/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 4: “Caching (2)”

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

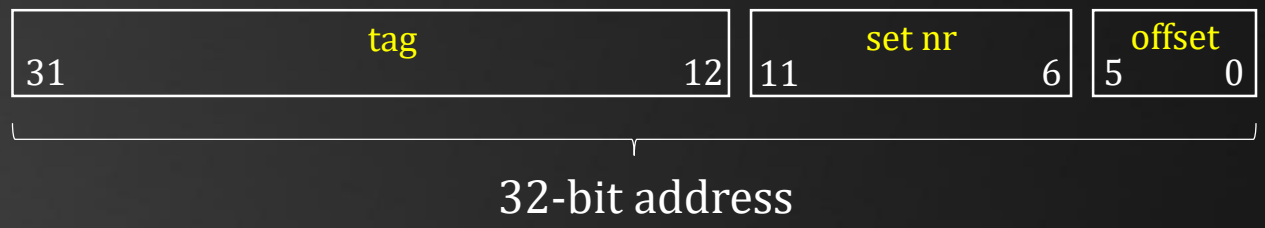
In an N-set associative cache, each memory address can be stored in N slots.

Example:

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

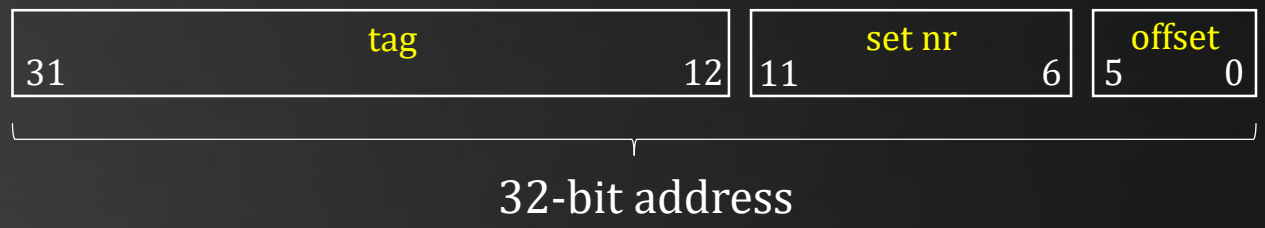
Recap

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



Recap

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

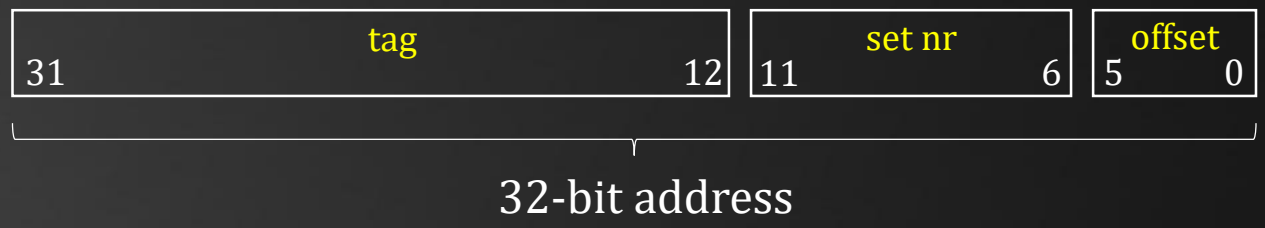


Examples:

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

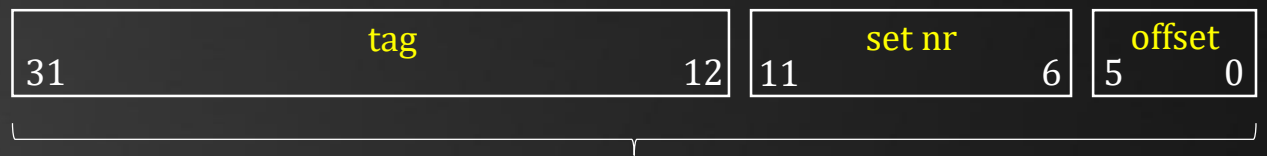


Examples:

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



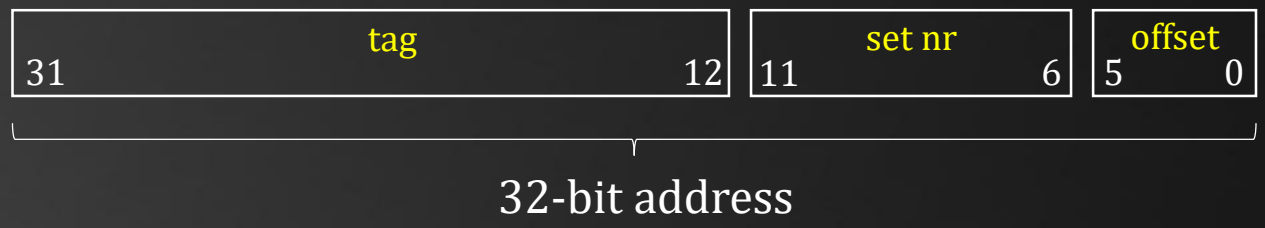
32-bit address

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



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*:

```
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?

*: http://igoro.com/archive/gallery-of-processor-cache-effects

Recap

64 bytes per cache line

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:

```
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

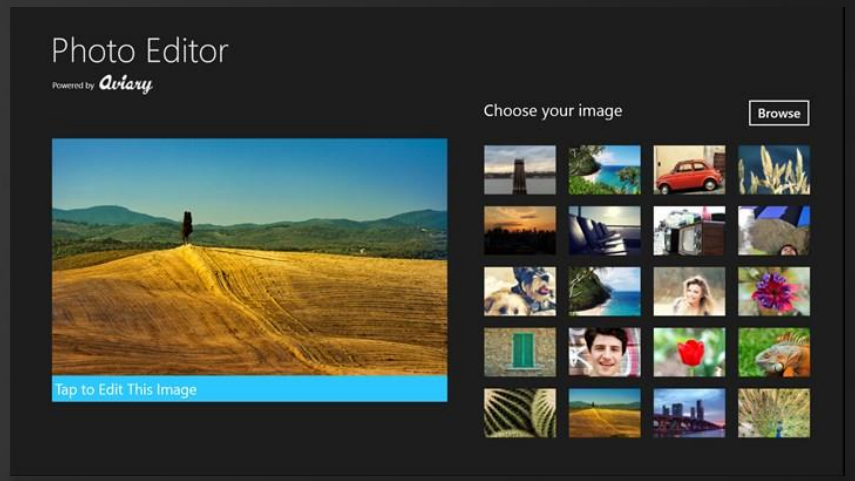
Today’s Agenda:

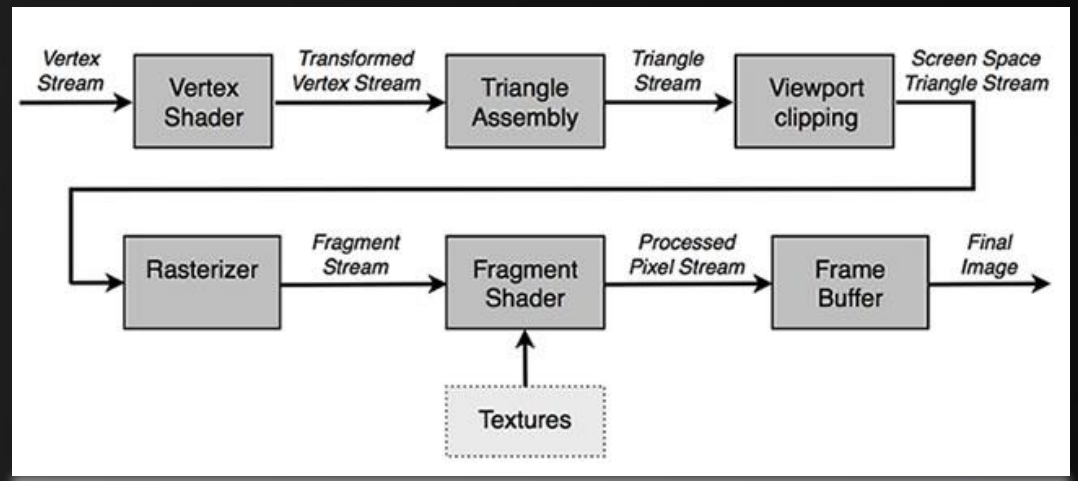
- Caching: Recap
- Data Locality
- Alignment
- False Sharing
- A Handy Guide *(to Pleasing the Cache)*

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

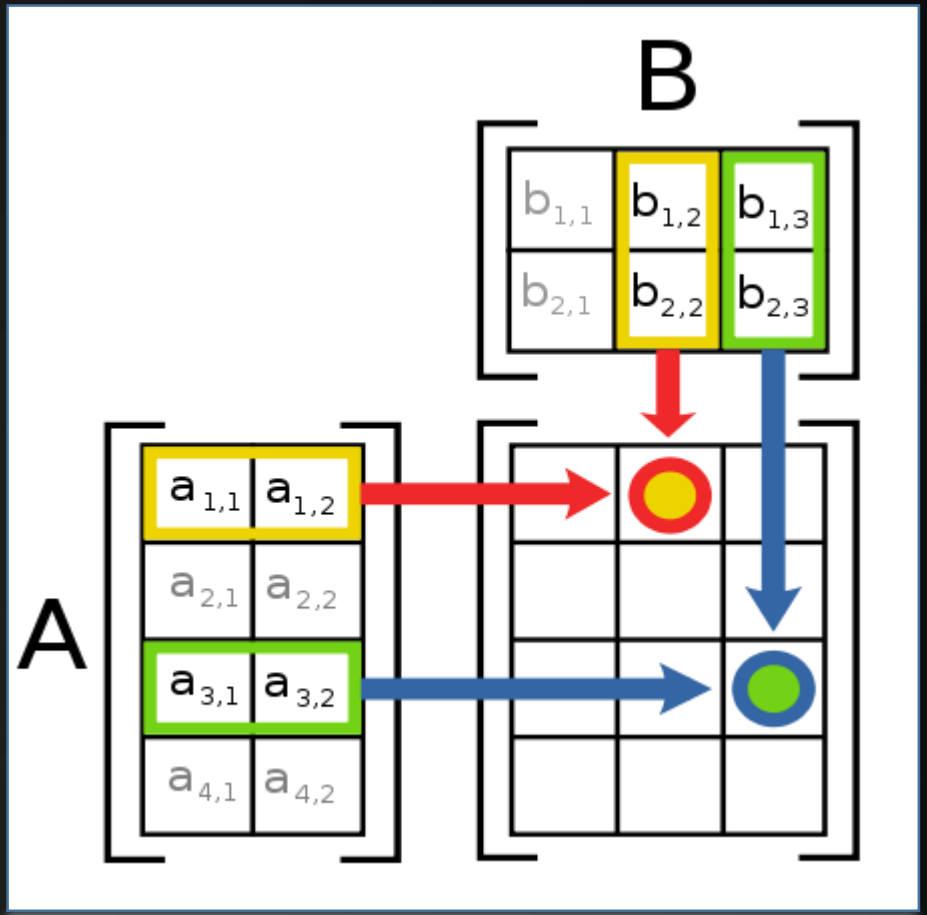
Reusing data

Ideal pattern:

- load data sequentially.

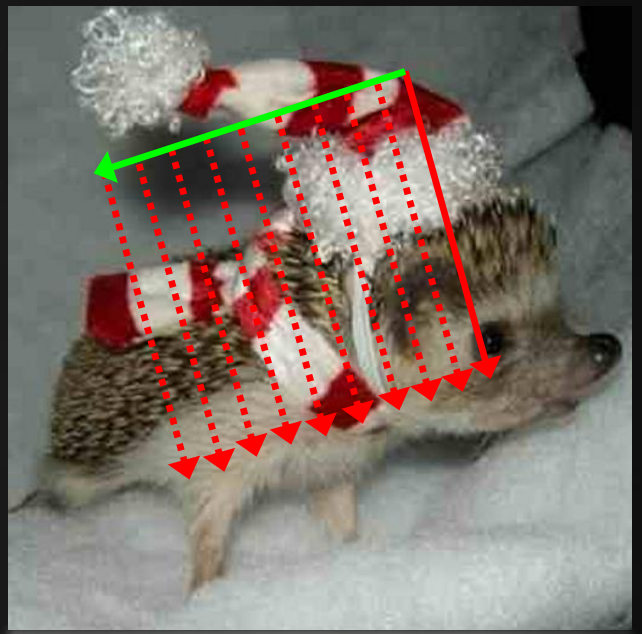
Typical pattern:

- *whatever the algorithm dictates.*



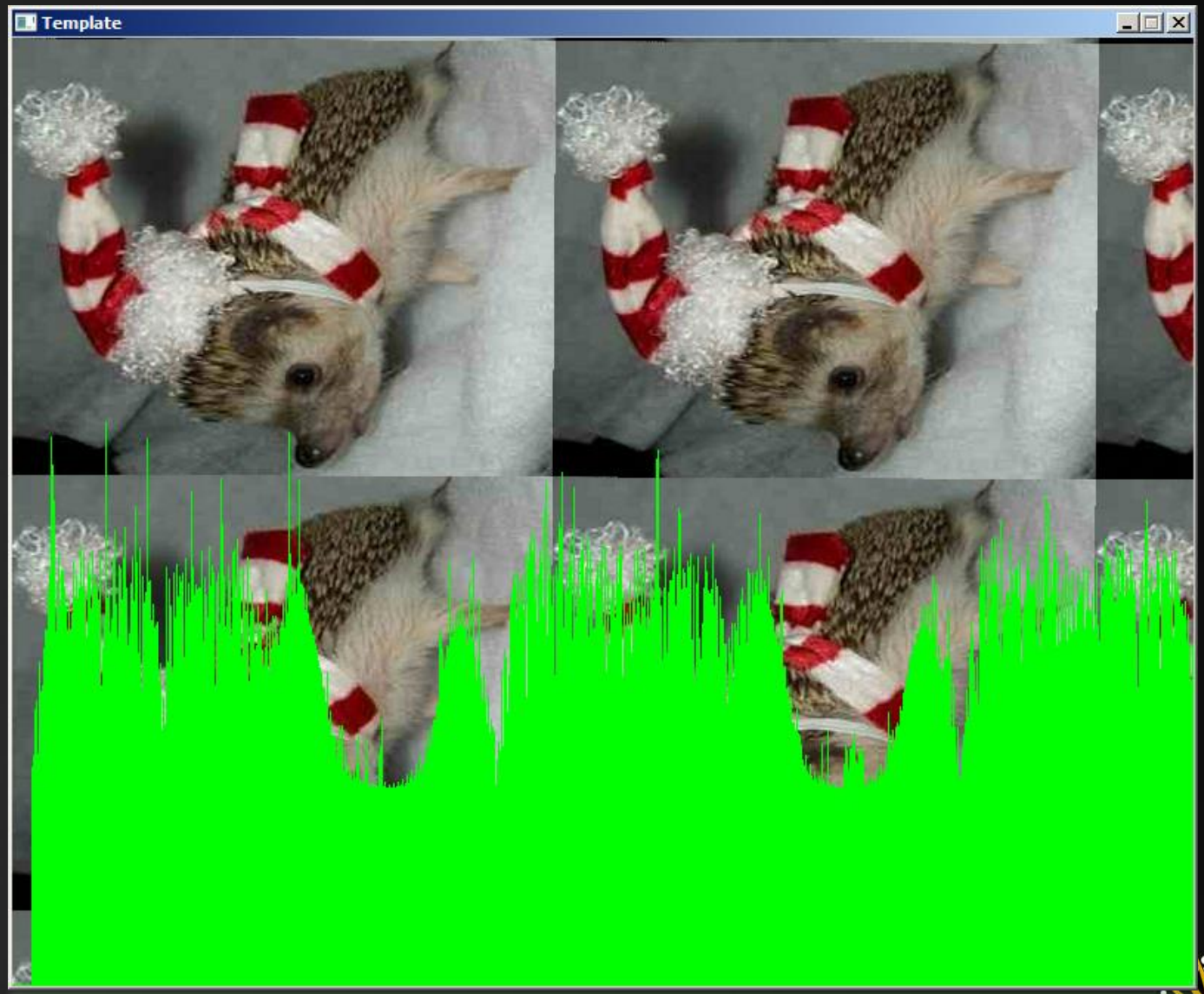
Data Locality

Example: rotozooming



Data Locality

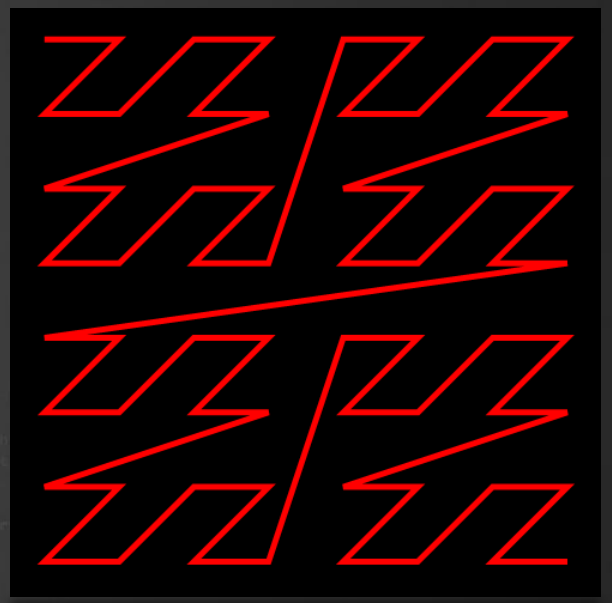
Example: rotozooming



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

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

M = 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:

- 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?

```
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];
```

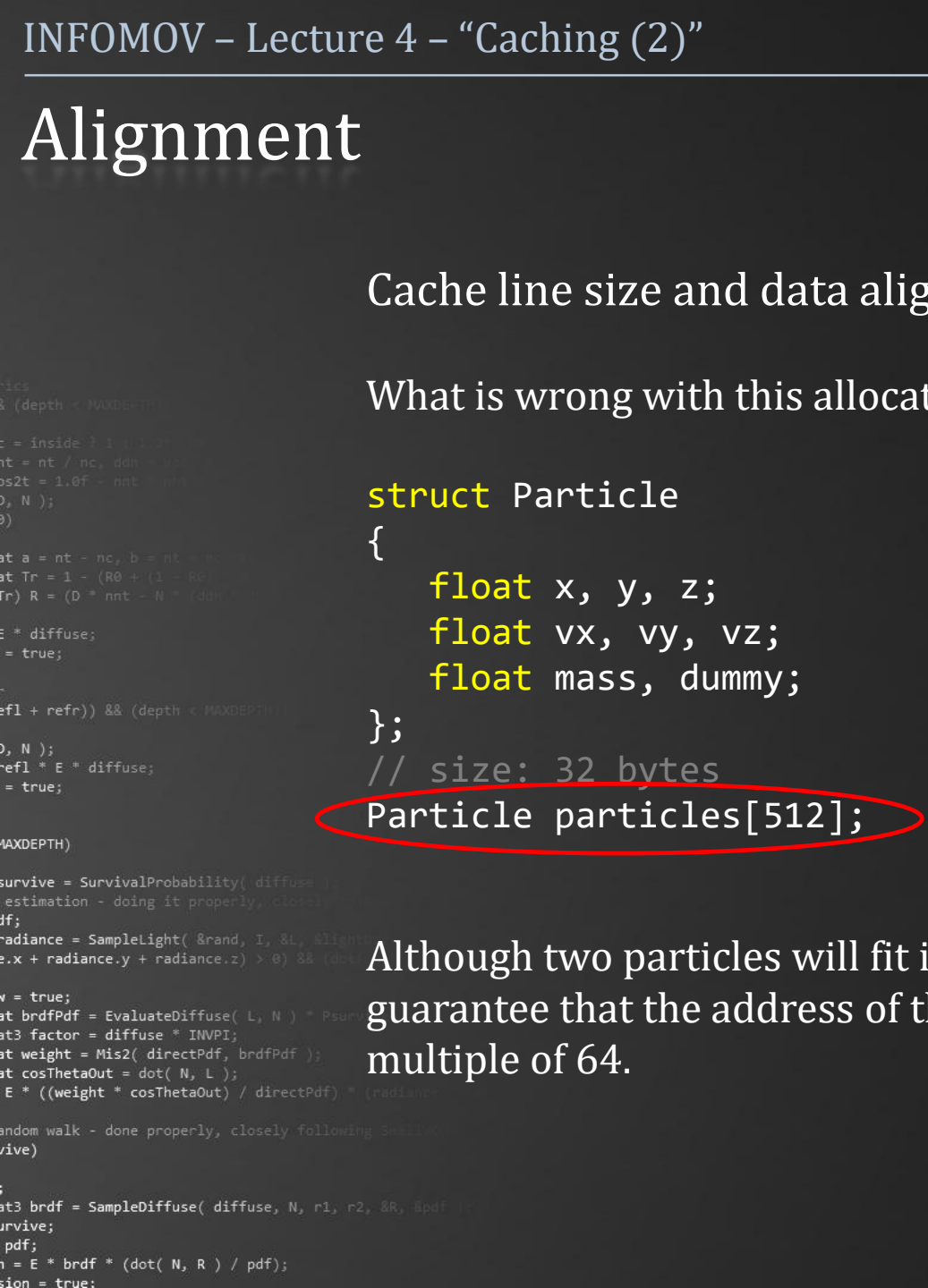
Although two particles will fit in a cache line, we have no guarantee that the address of the first particle is a multiple of 64.

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:

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

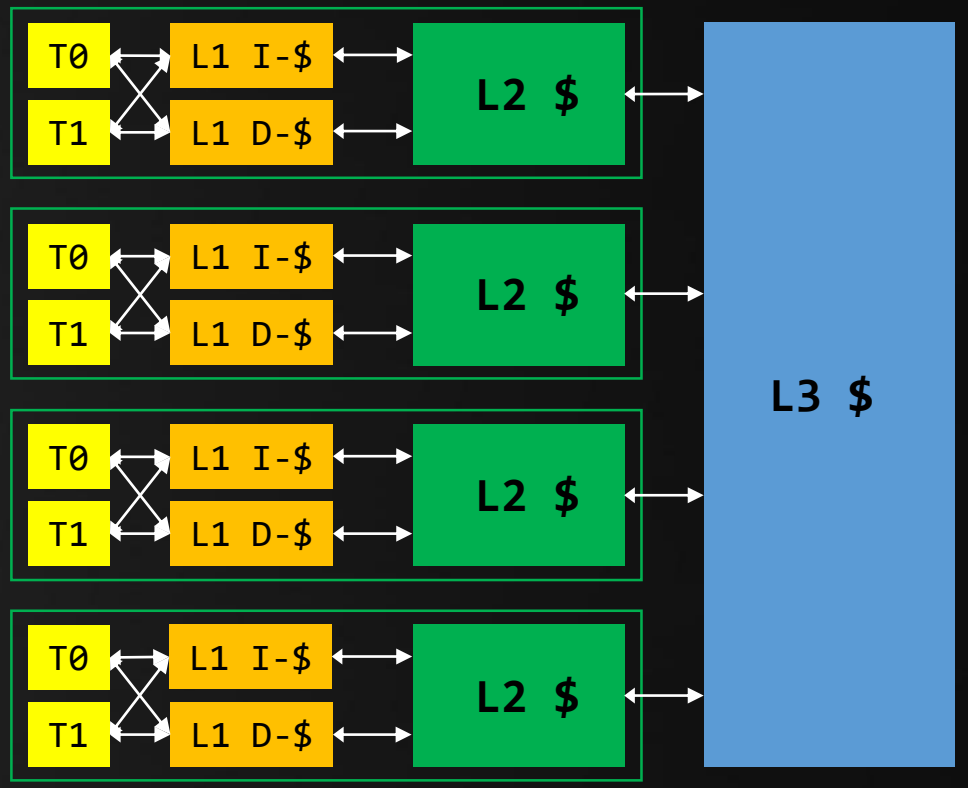
False Sharing

Multiple Cores using Caches

Two cores can hold copies of the same data.

Not as unlikely as you may think – Example:

```
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
Unions 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

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!

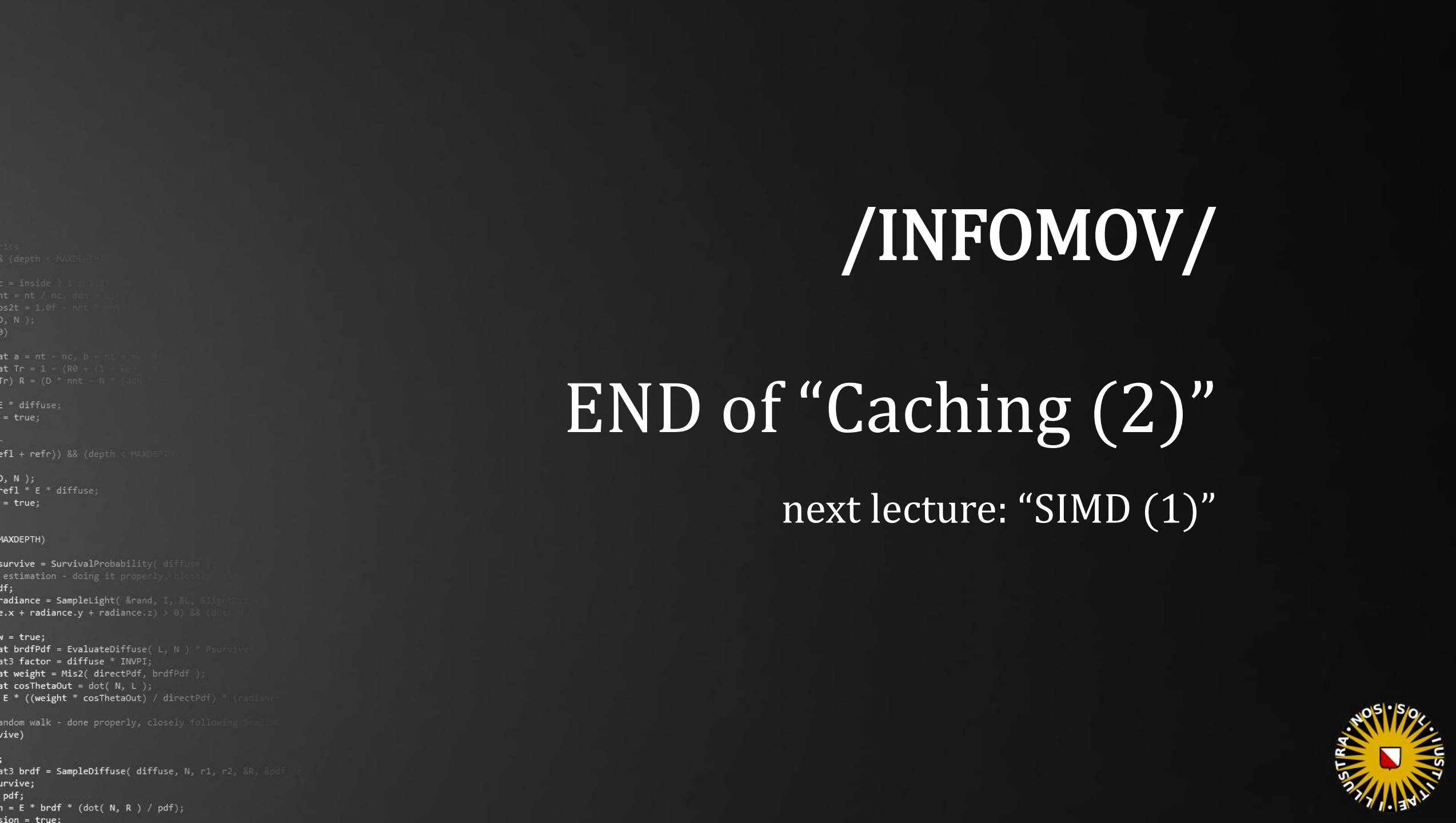
Today’s Agenda:

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

/INFOMOV/

END of “Caching (2)”

next lecture: “SIMD (1)”



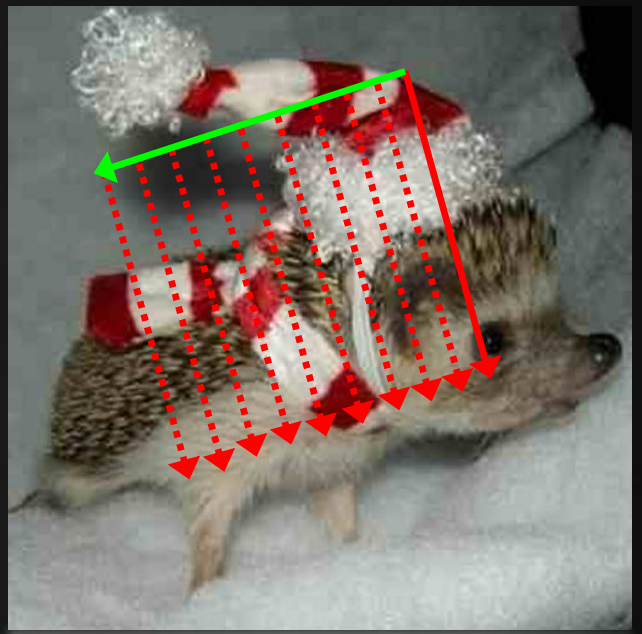
/INFOMOV/

Practical

1. Timing the Rotozoomer

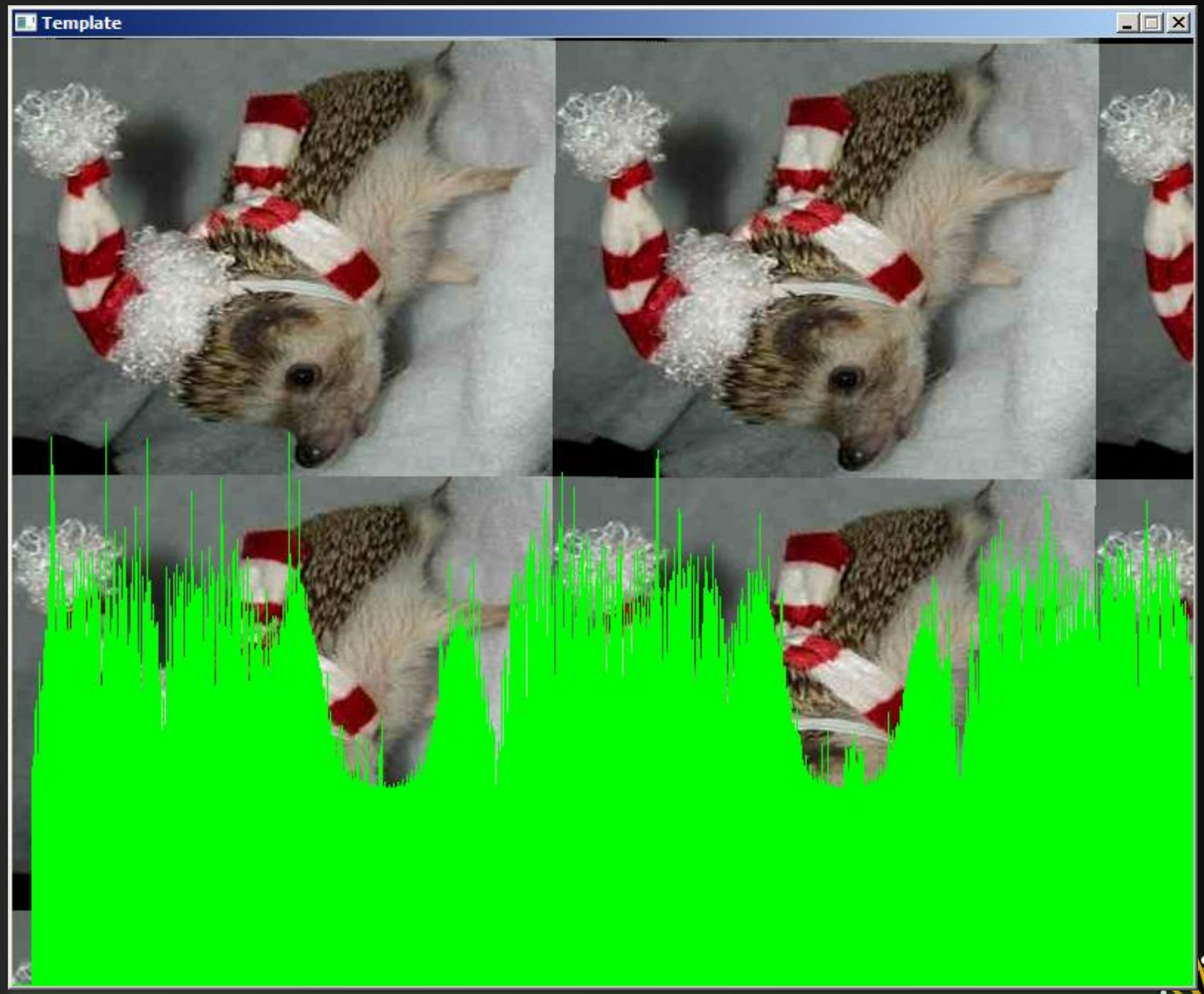
Data Locality

Example: rotozooming



Data Locality

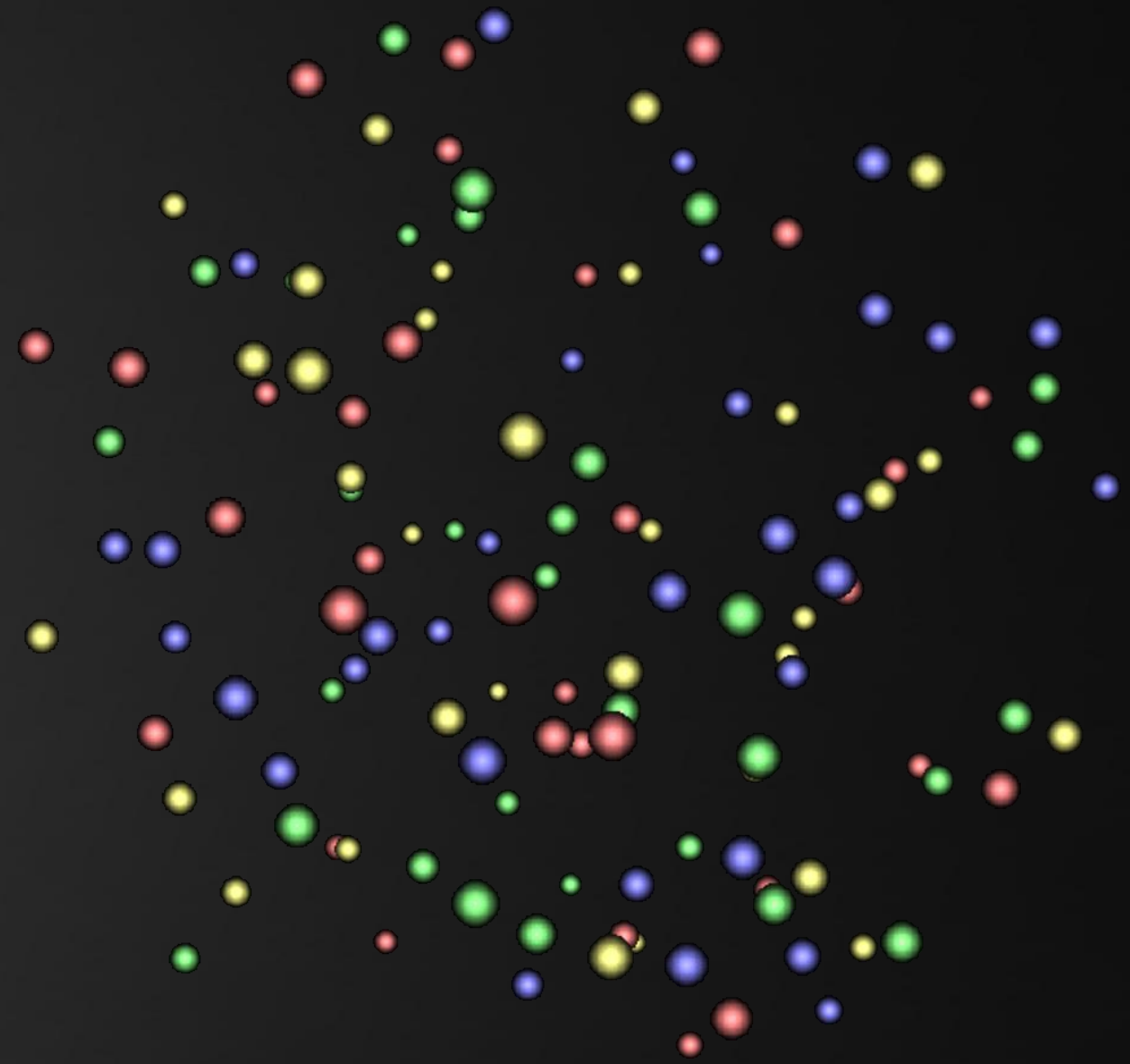
Example: rotozooming



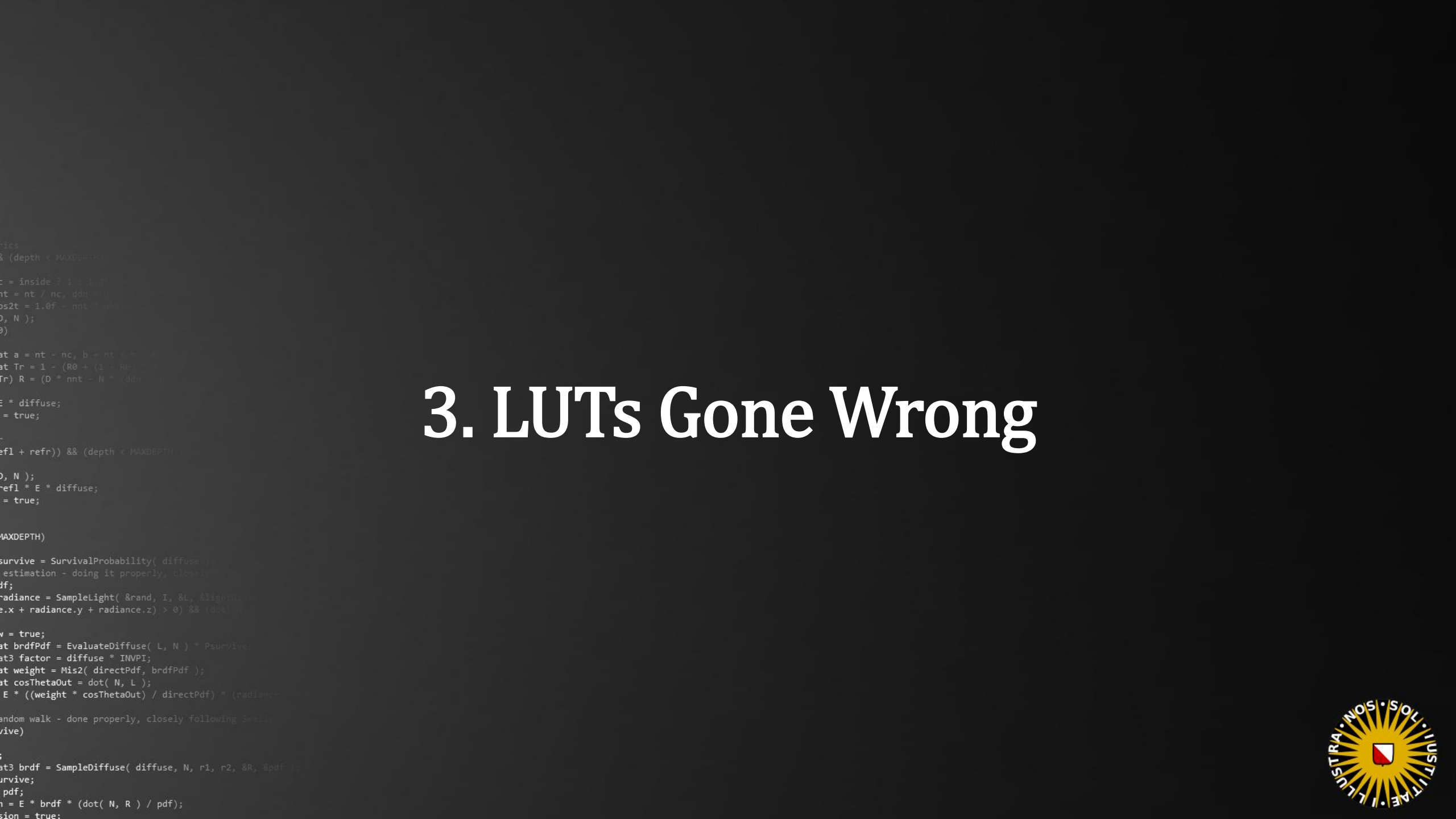
2. Simple Access Pattern

Data Locality

Example: dotcloud



3. LUTs Gone Wrong



Source | Assembly | Assembly grouping: Address

...	Source	Clockticks	Instructions Retired
474	#if 0		
475	Pixel p0 = src[x0 + y0 * m_Pitch];		
476	Pixel p1 = src[x1 + y0 * m_Pitch];		
477	Pixel p2 = src[x0 + y1 * m_Pitch];		
478	Pixel p3 = src[x1 + y1 * m_Pitch];		
479	Pixel scaledp0 = ScaleColor(p0, (int)(w0 * 255.9f));		
480	Pixel scaledp1 = ScaleColor(p1, (int)(w1 * 255.9f));		
481	Pixel scaledp2 = ScaleColor(p2, (int)(w2 * 255.9f));		
482	Pixel scaledp3 = ScaleColor(p3, (int)(w3 * 255.9f));		
483	#else		
484	Pixel scaledp0 = m_Surface->m_Scaled[(int)(w0 * 255.9f)][x0	870,400,000	384,000,000
485	Pixel scaledp1 = m_Surface->m_Scaled[(int)(w1 * 255.9f)][x1	4,406,400,000	3,267,200,000
486	Pixel scaledp2 = m_Surface->m_Scaled[(int)(w2 * 255.9f)][x0		
487	Pixel scaledp3 = m_Surface->m_Scaled[(int)(w3 * 255.9f)][x1	582,400,000	521,600,000
488	#endif		
489	Pixel color = scaledp0 + scaledp1 + scaledp2 + scaledp3;	7,881,600,000	8,633,600,000
490	a_Target->GetBuffer()[i + j * a_Target->GetPitch()] = AddBl	7,468,800,000	4,371,200,000
491	}		
492	}		
493			
494	void Sprite::InitializeStartData()		
495	{		
496	for (unsigned int f = 0; f < m_NumFrames; ++f)		
497	{		
498	m_Start[f] = new unsigned int[m_Height];		
499	for (int y = 0; y < m_Height; ++y)		
500	{		
501	m_Start[f][y] = m_Width;		
502	Pixel* addr = GetBuffer() + f * m_Width + y * m_Pitch;		
503	for (int x = 0; x < m_Width; ++x)		
504	{		
505	if (addr[x])		
506	{		
507	m_Start[f][y] = x;		
508	break;		
509	}		
510	}		
511	}		
512	}		
513	}		
514			
515	Font::Font(const char *a_File, const char *a_Chars)		
516	{		
517	m_Surface = new Surface(a_File);		
518	Pixel* b = m_Surface->GetBuffer();		
519	int w = m_Surface->GetWidth();		
520	int h = m_Surface->GetHeight();		
521	unsigned int charpr = 0, start = 0;		

Address ▲	Sour...	Assembly	Locators: ...d Bound	Bad Speculation	Back-End B... Memory Bound
0x140006d00	484	addss xmm0, xmm6	0.0%	0.0%	0.0%
0x140006d04	487	imul r10d, ecx	0.0%	0.1%	0.0%
0x140006d08	487	imul r11d, r8d	0.0%	0.1%	0.3%
0x140006d0c	487	addss xmm0, xmm4			
0x140006d10	487	subss xmm1, xmm0	0.0%	0.0%	
0x140006d14	487	movss xmm0, dword ptr [rip+0x307c]	0.0%	0.0%	0.0%
0x140006d1c	487	mulss xmm4, xmm0	0.0%	0.0%	0.7%
0x140006d20	487	mulss xmm6, xmm0	0.0%	0.0%	0.0%
0x140006d24	487	cvttss2si eax, xmm4			
0x140006d28	487	movsxd rcx, eax	0.0%	0.0%	0.5%
0x140006d2b	487	lea eax, ptr [r10+rdi*1]	0.0%	0.0%	0.0%
0x140006d2f	487	movsxd r9, eax			
0x140006d32	487	cvttss2si eax, xmm6	0.0%	0.0%	0.0%
0x140006d36	489	mov r8, qword ptr [rbx+rcx*8]	0.0%	0.0%	0.2%
0x140006d3a	489	movsxd rcx, eax	0.0%	0.0%	0.0%
0x140006d3d	489	lea eax, ptr [r11+rdi*1]	0.0%	0.0%	0.0%
0x140006d41	489	mov edi, dword ptr [r8+r9*4]	0.0%	0.0%	0.3%
0x140006d45	489	movsxd rdx, eax	0.0%	0.0%	19.0%
0x140006d48	489	mov rax, qword ptr [rbx+rcx*8]	0.0%	0.0%	0.2%
0x140006d4c	489	add edi, dword ptr [rax+rdx*4]	0.0%	0.0%	0.0%
0x140006d4f	485	mulss xmm1, xmm0	0.0%	0.0%	12.7%
0x140006d53	485	mulss xmm7, xmm0	0.0%	0.0%	0.0%
0x140006d57	487	cvttss2si eax, xmm1			
0x140006d5b	487	xorps xmm1, xmm1	0.0%	0.0%	0.5%
0x140006d5e	487	movsxd rcx, eax			
0x140006d61	487	lea eax, ptr [r10+rsi*1]	0.0%	0.0%	0.0%
0x140006d65	490	mov r10, qword ptr [r13]	0.0%	0.0%	
0x140006d69	490	movsxd rdx, eax	0.0%	0.0%	0.3%
0x140006d6c	490	mov rax, qword ptr [rbx+rcx*8]	0.0%	0.0%	0.0%
0x140006d70	490	add edi, dword ptr [rax+rdx*4]	0.0%	0.0%	0.0%
0x140006d73	490	cvttss2si eax, xmm7	0.0%	0.8%	9.4%
0x140006d77	490	movsxd rcx, eax	0.0%	0.1%	0.3%
0x140006d7a	490	lea eax, ptr [r11+rsi*1]			
0x140006d7e	490	movsxd rdx, eax	0.0%	0.0%	0.0%
0x140006d81	490	mov rax, qword ptr [rbx+rcx*8]	0.0%	0.0%	0.3%
0x140006d85	490	add edi, dword ptr [rax+rdx*4]			
0x140006d88	490	mov eax, dword ptr [r13+0x10]	0.0%	0.2%	6.3%
0x140006d8c	490	imul eax, r15d	0.0%	0.1%	0.0%
0x140006d90	490	inc r15d	0.0%	0.0%	0.2%
0x140006d93	490	add eax, ebp	0.0%	0.0%	0.0%
0x140006d95	490	movsxd r11, eax	0.0%	0.0%	0.2%
0x140006d98	490	mov eax, edi	0.0%	0.0%	0.0%
0x140006d9a	490	and eax, 0xff0000	0.0%	0.0%	0.0%
0x140006d9f	490	mov ecx, dword ptr [r10+r11*4]			
0x140006da3	490	mov r8d, ecx	0.0%	0.0%	0.0%
0x140006da6	490	mov r9d, ecx	0.0%	0.0%	0.2%

/INFOMOV/

End of Practical