parallel processing requires coherent program flow.

“You cannot **multi-thread** without knowing how data is touched.”
“Death by a Thousand Cuts”

```cpp
class Bot : public Enemy {
    ....
    vec3 m_position;
    ....
    float m_mod;
    ....
    float m_aimDirection;
    ....
    virtual void updateAim( vec3 target )
    {
        m_aimDirection = dot3( m_position, target ) * m_mod;
    }
    ....
```

Cached but not used
Cache miss
Branch cache miss

INFOMOV – Lecture 10 – “Data-Oriented Design”
“Death by a Thousand Cuts”

```c
void updateAims(
    float* aimDir,
    const AimingData* aim,
    vec3 target,
    uint count)
{
    for (uint i = 0; i < count; ++i)
    {
        aimDir[i] = dot3(aim->positions[i], target) * aim->mod[i];
    }
}
```
Algorithm Performance Factors

Estimating algorithm cost:

1. Algorithmic Complexity: \( O(N), O(N^2), O(N \log N), \ldots \)
2. Cyclomatic Complexity* (or: Conditional Complexity)
3. Amdahl’s Law / Work-Span Model
4. Cache Effectiveness

---

Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
Data Oriented Design*

Origin: low-level game development.

Core idea: *focus software design on CPU- and cache-aware data layout.*

Take into account:

- Cache line size
- Data alignment
- Data size
- Access patterns
- Data transformations

Strive for a simple, linear access pattern as much as possible.

Bad Access Patterns: Linked List

The Perfect LinkedList™:

```c
struct LLNode {
    LLNode* next;
    int value;
};

LLNode* nodes = new LLNode[...];
LLNode* pool = nodes;

for (int i = 0; i < ...; i++)
    nodes[i].next = &nodes[i + 1];

LLNode* NewNode( int value )
{
    LLNode* retval = pool;
    pool = pool->next;
    retval->value = value;
    return retval;
}

list = NewNode( -MAXINT );
list->next = NewNode( MAXINT );
list->next->next = 0;
```

nodes: 0 0 0 0 0 0 0 0 ... 

list: -10000 10000 0
Bad Access Patterns: Linked List

The Perfect LinkedList™, experiment:

Insert 25000 random values in the list so that we obtain a sorted sequence.

```c
for( int i = 0; i < COUNT; i++ )
{
  LLNode* node = NewNode( rand() & 8191);
  LLNode* iter = list;
  while (iter->next->value < node->value)
  {
    iter = iter->next;
  }
  node->next = iter->next;
  iter->next = node;
}
```
Bad Access Patterns: Linked List

KISS Array™:

data = new int[...];
memset( data, 0, ... * sizeof( int ) );
data[0] = -10000;
data[1] = 10000;
N = 2;

for( int i = 0; i < COUNT; i++ )
{
    int pos = 1, value = rand() & 8191;
    while (data[pos] < value) pos++;
    memcpy( data + pos + 1, data + pos, (N - pos + 1) * sizeof( int ) );
    data[pos] = value, N++;
}
for( int i = 0; i < COUNT; i++ )
{
    LLNode* node = NewNode( rand() & 8191);
    LLNode* iter = list;
    while (iter->next->value < node->value)
    {
        iter = iter->next;
    }
    node->next = iter->next;
    iter->next = node;
}

for( int i = 0; i < COUNT; i++ )
{
    int pos = 1, value = rand() & 8191;
    while (data[pos] < value) pos++;
    memcpy( data + pos + 1, data + pos,
            (N - pos + 1) * sizeof( int ) );
    data[pos] = value, N++;
}
Bad Access Patterns: Linked List*

Inserting elements in an array by shifting the remainder of the array is *significantly faster* than using an optimized linked list.

**Why?**

- Finding the location in the array: pure linear access
- Shifting the remainder: pure linear access.

➔ Even though the amount of transferred memory is huge, this approach wins.

*: Also see: Nathan Reed, Data Oriented Hash Table, 2015. [http://www.reedbeta.com/blog/data-oriented-hash-table](http://www.reedbeta.com/blog/data-oriented-hash-table)
Bad Access Patterns: Octree
Bad Access Patterns: Octree

Query: find the color of a voxel visible through pixel \((x,y)\).
Operation: ‘3DDDA’ (basically: Bresenham).
Data layout:

Color data: 32-bit (ARGB).
Bad Access Patterns: Octree

Alternative layout:

1. Tree 1: occlusion (1 bit per voxel);
2. Tree 2: color information (32 bits per voxel).

Use tree 1 to find the voxel you are looking for. Lookup the correct voxel (incurring a single cache miss) in tree 2.

Caching in tree 1:

- A cache line holds 64*8=512 voxels
- Accessing the root gets several levels in L1 cache
Bad Access Patterns: Textures in a Ray Tracer

Typical process for tracing a ray:

- Traverse a tree (multiple kilobytes)
- Intersect triangles in the leaf nodes (quite a few bytes)
- If a hit is found, fetch texture.

_This is almost always a cache miss._
Bad Access Patterns: Textures in a Ray Tracer

We suffer the cache miss *twice*:

- Once for the texture;
- Once for the normal map.

Note: both values are 32-bit.
Bad Access Patterns: Textures in a Ray Tracer

Interleaved texture / normal:

- One value now becomes 64-bit and contains the normal and the color.
- We still suffer a cache miss –
- But only once.
Previously in INFOMOV

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass;
};
// size: 28 bytes
```

Better:

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass, dummy;
};
// size: 32 bytes
```
Previously in INFOMOV

```c
struct Particle
{
    float x, y, z;
    int mass;
};

Particle particle[512];
```

**AOS**

```c
union { float x[512]; __m128 x4[128]; }
union { float y[512]; __m128 y4[128]; }
union { float z[512]; __m128 z4[128]; }
union { int mass[512]; __m128i mass4[128]; }
```

**SOA**

```c
AOS structure of arrays
```
Previously in INFOMOV

**Method:**

\[
X = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}
\]

\[
Y = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}
\]

\[
M = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}
\]
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
ECS

Entity – Component - System

OOP:

Actor
vec3 position;
uint spriteId;
bool Tick();

Rock
uint particles;
int angle;

Wind
...

Boss
...

---

```
Actor
vec3 position;
uint spriteId;
bool Tick();

Rock
uint particles;
int angle;

Wind
...

Boss
...
```
ECS

Entity – Component - System

OOP:

Actor

void position;
uint spriteId;
bool Tick();

Rock
[sprite]
[position]
[velocity]

Wind
[particles]

Boss
[sprite]
[position]
[velocity]
[sound effect]

ECS:

Entities: Rock, Wind, Boss, ...
Components: sprite, position, velocity, sound effect, ...
Systems: UpdatePosition, Collide, UpdatePath, ...

Entities: quite similar to objects, but without methods.
(an entity is a collection of components)

Components: properties or compound properties.

Systems: functions that operate on groups of entities
with the same components.

Unlike OOP, ECS separates data from behavior.
ECS

Entity – Component - System

ECS improves cache utilization:

- Systems are executed for all relevant entities, maximizing L1I$ locality.
- Input data for systems (components) is read sequentially.
- Component data is not interleaved as in OOP.

ECS encourages SIMD and GPU use:

- Systems are fundamentally data-parallel functions.
- Data is already in SoA format.
Today’s Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
OO = Evil, DO = Good?

10% of your code runs 90% of the time. DO is good for this 10%.

For all other code, please:

- Use STL
- Apply OO
- Program in C#
- Use event handling
- Check return values
- Focus on productivity
DATA-ORIENTED DESIGN AND C++

Presenter: Mike Acton

https://www.youtube.com/watch?v=rX0ItVEVjHc
Data-Oriented Design

Richard Fabian

- Contents
- Data-Oriented Design
  - It's all about the data
  - Data is not the problem domain
  - Data and statistics
  - Data can change
  - How is data formed?
  - The framework
  - Conclusions and takeaways

http://www.dataorienteddesign.com/dodbook/
Data Oriented Design Resources

A curated list of awesome data oriented design resources.

Feel free to contribute by sending PR!

- **Data Oriented Design**
  - Presentations
  - Blog Posts
  - Videos
  - Other
  - Code Examples

### Presentations

- A Step Towards Data Orientation (2010) - Johan Torp
- Introduction To Data Oriented Design (2010) - DICE
- Memory Optimization (2003) - Christer Ericson
- Practical Examples In Data Oriented Design (2013) - Niklas Frykholm
- Three Big Lies (2008) - Mike Acton
- Typical C++ Bullshit (2008) - Mike Acton
- Data-Oriented Design and C++ (2014) - Mike Acton
- Entity Component Systems & Data Oriented Design (2018) - Aras Pranckevičius

### Blog Posts

- [https://github.com/dbartolini/data-oriented-design](https://github.com/dbartolini/data-oriented-design)
- Adventures in data-oriented design – Part 1: Mesh data (2011) - Stefan Reinalter
Adventures in data-oriented design – Part 1: Mesh data

Let's face it, performance on modern processors (be it PCs, consoles or mobiles) is mostly governed by memory access patterns. Still, data-oriented design is considered something new and novel, and only slowly creeps into programmers' brains, and this really needs to change. Having co-workers fix your code and improving its performance really is no excuse for writing crappy code (from a performance point-of-view) in the first place.

This post is the first in an ongoing series about how certain conventions in the Molecule Engine are done in a data-oriented fashion, while still making use of OOP concepts. A common misconception about data-oriented design is that it is "C-like", and "not OOP", and therefore less maintainable – but that need not be the case. The concrete example we are going to look at today is how to organize your mesh data, but let's start with the pre-requirements first.

https://blog.molecular-matters.com/2013/05/02/adventures-in-data-oriented-design-part-3a-ownership/
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Entity Component System
- DOD or OO?
END of “Data-Oriented Design”
next lecture: “GUEST LECTURE”