Welcome!
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide  (*to Pleasing the Cache*)
Recap

Refresher:

Three types of cache:

- Fully associative
- Direct mapped
- N-set associative

In an N-set associative cache, each memory address can be stored in N slots.

Example:

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

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

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

32-bit address
Recap

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

32-bit address

Examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000001234</td>
<td>0001</td>
<td>001000</td>
<td>110100</td>
</tr>
<tr>
<td>0x000008234</td>
<td>1000</td>
<td>001000</td>
<td>110100</td>
</tr>
<tr>
<td>0x000006234</td>
<td>0110</td>
<td>001000</td>
<td>110100</td>
</tr>
<tr>
<td>0x0000A234</td>
<td>1010</td>
<td>001000</td>
<td>110100</td>
</tr>
<tr>
<td>0x0000A240</td>
<td>1010</td>
<td>001001</td>
<td>000000</td>
</tr>
<tr>
<td>0x0000F234</td>
<td>1111</td>
<td>001000</td>
<td>110100</td>
</tr>
</tbody>
</table>
Recap

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

32-bit address

Examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00001234</td>
<td>0001</td>
<td>00100</td>
<td>110100</td>
</tr>
<tr>
<td>0x00008234</td>
<td>1000</td>
<td>00100</td>
<td>110100</td>
</tr>
<tr>
<td>0x00006234</td>
<td>0110</td>
<td>00100</td>
<td>110100</td>
</tr>
<tr>
<td>0x0000A234</td>
<td>1010</td>
<td>00100</td>
<td>110100</td>
</tr>
<tr>
<td>0x0000A240</td>
<td>1010</td>
<td>00100</td>
<td>000000</td>
</tr>
<tr>
<td>0x0000F234</td>
<td>1111</td>
<td>00100</td>
<td>110100</td>
</tr>
</tbody>
</table>
Recap

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

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
Recap

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

Theoretical consequence:

- Address 0, 4096, 8192, ... map to the same set (which holds max. 8 addresses)
- consider `int value[1024][1024]`:
  - `value[0,1,2...][x]` map to the same set
  - querying this array vertically:
    - will quickly result in evictions
    - will use only 512 bytes of your cache
**Recap**

**64 bytes per cache line**

Theoretical consequence:

- If address \( X \) is pulled into the cache, so is \((X+1,...,X+63)\).

Example*:

```java
int arr = new int[64 * 1024 * 1024];
// loop 1
for (int i = 0; i < 64 * 1024 * 1024; i++) arr[i] *= 3;
// loop 2
for (int i = 0; i < 64 * 1024 * 1024; i += 16) arr[i] *= 3;
```

Which one takes longer to execute?

Theoretical consequence:

- If address $X$ is removed from cache, so is ($X+1$....$X+63$).
- If the object you’re querying straddles the cache line boundary, you may suffer not one but two cache misses.

Example:

```c
struct Pixel { float r, g, b; }; // 12 bytes
Pixel screen[768][1024];
```

Assuming pixel (0,0) is aligned to a cache line boundary, the offsets in memory of pixels (0,1..5) are 12, 24, 36, 48, 60, ... . Walking column 5 will be very expensive.
Recap

Considering the Cache

- Size
- Cache line size and alignment
- Aliasing
- Sharing
- Access patterns
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide *(to Pleasing the Cache)*
Data Locality

Why do Caches Work?

1. Because we tend to reuse data.
2. Because we tend to work on a small subset of our data.
3. Because we tend to operate on data in patterns.
Data Locality

Reusing data

- Very short term: variable ‘i’ being used intensively in a loop ➔ register
- Short term: lookup table for square roots being used on every input element ➔ L1 cache
- Mid-term: particles being updated every frame ➔ L2, L3 cache
- Long term: sound effect being played ~ once a minute ➔ RAM
- Very long term: playing the same CD every night ➔ disk
Data Locality

Reusing data

Ideal pattern:
- load data once, operate on it, discard.

Typical pattern:
- operate on data using algorithm 1, then using algorithm 2, ...

Note:
GPUs typically follow the ideal pattern.

*(more on that later)*
Data Locality

Reusing data

Ideal pattern:
- load data sequentially.

Typical pattern:
- whatever the algorithm dictates.
Data Locality

Example: rotozooming
Data Locality

Example: rotozooming
Data Locality

Example: rotozooming

Improving data locality: z-order / Morton curve

Method:

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

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

\[
M = 1101101000111001110011111001
\]
Data Locality

Wikipedia:

**Temporal Locality** – “If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.”

**Spatial Locality** – “If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.”

Data Locality

How do we increase data locality?

**Linear access** – Sometimes as simple as swapping for loops *

**Tiling** – Example of working on a small subset of the data at a time.

**Streaming** – Operate on/with data until done.

**Reducing data size** – Smaller things are closer together.

* For an elaborate example see [https://www.cs.duke.edu/courses/cps104/spring11/lects/19-cache-sw2.pdf](https://www.cs.duke.edu/courses/cps104/spring11/lects/19-cache-sw2.pdf)
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide (to Pleasing the Cache)
Alignment

Cache line size and data alignment

What is wrong with this struct?

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass;
};
```

// size: 28 bytes

Two particles will fit in a cache line (taking up 56 bytes).

The next particle will be in two cache lines.

Better:

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass, dummy;
};
```

// size: 32 bytes

Note:

As soon as we read any field from a particle, the other fields are guaranteed to be in L1 cache.

If you update x, y and z in one loop, and vx, vy, vz in a second loop, it is better to merge the two loops.
Alignment

Cache line size and data alignment

What is wrong with this allocation?

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass, dummy;
};
// size: 32 bytes
Particle particles[512];
```

Although two particles will fit in a cache line, we have no guarantee that the address of the first particle is a multiple of 64.

Note:

Is it bad if particles straddle a cache line boundary?

Not necessarily: if we read the array sequentially, we sometimes get 2, but sometimes 0 cache misses.

For random access, this is not a good idea.
Cache line size and data alignment

Controlling the location in memory of arrays:

An address that is dividable by 64 has its lowest 6 bits set to zero. In hex: all addresses ending with 40, 80 and C0.

Enforcing this:

Particle* particles = __aligned_malloc(512 * sizeof(Particle), 64);

Or:

__declspec(align(64)) struct Particle { ... };

Alignment
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide (to Pleasing the Cache)
False Sharing

Multiple Cores using Caches

Two cores can hold copies of the same data.

Not as unlikely as you may think – Example:

```java
byte data = new byte[COUNT];
for (int i = 0; i < COUNT; i++)
    data[i] = rand() % 256;
// count byte values
int counter[256];
for (int i = 0; i < COUNT; i++)
    counter[byteArray[i]]++;
```
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide  
  (to Pleasing the Cache)
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

1. Keep your data in registers

Use fewer variables

Limit the scope of your variables

Pack multiple values in a single variable

Use floats and ints (they use different registers)

Compile for 64-bit (more registers)

Arrays will never go in registers
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

2. Keep your data local

Read sequentially
Keep data small
Use tiling / Morton order
Fetch data once, work until done (streaming)
Reuse memory locations
3. Respect cache line boundaries

Use padding if needed
Don’t pad for sequential access
Use aligned malloc / __declspec align
Assume 64-byte cache lines
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

4. Advanced tricks

Prefetch
Use a prefetch thread
Use streaming writes
Separate mutable / immutable data
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

5. Be informed

Use the profiler!
Today's Agenda:

- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide  *(to Pleasing the Cache)*
END of “Caching (2)"

next lecture: “SIMD (1)”