/INFOMOV/

Optimization & Vectorization

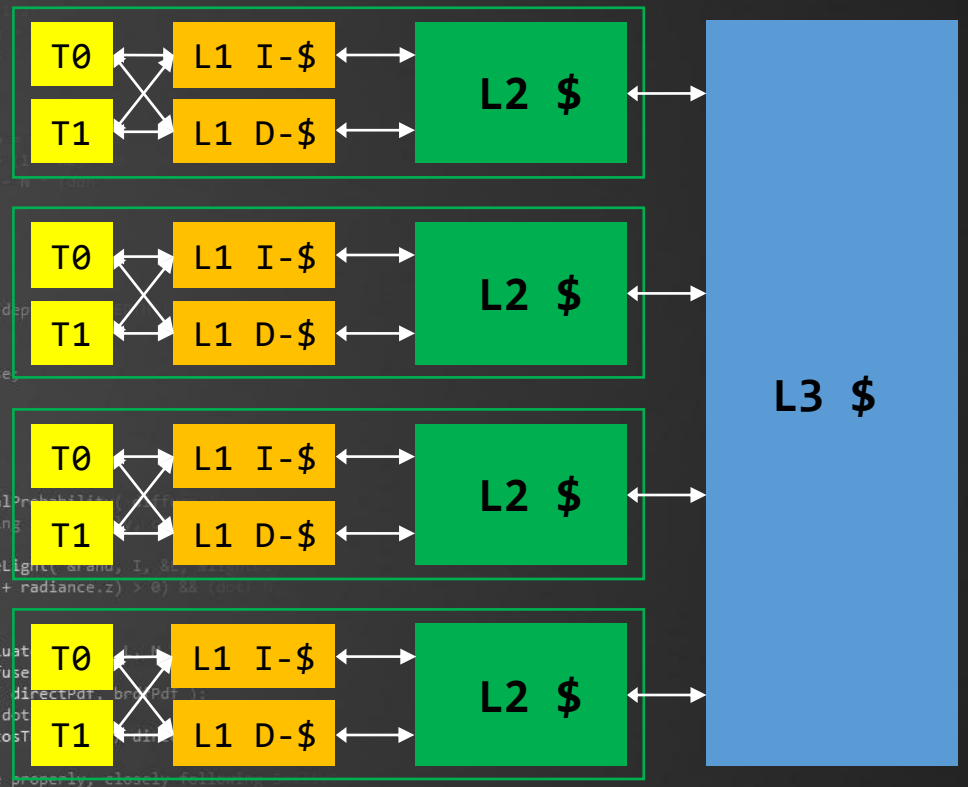
J. Bikker - April - June 2024 - Lecture 6: “Caching (2)”

Welcome!

Today’s Agenda:

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

Recap

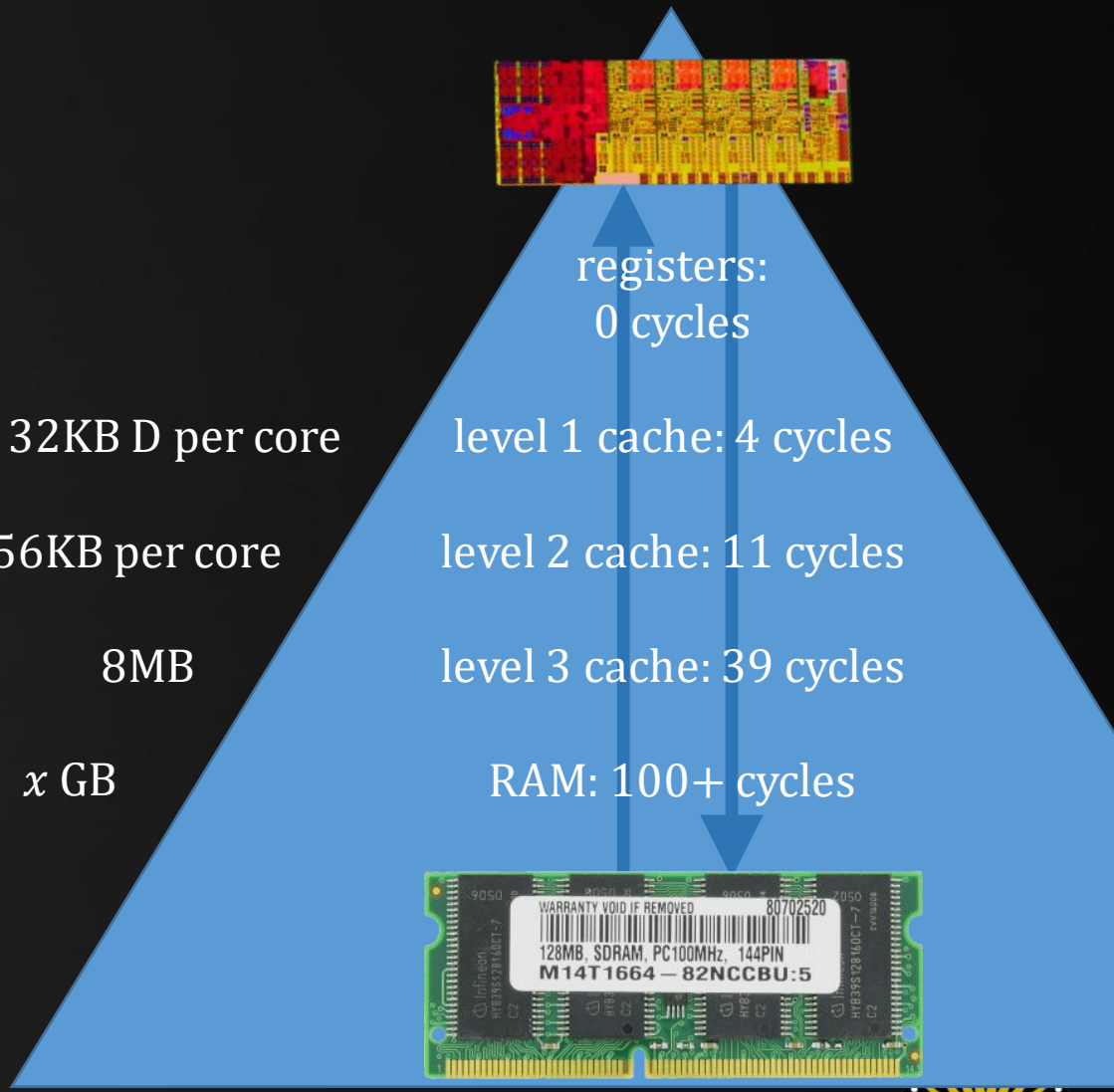


32KB I / 32KB D per core

256KB per core

8MB

x GB



Recap

Three types of cache

- Fully associative
- Direct mapped
- N-set associative

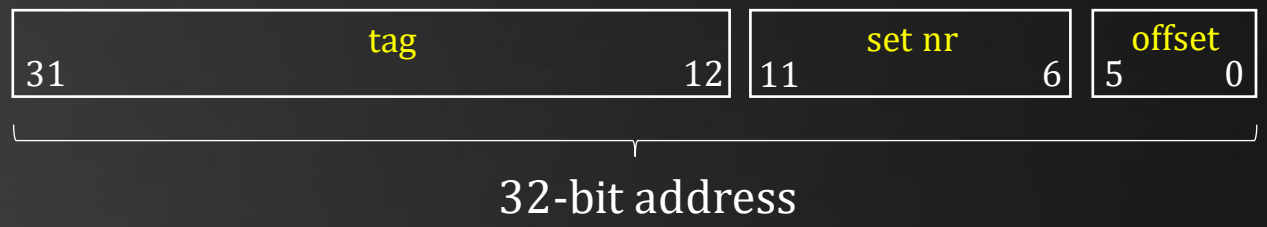
In an *N-set associative cache*, each memory address can be stored in N slots.

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



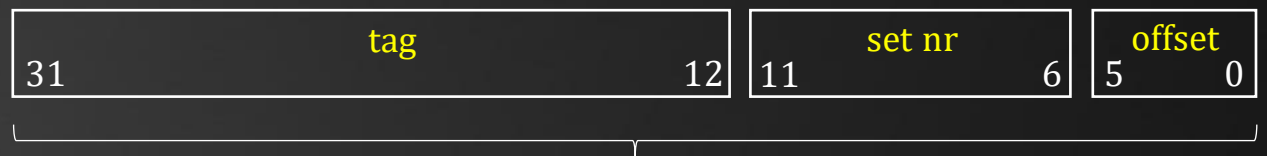
Fetching a byte at address `a`:

offset = a & 63
set = (a >> 6) & 63
tag = a >> 12, range: 0 .. 2^{20}-1

return cache[tag][0..7].data[offset]

Recap

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



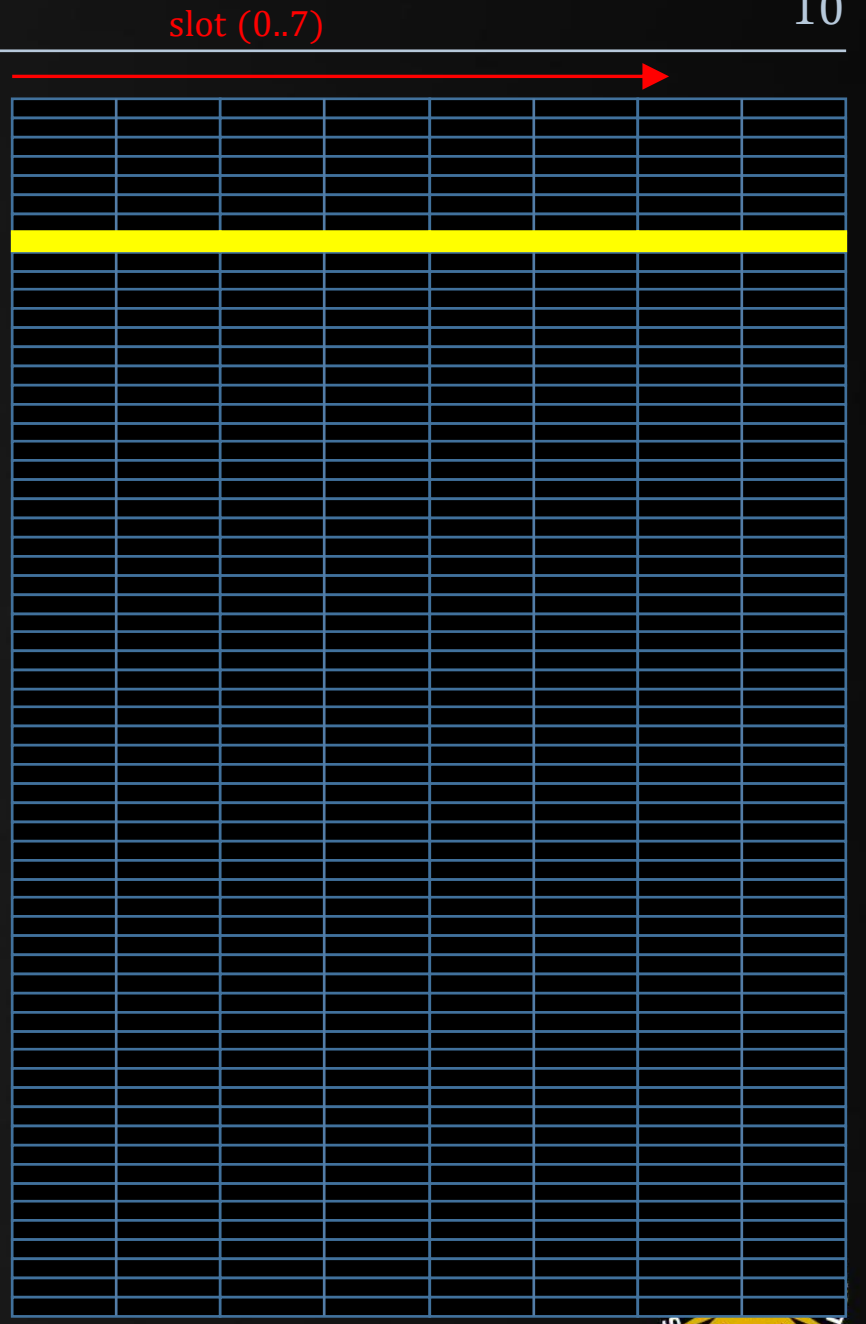
32-bit address

Examples:

set: 001000b = 8

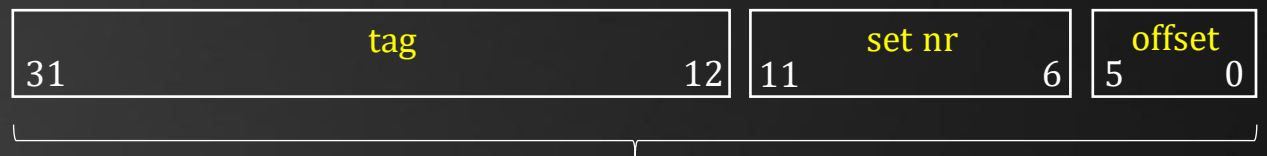
offset: 110100b = 52

```
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

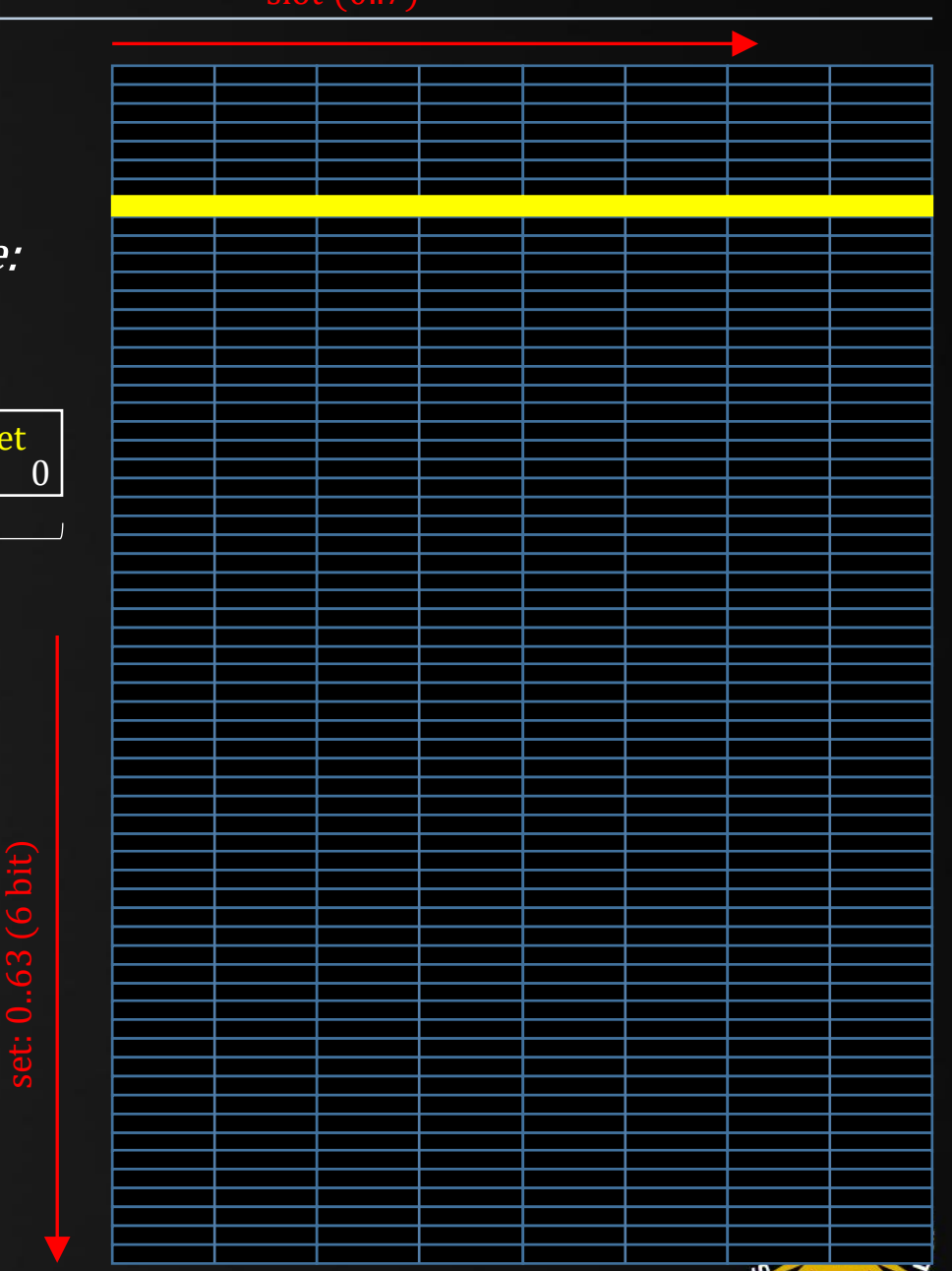
32KB, 8-way set-associative, 64 bytes per cache line:
64 sets of 512 bytes



32-bit address

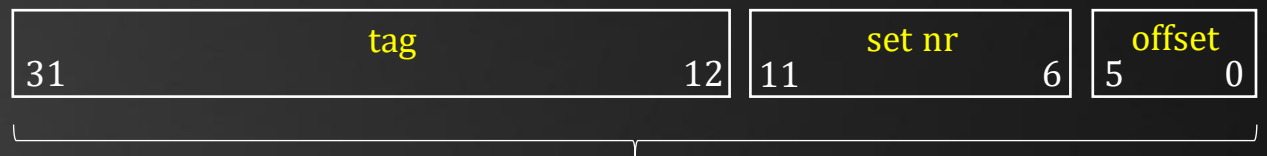
Examples:

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



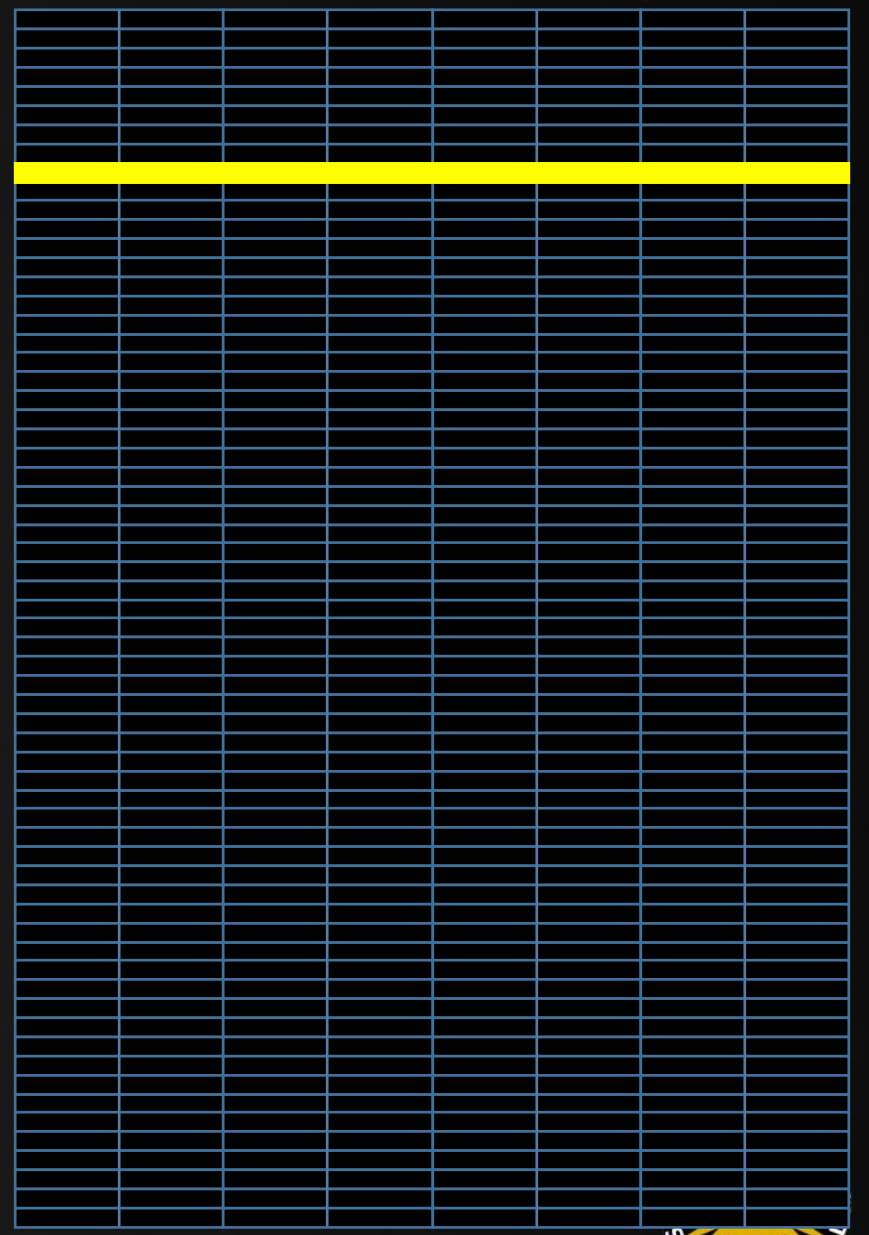
Recap

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



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
```



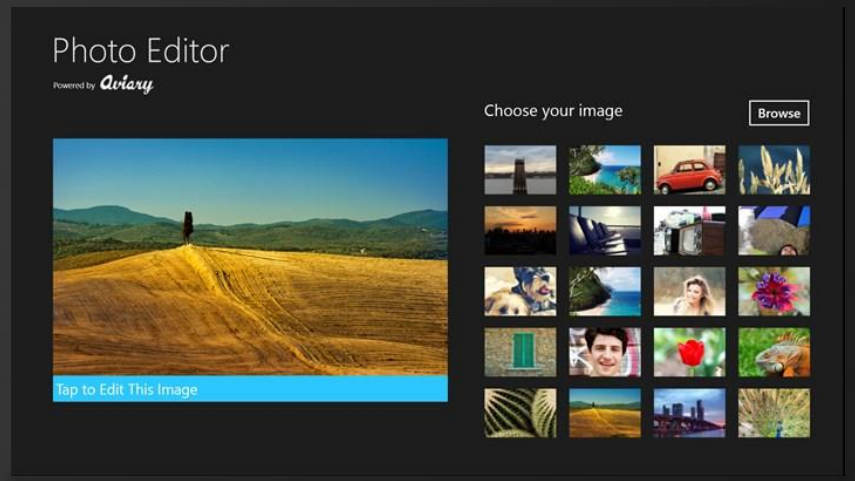
Today’s Agenda:

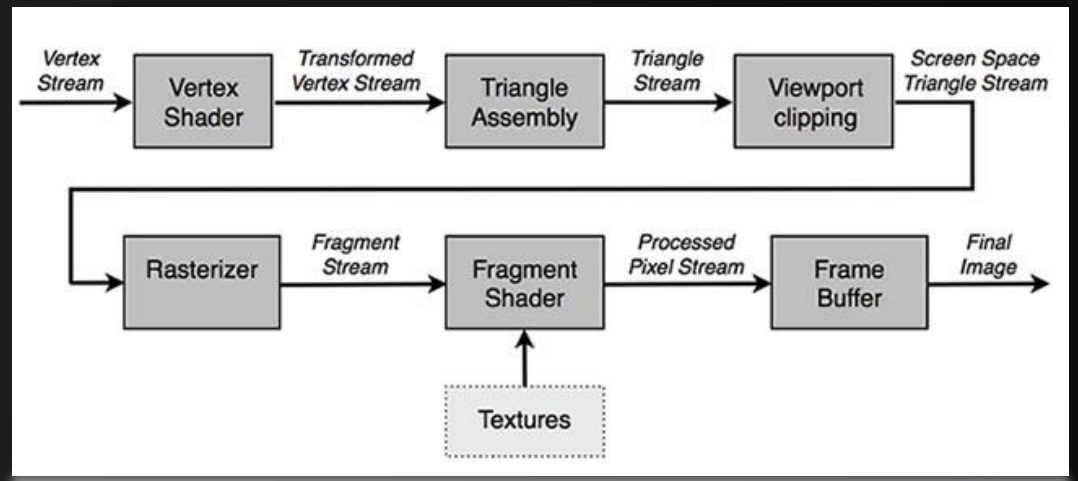
- Recap
- Data Locality
- Alignment
- False Sharing
- Experiments
- A Handy Guide *(to Pleasing the Cache)*

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 game disk every night ➔ disk

Data Locality

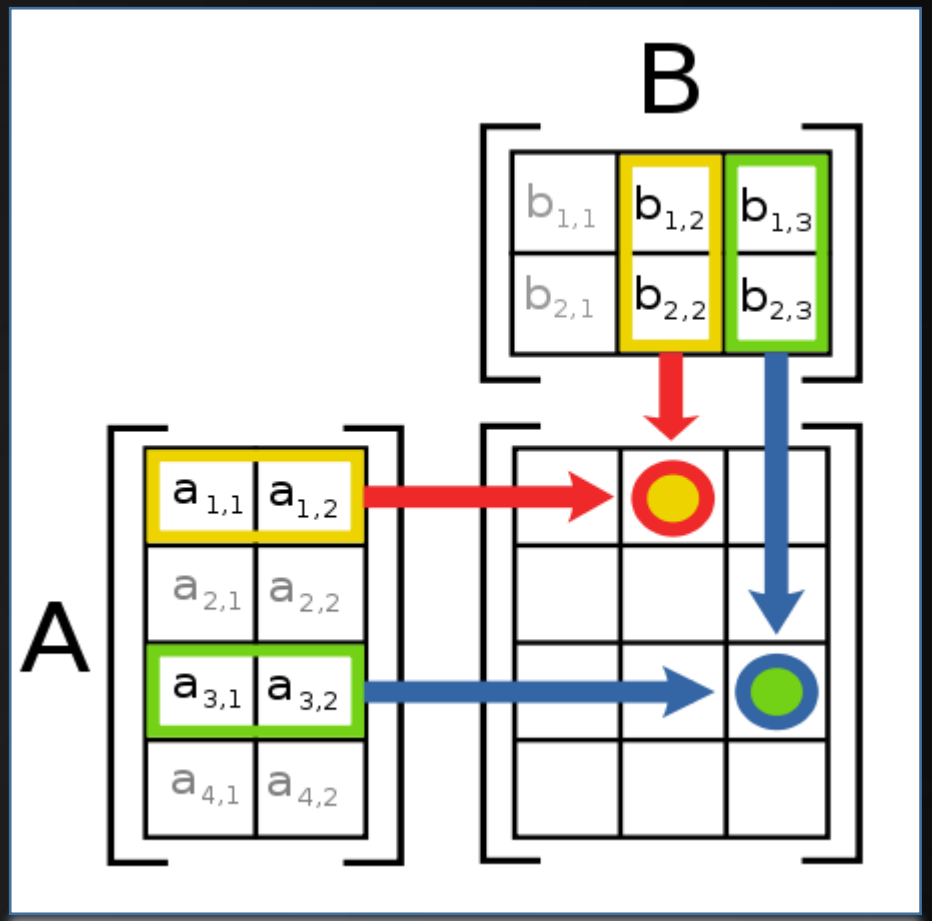
Reusing data

Ideal pattern:

- load data sequentially.

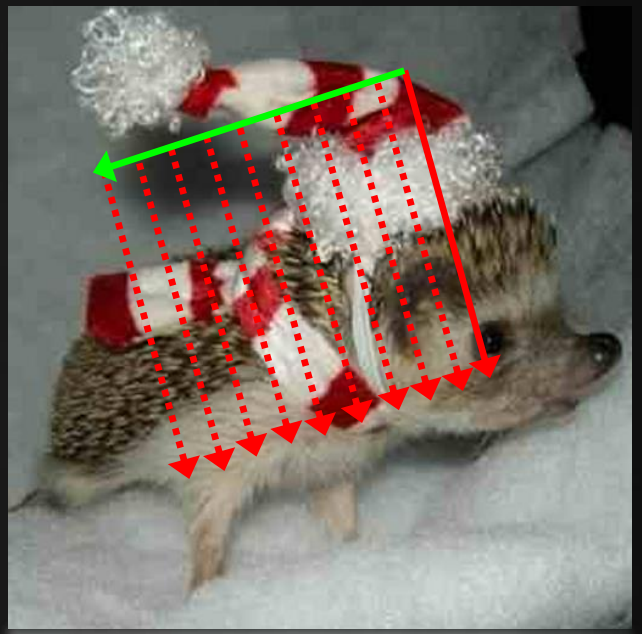
Typical pattern:

- *whatever the algorithm dictates.*



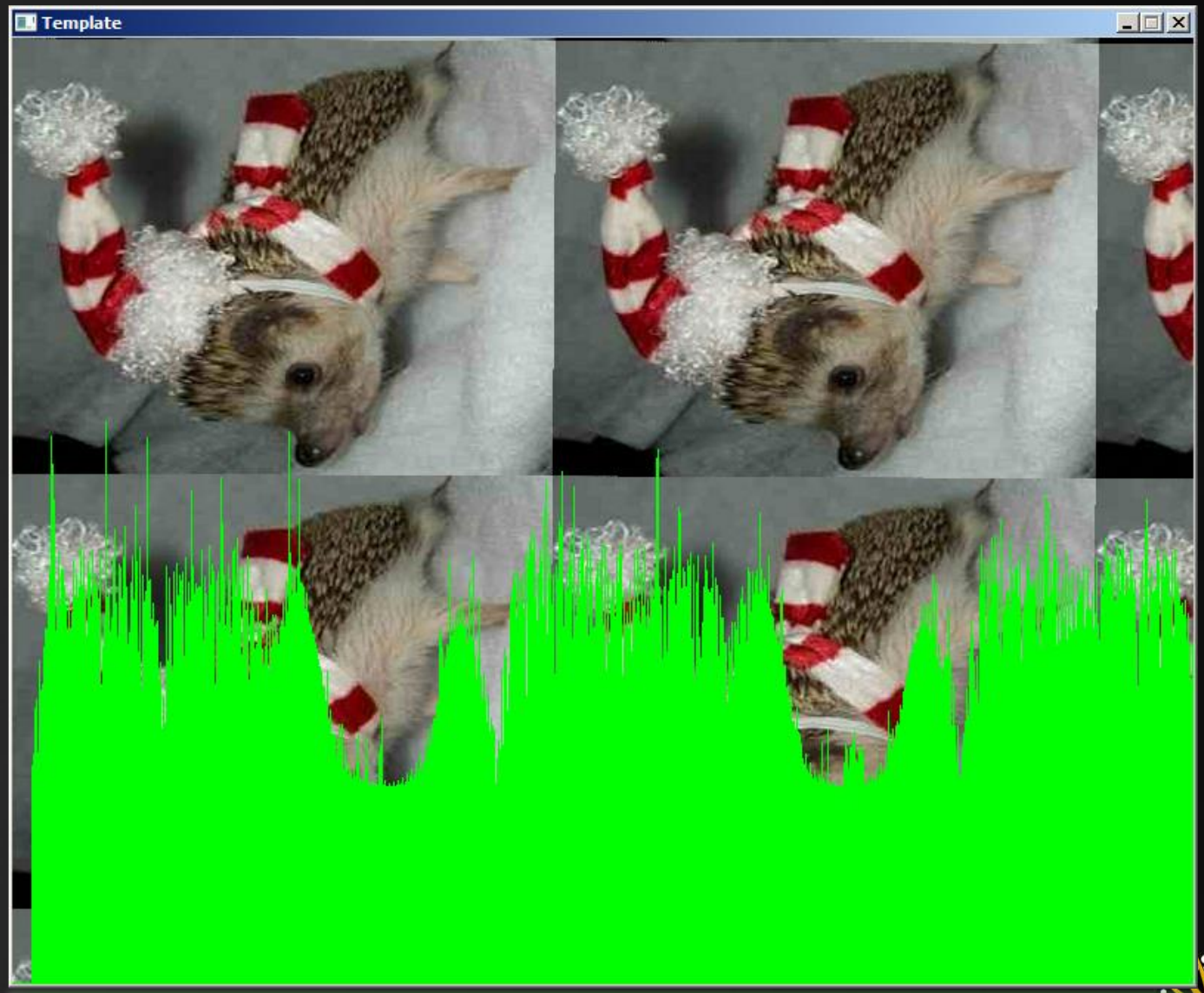
Data Locality

Example: rotozooming



Data Locality

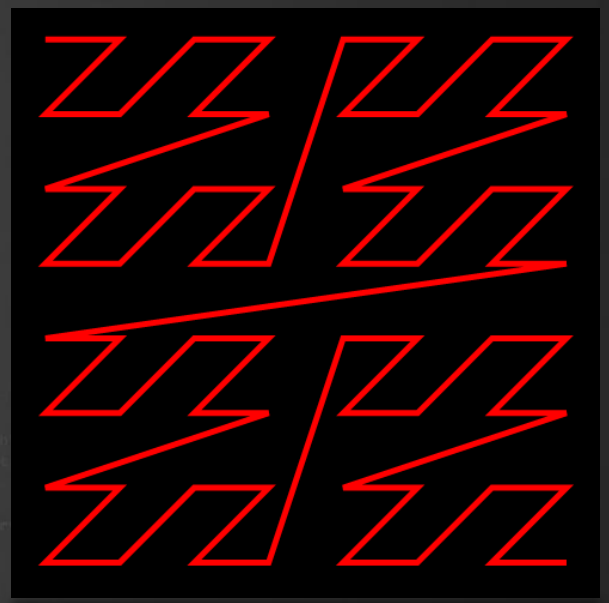
Example: rotozooming



Data Locality

Example: rotozooming

Improving data locality: z-order / Morton curve



Method:

```
X =  1 1 0 0 0 1 0 1 1 0 1 1 0 1

Y = 1 0 1 1 0 1 1 0 1 0 1 1 1 0

-------------------------------

address = 11011010001110011100111111001
```

Data Locality

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.”

* More info: http://gameprogrammingpatterns.com/data-locality.html

Data Locality

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.

How do trees/linked lists/hash tables fit into this?

* For an elaborate example see https://www.cs.duke.edu/courses/cps104/spring11/lects/19-cache-sw2.pdf

Today’s Agenda:

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

Alignment

Cache line size and data alignment

What is wrong with this struct?

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

Better:

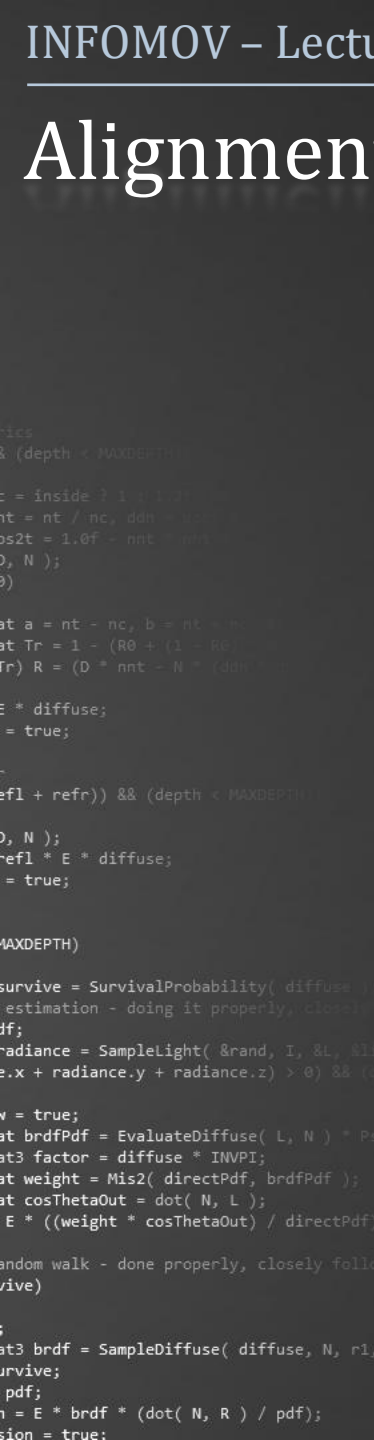
```
struct Particle
{
   float x, y, z;
   float vx, vy, vz;
   float mass, dummy;
};
// size: 32 bytes
```

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.

Two particles will fit in a cache line (taking up 56 bytes).
The next particle will be in *two* cache lines.



Alignment

Cache line size and data alignment

What is wrong with this allocation?

```
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.

Alignment

Cache line size and data alignment

Controlling the location in memory of arrays:

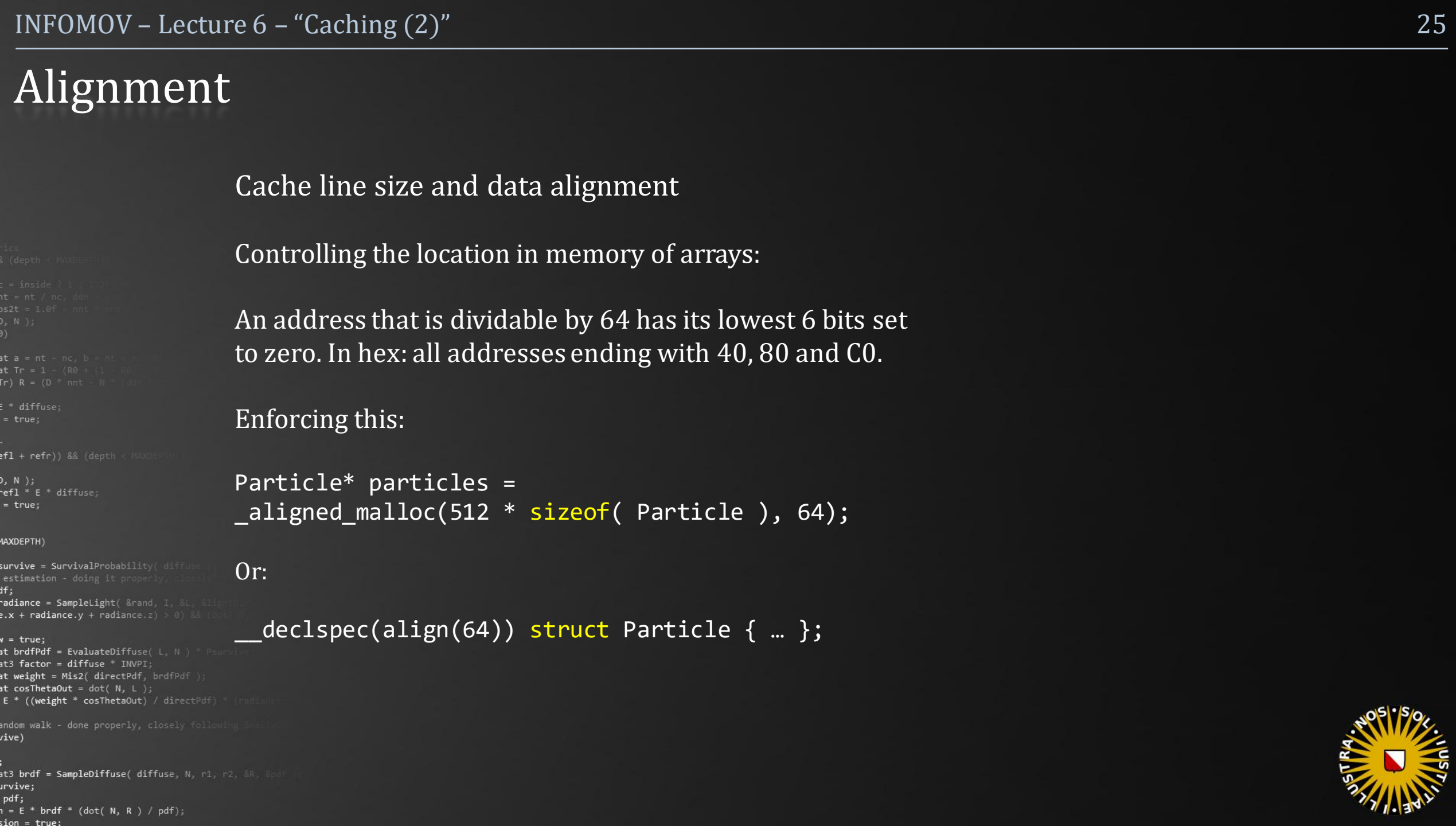
An address that is dividable by 64 has its lowest 6 bits set to zero. In hex: all addresses ending with 40, 80 and C0.

Enforcing this:

```
Particle* particles =
_aligned_malloc(512 * sizeof( Particle ), 64);
```

Or:

```
__declspec(align(64)) struct Particle { … };
```



Today’s Agenda:

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

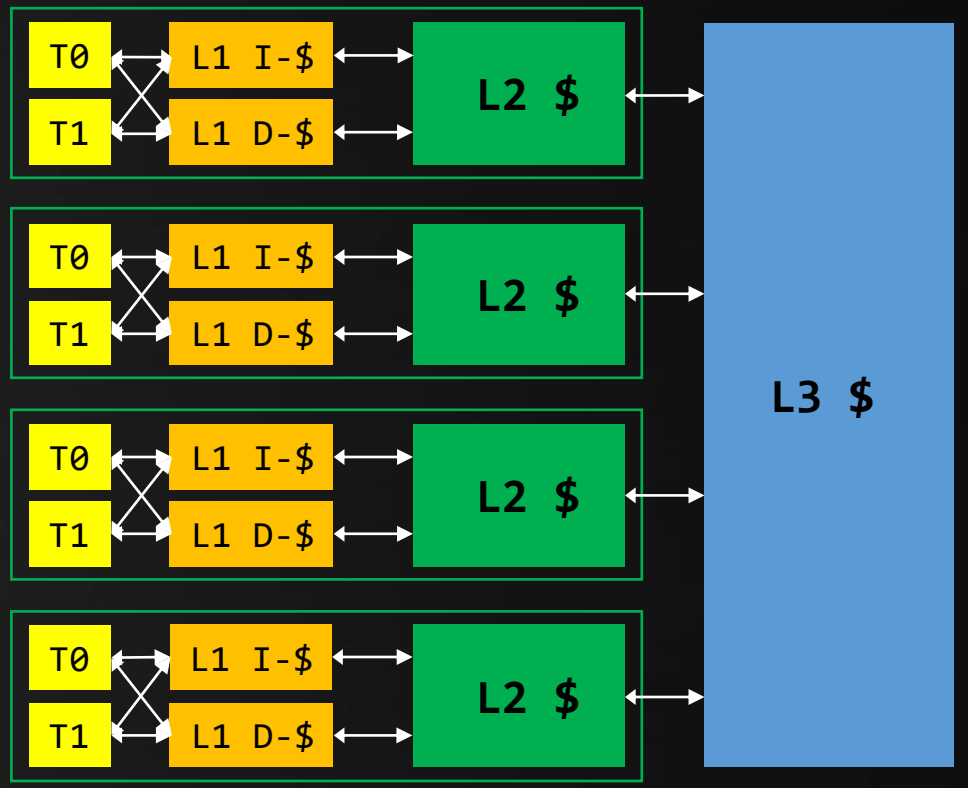
False Sharing

Multiple Cores using Caches

Two cores can hold copies of the same data.

Not as unlikely as you may think – Example:

```
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:

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

Experiments

Cache Size

Basic test:

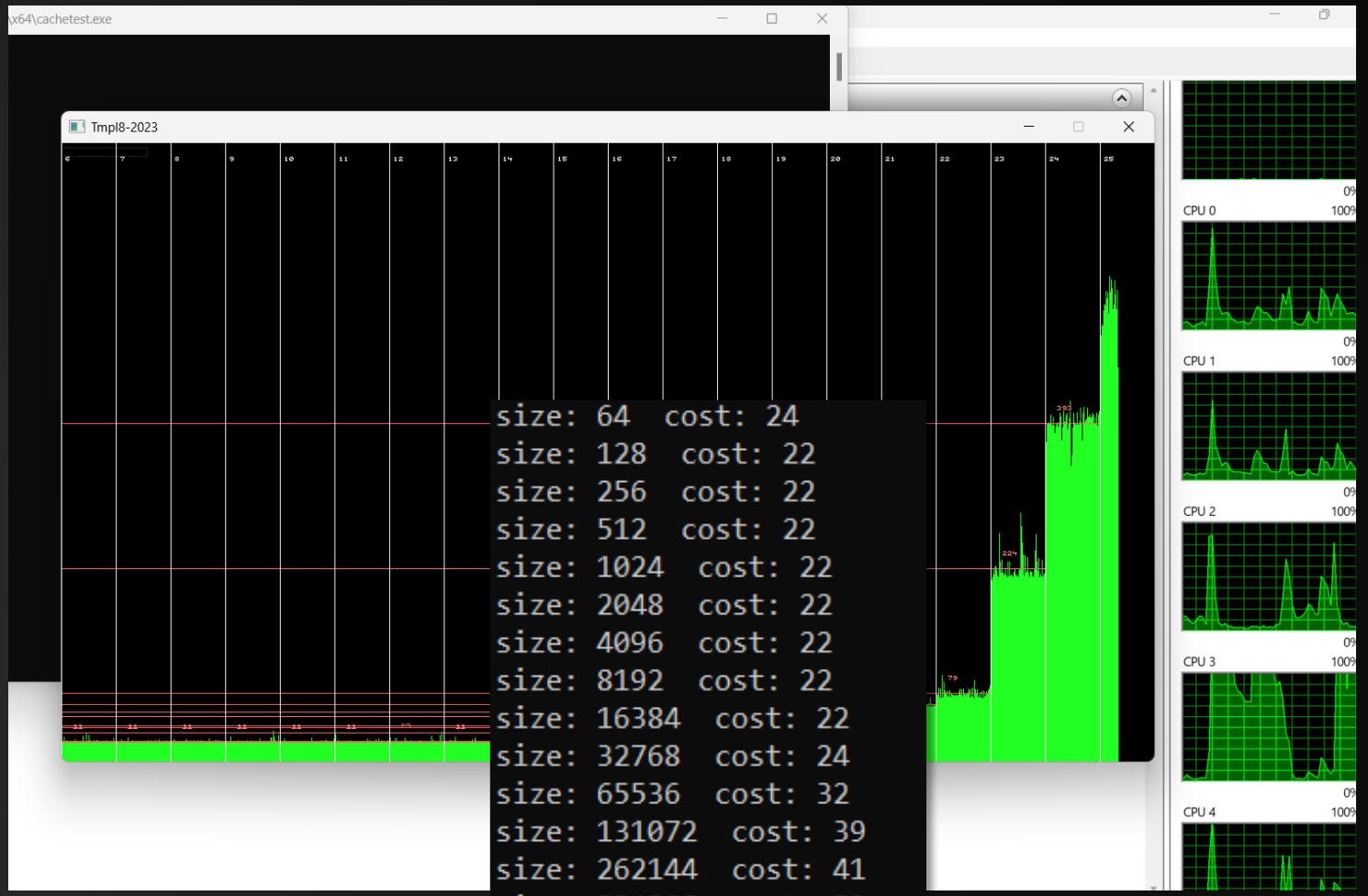
- random access
- increasing data size

Additional test:

- payload size
- data type

Additional test:

- straddling cache lines



Experiments

Cache Size

Basic test:

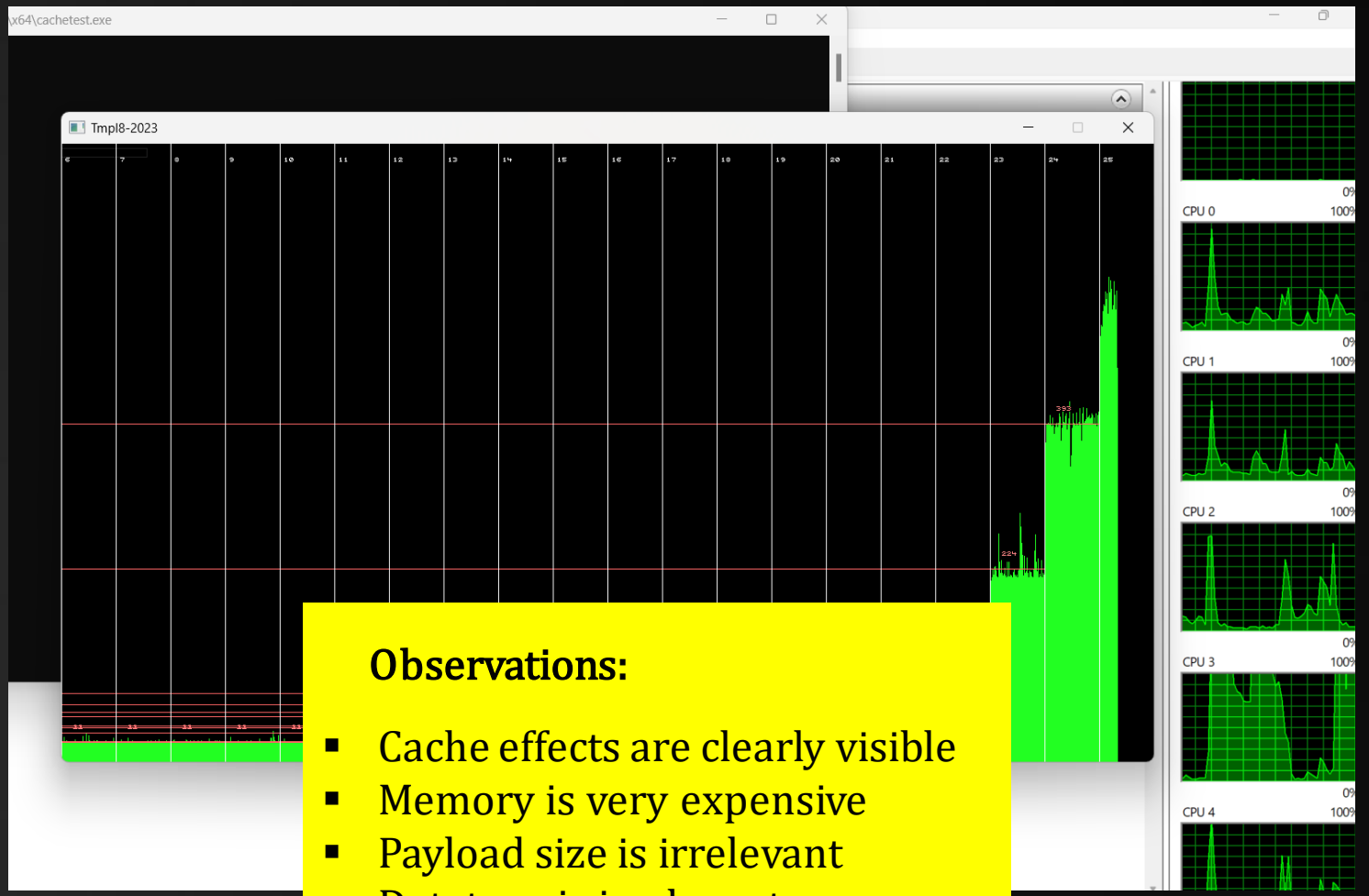
- random access
- increasing data size

Additional test:

- payload size
- data type

Additional test:

- straddling cache lines



Observations:

- Cache effects are clearly visible
- Memory is very expensive
- Payload size is irrelevant
- Datatype is irrelevant
- Straddling is irrelevant (!)
- …

Experiments

Linear Data Access (1)

Horizontal versus vertical…

Experiment:

Getting it to work

- suspiciously fast…
- …until we include i in the result. ☺

Changing access pattern

- running average
- swapping loops

```
static Payload table[1024][1024]; // 4MB
Timer t;
Payload sum;
for( int i = 0; i < 100; i++ )
{
    for (int x = 0; x < 1024; x++ )
    {
        for( int y = 0; y < 1024; y++ )
        {
            sum += table[y][x];
        }
    }
}
float elapsed = t.elapsed() * 1000;
```

Experiments

Linear Data Access

Horizontal versus vertical…

Experiment:

Getting it to work

- suspiciously fast…
- …until we include i in the res

Changing access pattern

- running average
- swapping loops

Observations:

- Compiler is clever
- Linear access matters!
- …

```
static Payload table[1024][1024]; // 4MB
Timer t;
Payload sum;
for( int i = 0; i < 100; i++ )
{
    for (int x = 0; x < 1024; x++ )
    {
        for (int y = 0; y < 1024; y++ )
        {
            sum += table[y][x];
        }
    }
} * 1000;
```

```
        for (int x = 0; x < 1024; x++)
0000000140001090  mov         edx,400h
0000000140001095  nop         word ptr [rax+rax]
    {
            for (int y = 0; y < 1024; y++)
00000001400010A0  imul        eax,dword ptr [rcx],64h
00000001400010A3  add         rcx,4
00000001400010A7  add         edi,eax
00000001400010A9  sub         rdx,1
00000001400010AD  jne         Tmpl8::Game::Tick+40h (01400010A0h)
    Payload sum;
    for (int i = 0; i < 100; i++)
00000001400010AF  sub         r8,1
00000001400010B3  jne         Tmpl8::Game::Tick+30h (0140001090h)
            {
                sum += table[y][x];
            }
        }
    }
```

Experiments

False Sharing

Experiment:

- Counting, single-threaded
- Counting, multi-threaded

```
static int counters[256];
Timer t;
#pragma omp parallel for schedule(dynamic)
for( int chunk = 0; chunk < 16; chunk++ )
{
    Payload* chunkStart = mem + chunk * 1024 * 1024;
    for( int i = 0; i < 1024 * 1024; i++ )
    {
        Payload value = chunkStart[i];
        counters[value]++;
    }
}
float elapsed = t.elapsed() * 1000;
```

Observations:

- False sharing is no joke!
- Use per-thread counters.

https://cdrdv2-public.intel.com/671363/vtune-tutorial-linux-identifying-false-sharing.pdf

Common pitfall:

- Random number generator seed
- Any case of ‘why is this not scaling’.
- Solution: Use thread_local keyword.

Today’s Agenda:

- Recap
- Data Locality
- Alignment
- False Sharing
- Experiments
- 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
Unions technically can 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

Easy Steps

How to Please the Cache

Or: “how to evade RAM”

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 (theoretical…)
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:

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

/INFOMOV/

END of “Caching (2)”

next lecture: “GPGPU (1)”