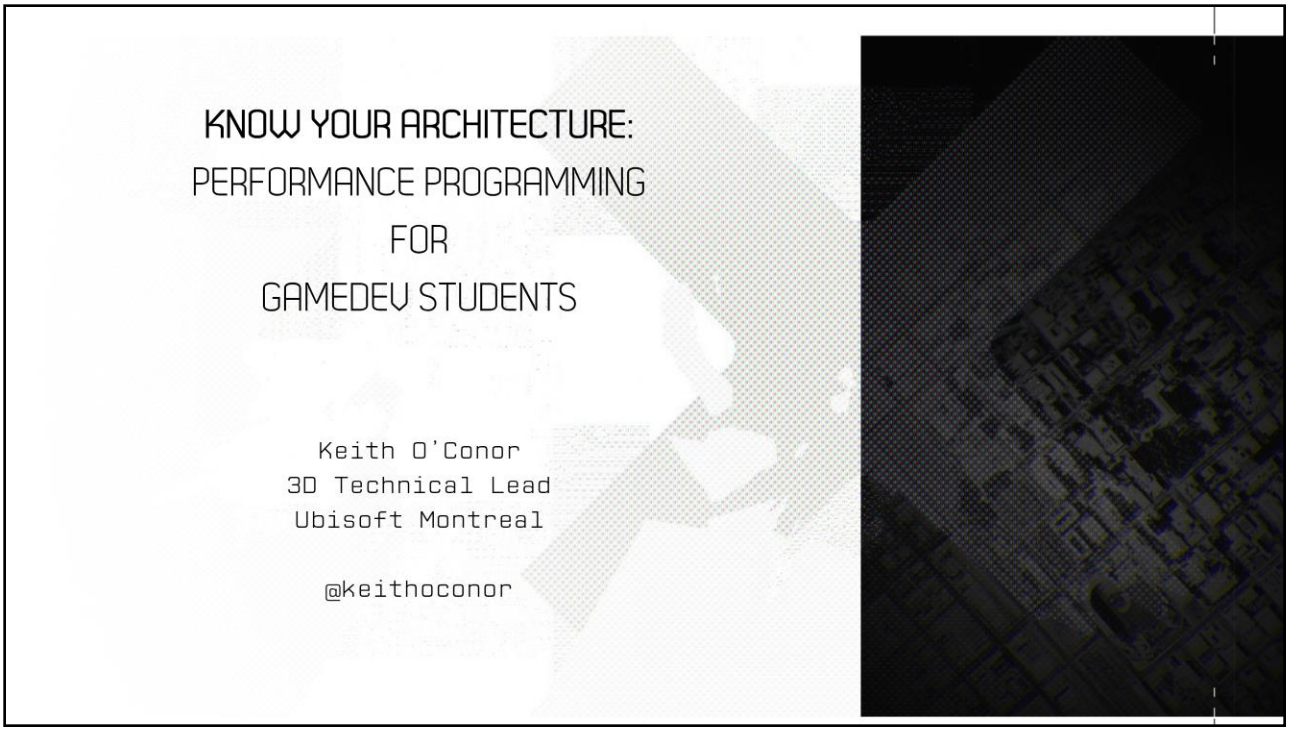
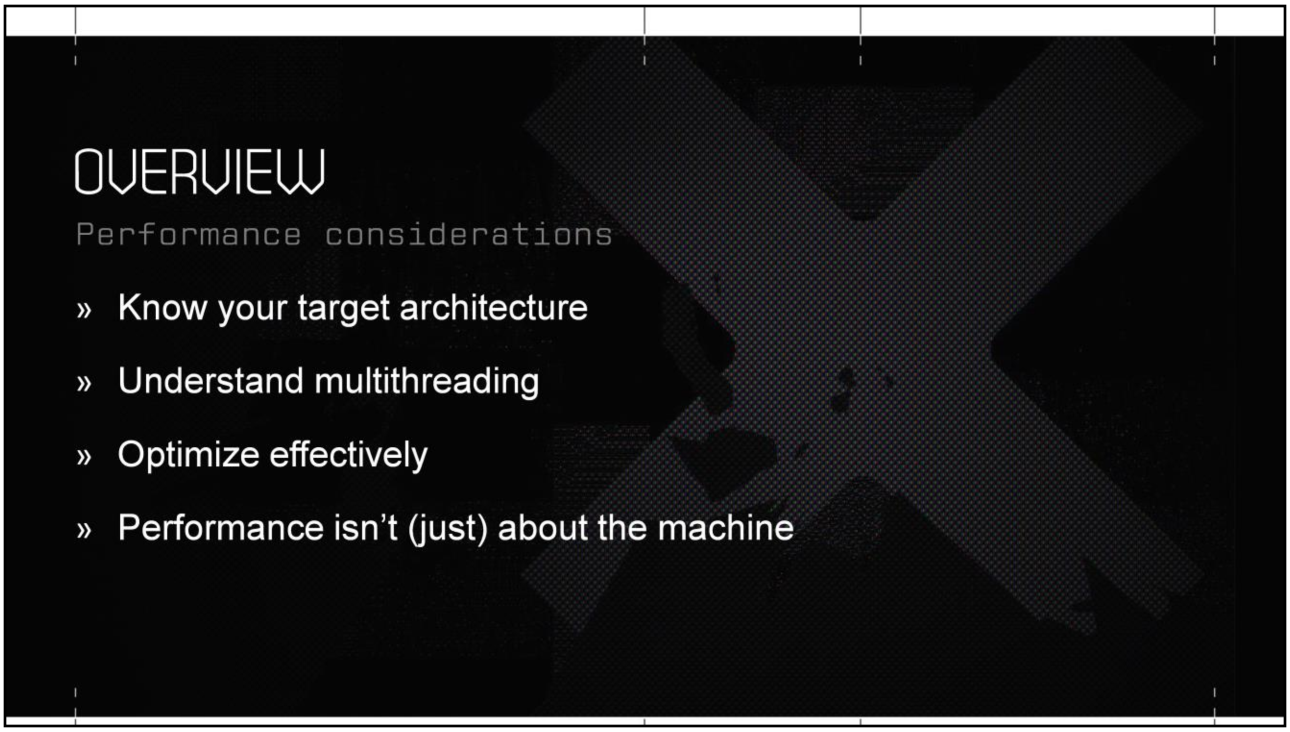


KNOW YOUR ARCHITECTURE: PERFORMANCE PROGRAMMING FOR GAMEDEV STUDENTS

Keith O'Conor
3D Technical Lead
Ubisoft Montreal

@keithoconor

- Who I am
    - PhD (Trinity College Dublin), Radical (shipped Prototype 1 & 2), Ubisoft Montreal (shipped Watch_Dogs, now on Far Cry 4)
- Who this is aimed at
    - Game programming students who don't necessarily come from a strict computer science background
    - Some points might be basic for CS students, but all are important
- All points could be talked about much more
    - Use as a starting point for further reading – see references at end of deck



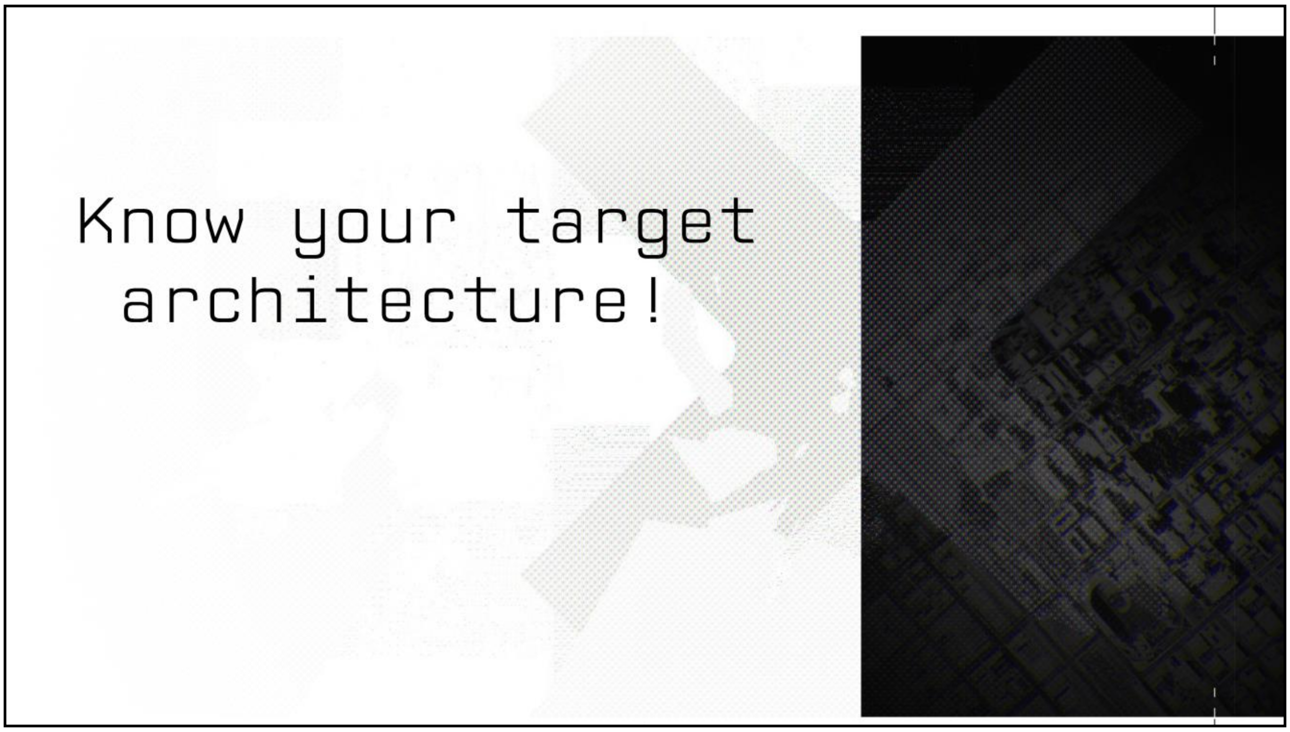
OVERVIEW

Performance considerations

- Know your target architecture
- Understand multithreading
- Optimize effectively
- Performance isn't (just) about the machine

Easy to lose sight of perf

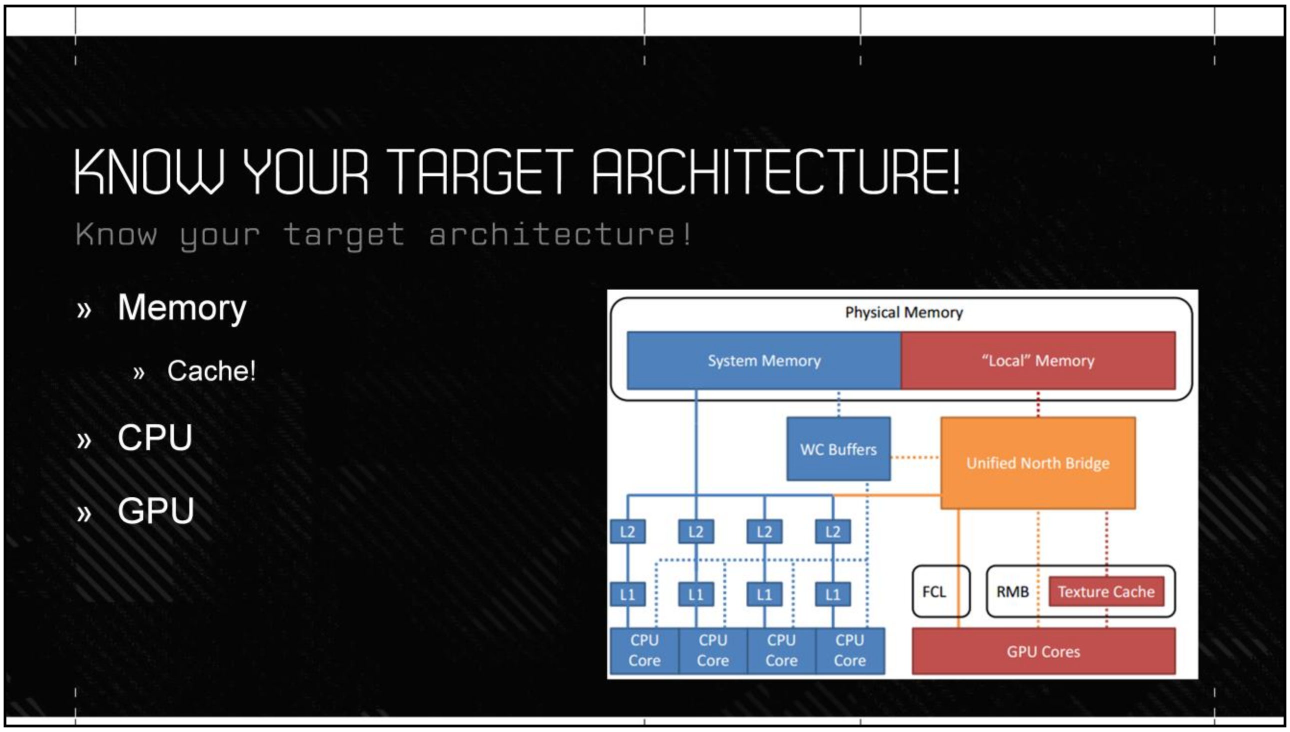
- High-level goals
- Time pressures
- Designitis
- "I'll fix it later" doesn't happen as often as you'd like



Know your target architecture!

One takeaway from this talk!

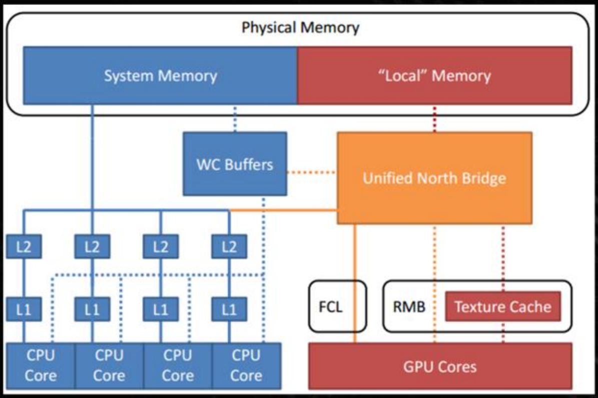
You can't get (closer to) optimal performance unless you know what's going on at a low level



KNOW YOUR TARGET ARCHITECTURE!

Know your target architecture!

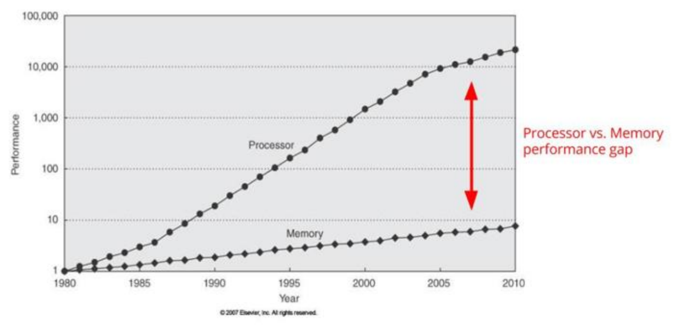
- Memory
  - Cache!
- CPU
- GPU



Memory

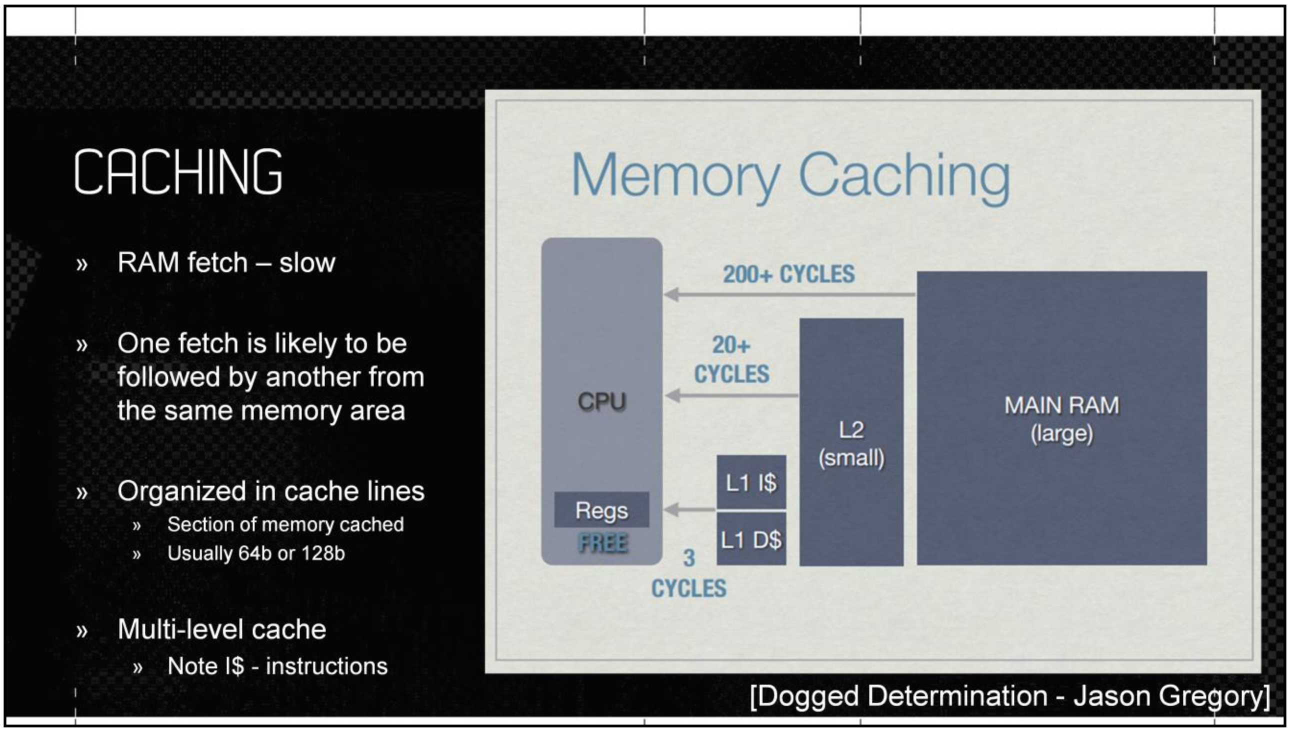
MEMORY

» Data fetch speed can be a bottleneck



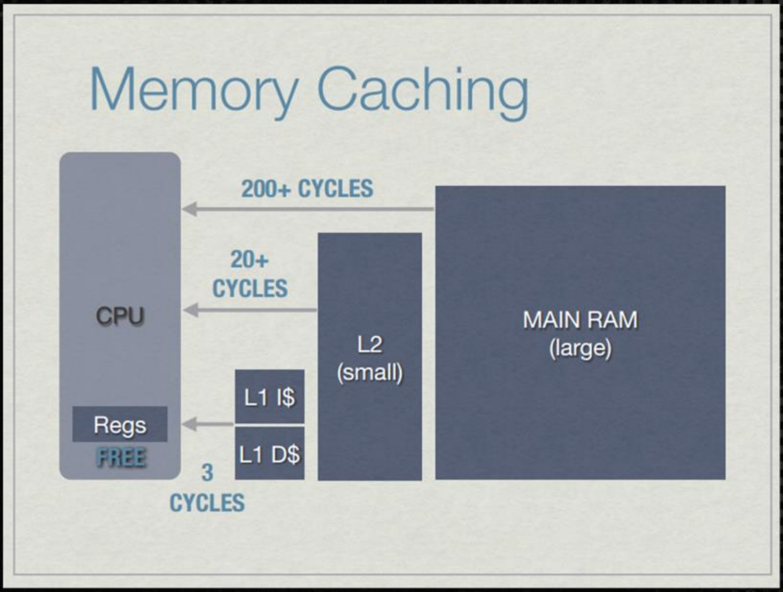
Memory speed lags behind CPU speed

Obligatory Moore's-Law-vs-memory image



CACHING

- RAM fetch – slow
- One fetch is likely to be followed by another from the same memory area
- Organized in cache lines
  - Section of memory cached
  - Usually 64b or 128b
- Multi-level cache
  - Note I$ - instructions



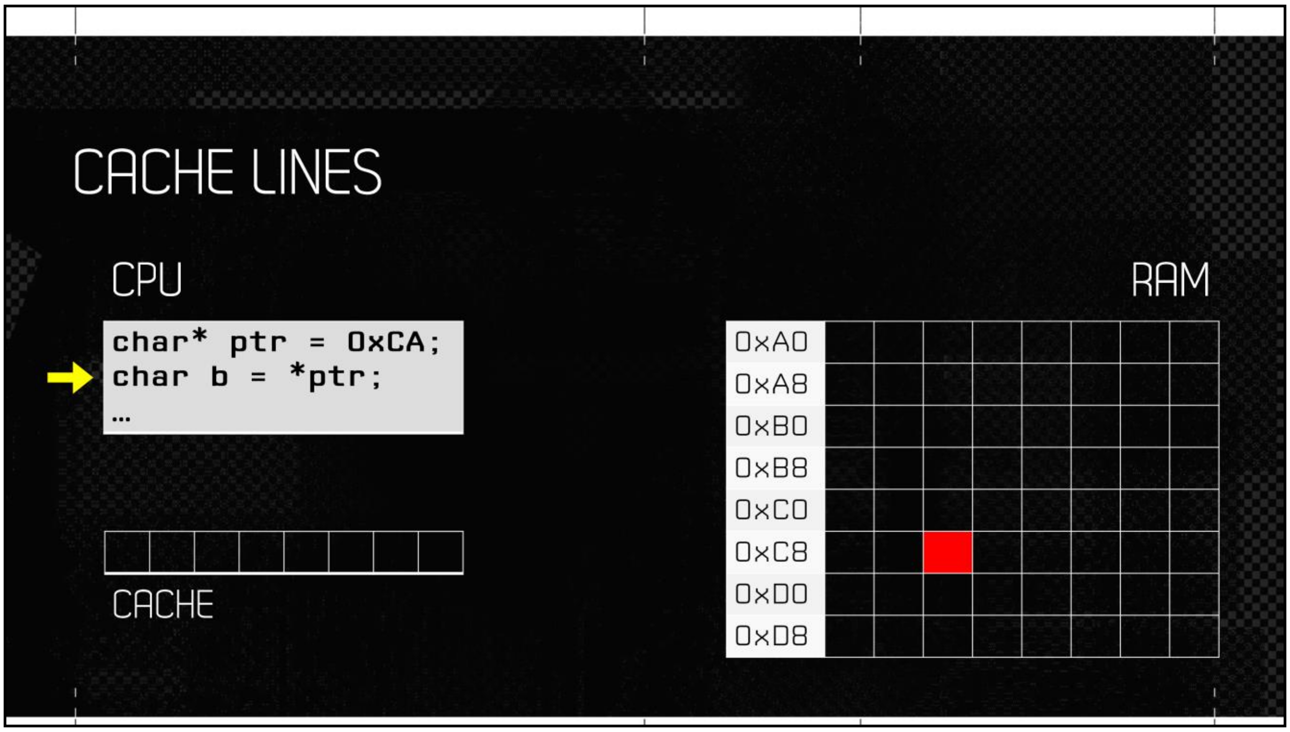
[Dogged Determination - Jason Gregory]

Note instruction cache – usually not as much of a focus as data cache, but still worth considering.

Good advice from Jason Gregory:

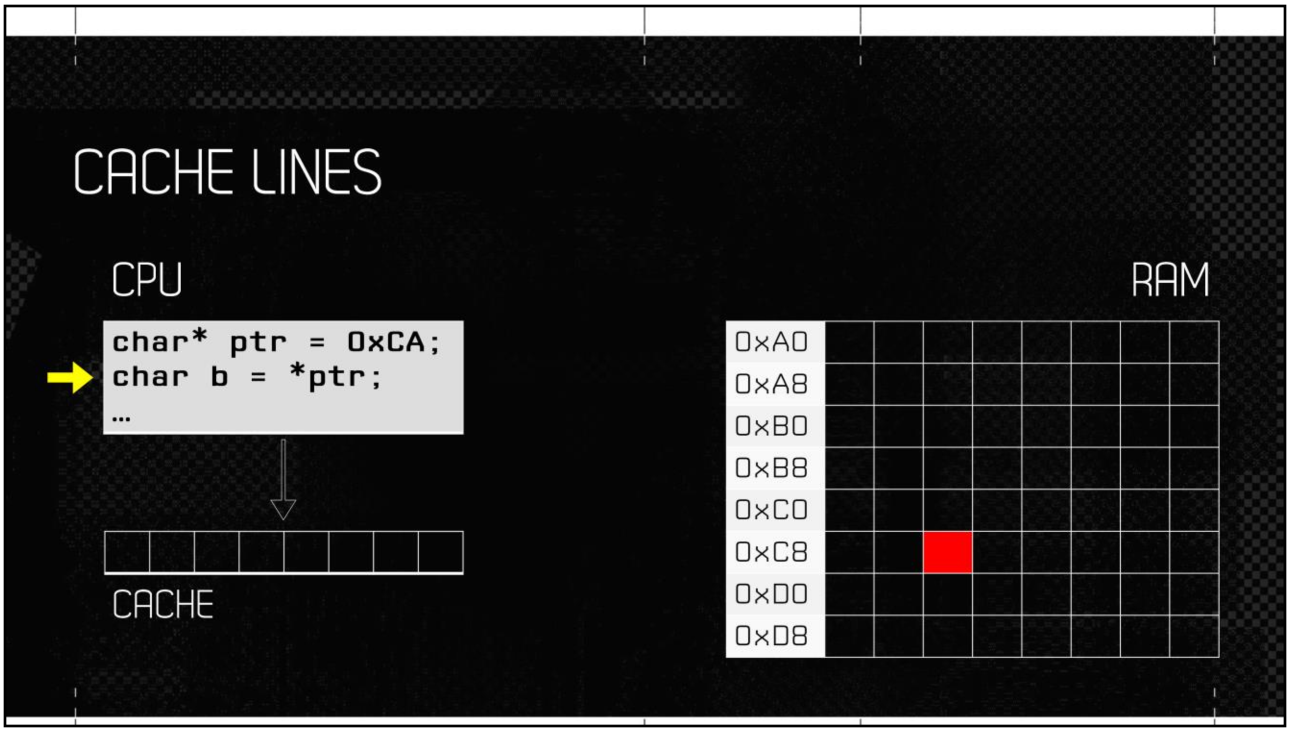
Keep high-performance code small

Keep high-performance data small and contiguous



CACHE LINES

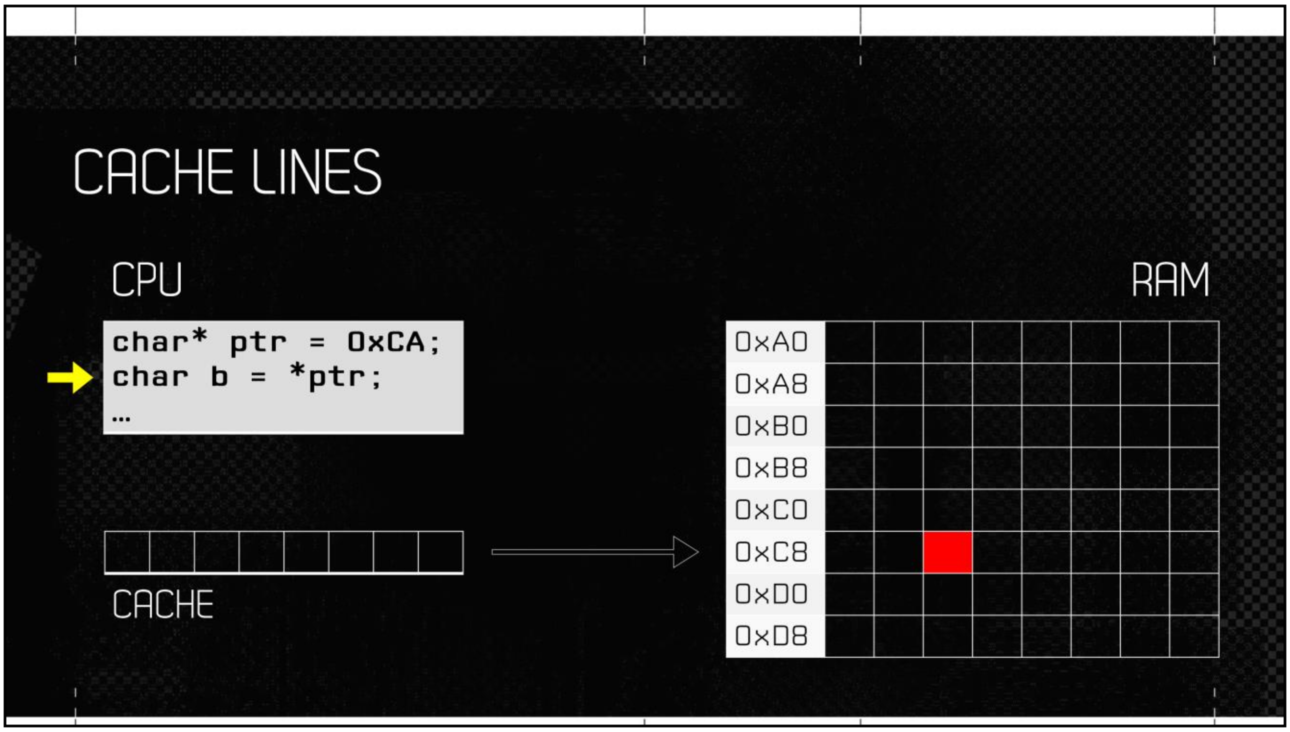
```
char* ptr = 0xCA;
char b = *ptr;
…
```



CACHE LINES

```
char* ptr = 0xCA;
char b = *ptr;
…
```

Pointer dereference

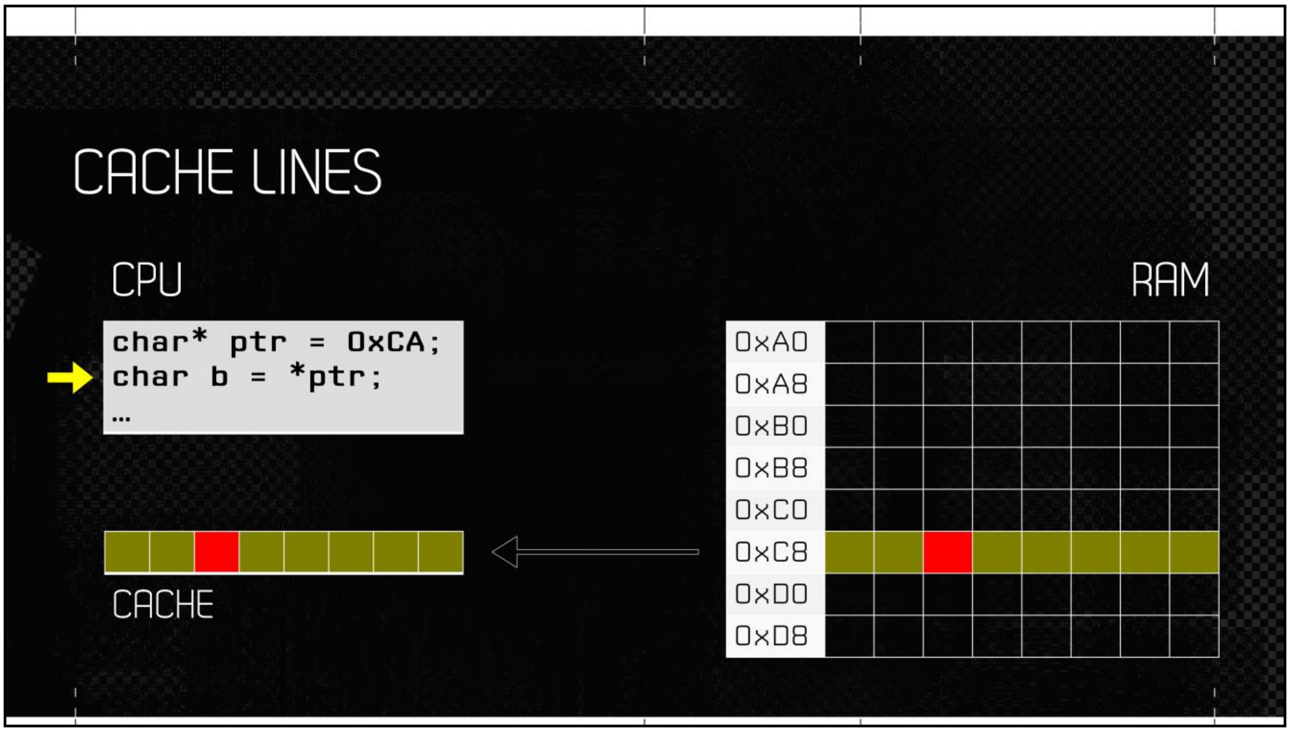


CACHE LINES

```
char* ptr = 0xCA;
char b = *ptr;
…
```

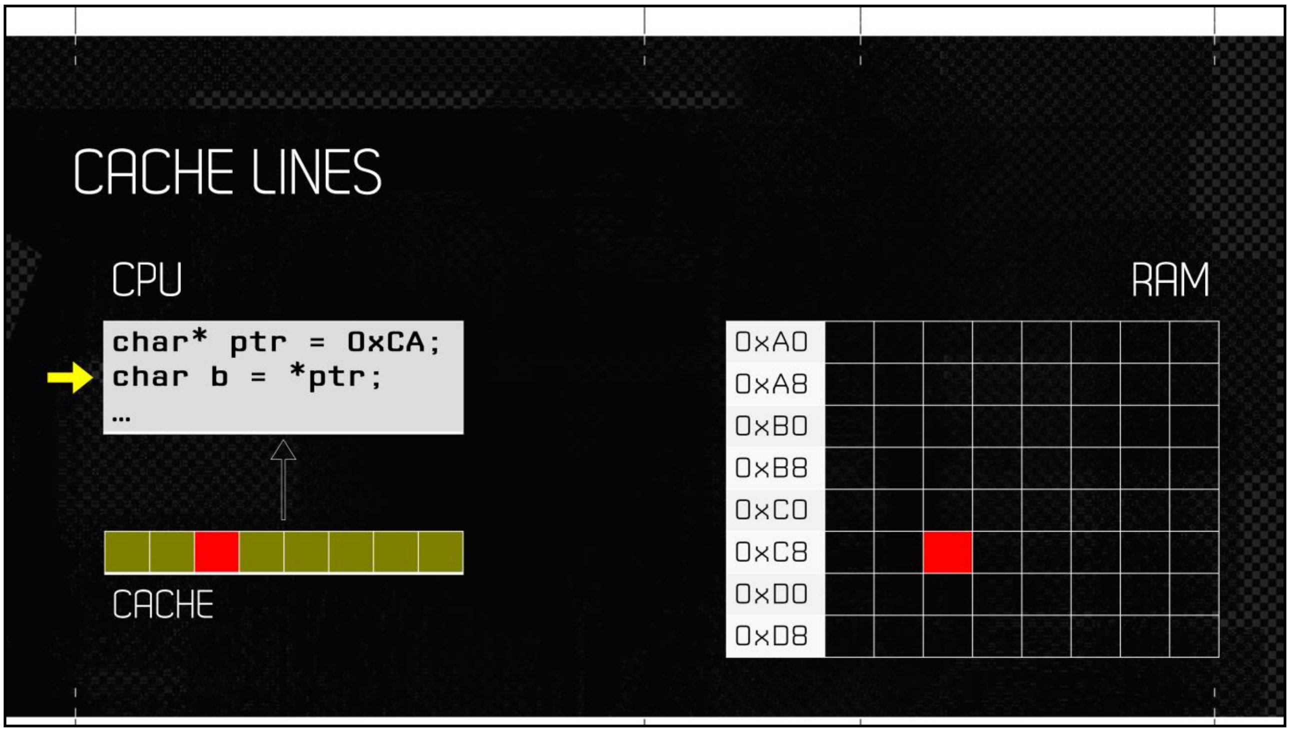
Cache miss!

Results in expensive fetch from main memory



```
char* ptr = 0xCA;
char b = *ptr;
…
```

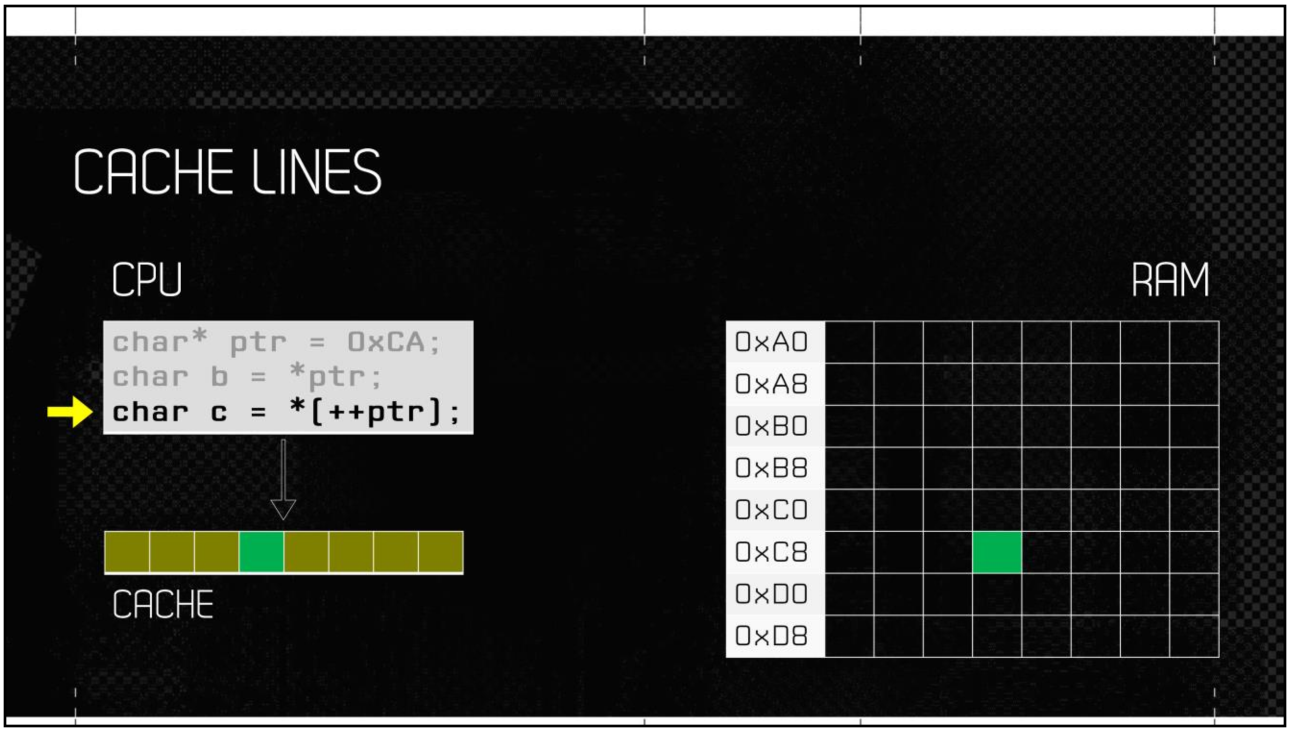
Whole cache line (8b in this case) worth of memory fetched and cached



CACHE LINES

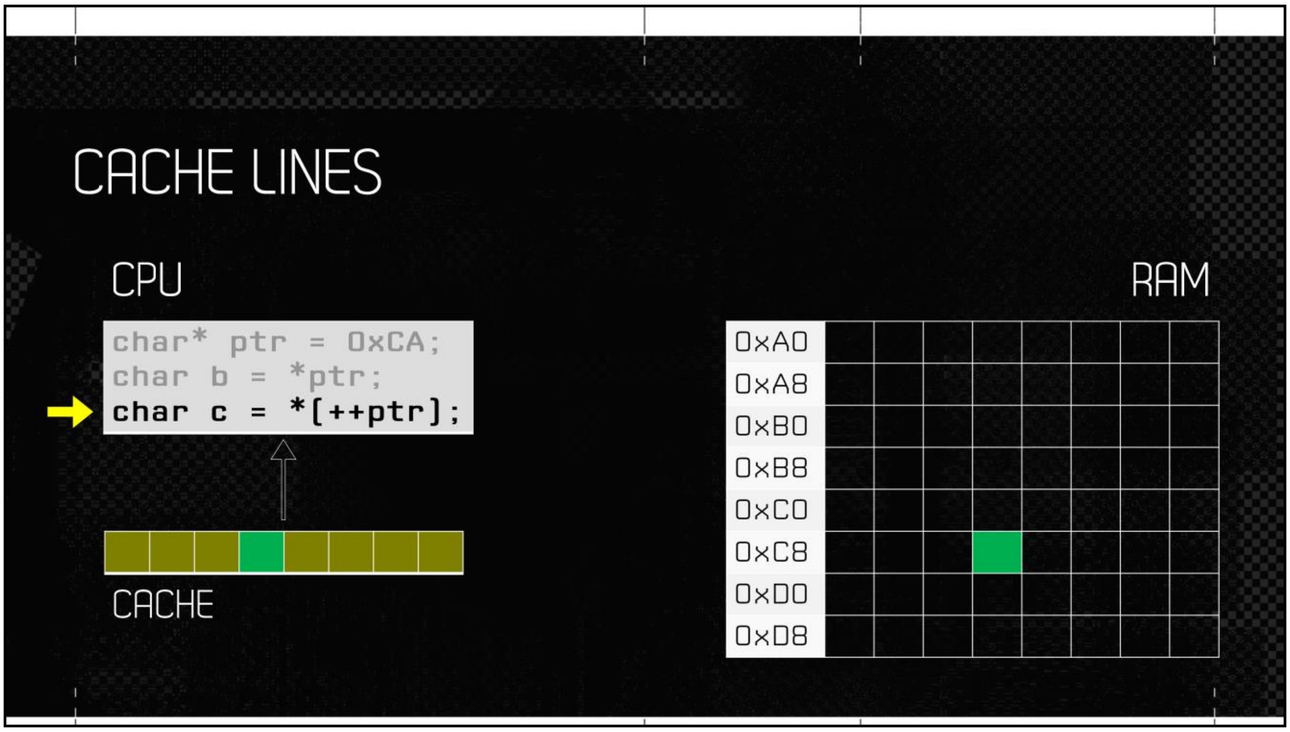
```
char* ptr = 0xCA;
char b = *ptr;
…
```

Value returned



```
char* ptr = 0xCA;
char b = *ptr;
char c = *(++ptr);
```

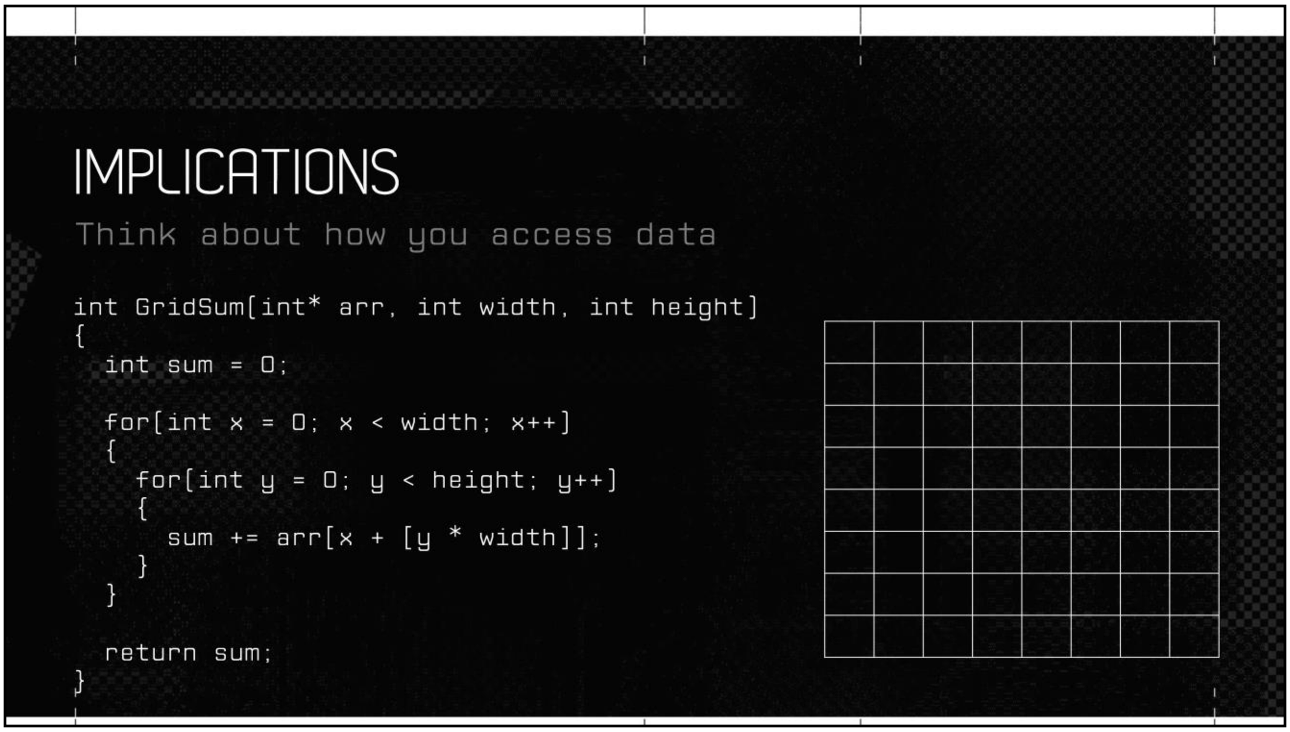
Adjacent memory location is fetched



CACHE LINES

```
char* ptr = 0xCA;
char b = *ptr;
char c = *(++ptr);
```

Cache hit!



IMPLICATIONS

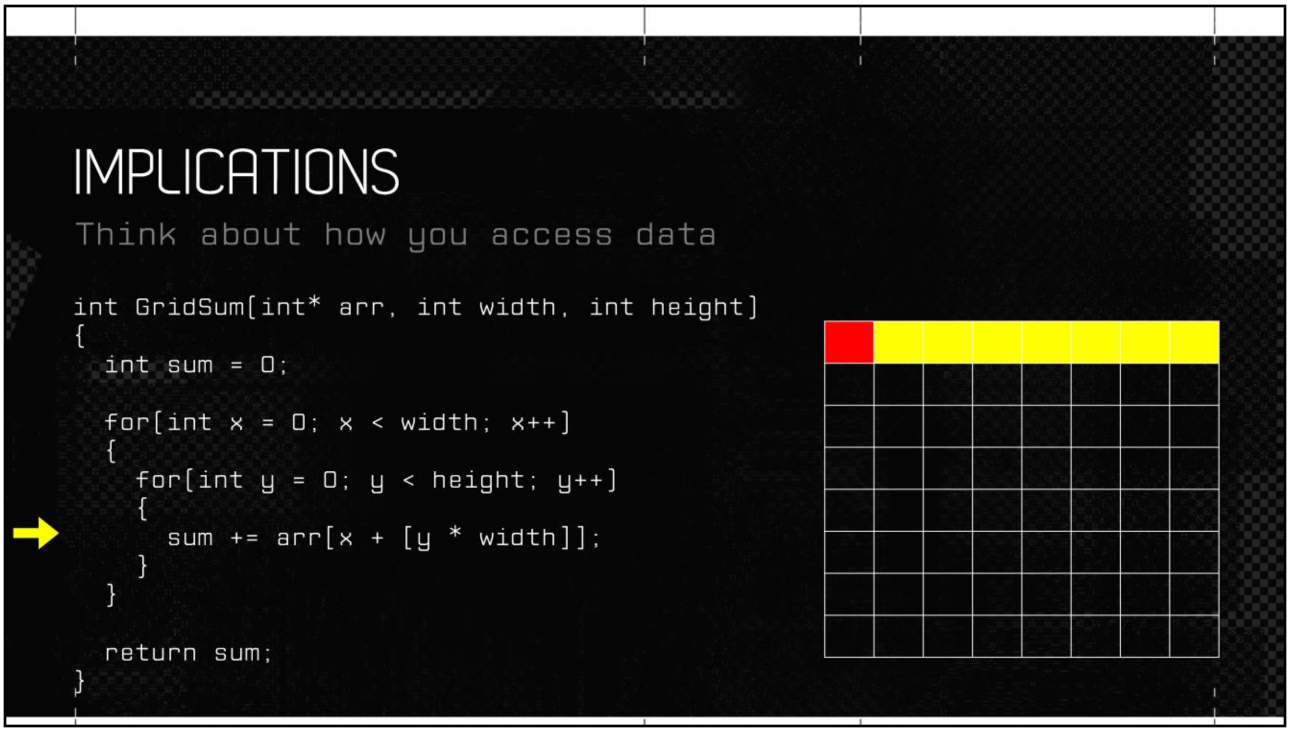
Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Implication: think about how you access data

Canonical example: 2D array sum

Contrived, but not that far from real-world scenarios

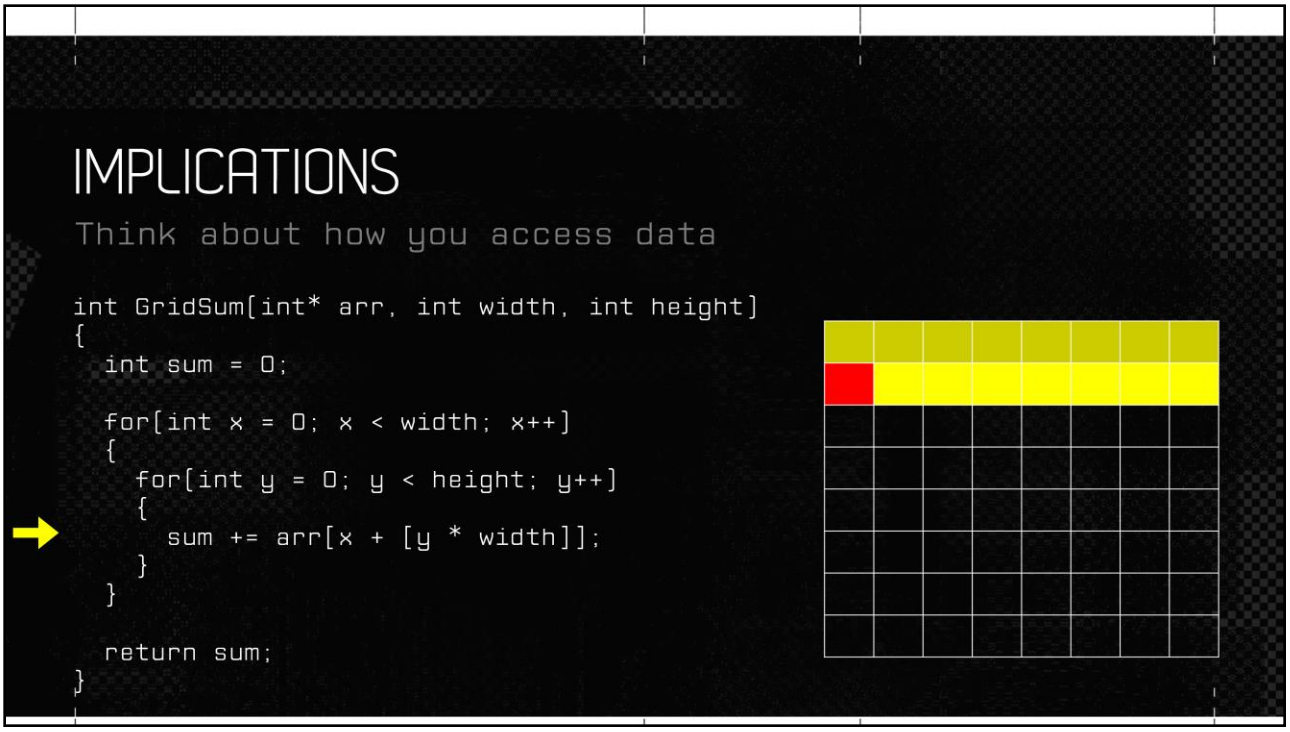


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

First iteration – cache miss

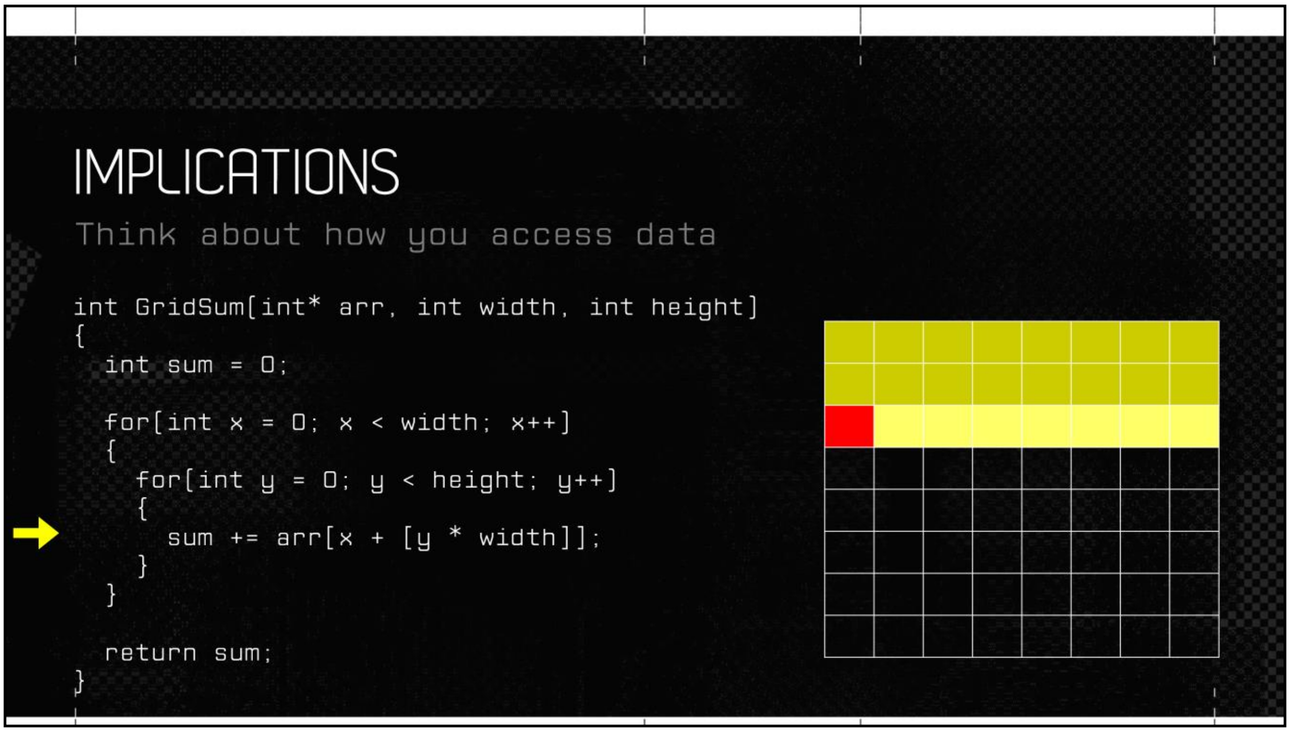


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Second iteration – cache miss!

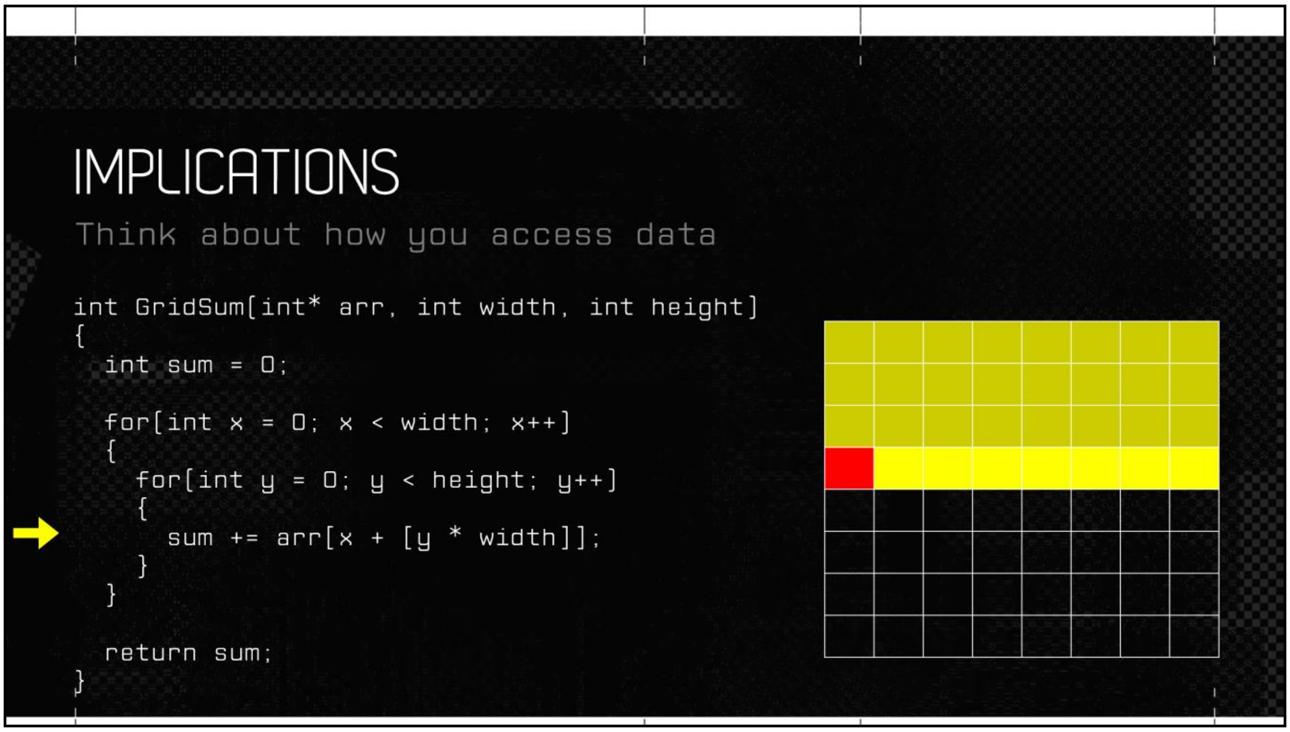


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

…and again…



IMPLICATIONS

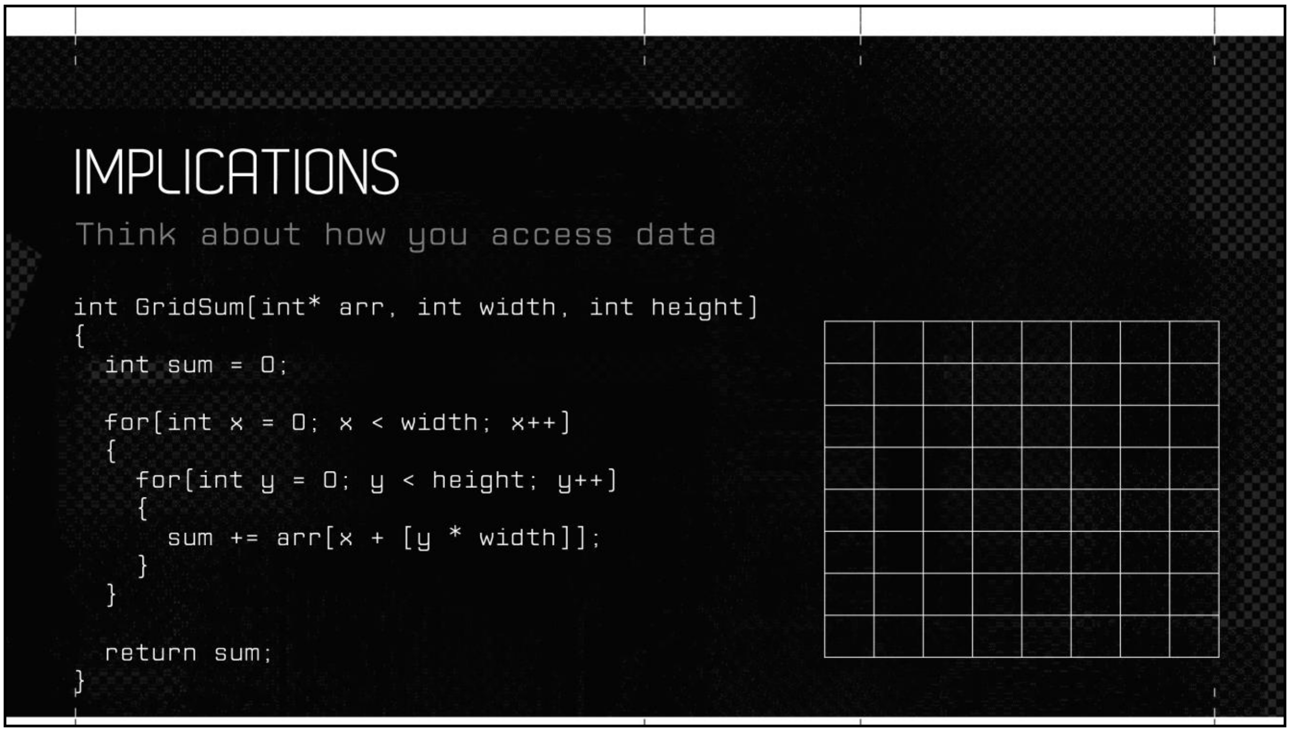
Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

…and so on.

If the data is large vs. cache size, you may completely fill the cache before ending the first loop - 0% cache utilization!

Performance will be dominated by memory fetch speed – slow.

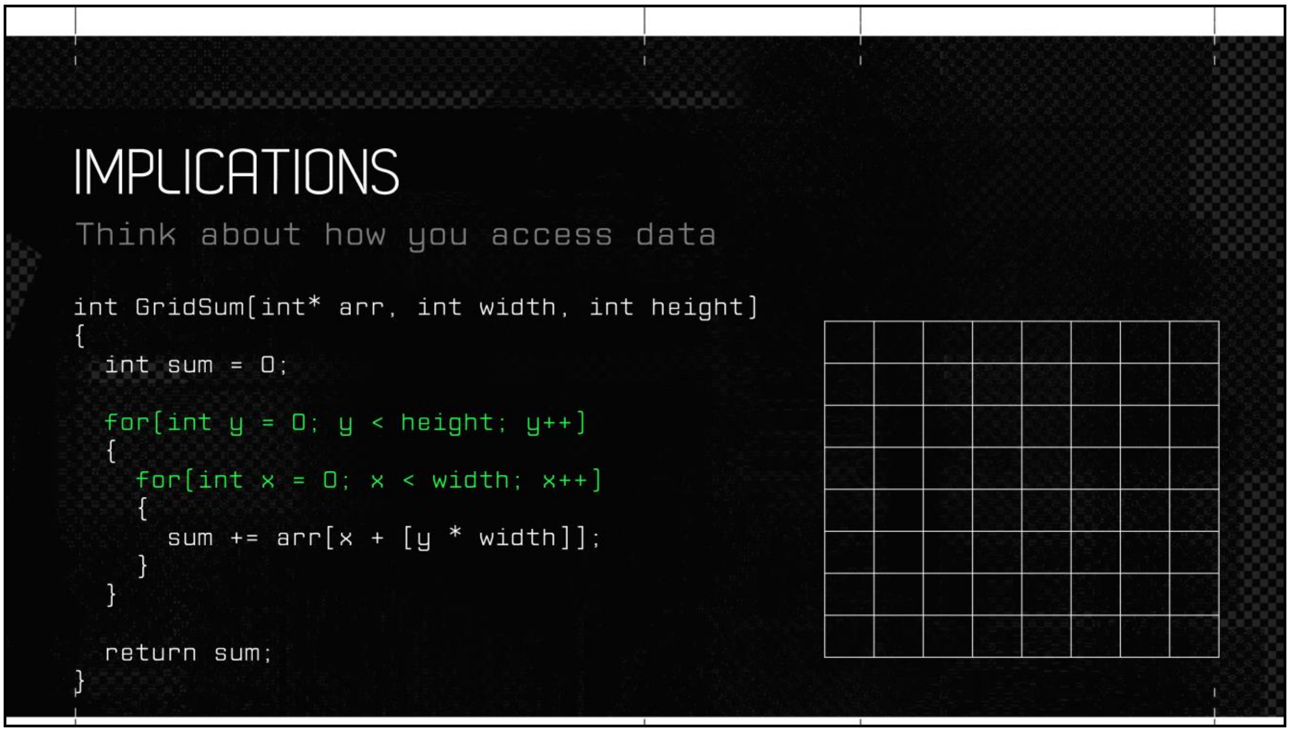


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int x = 0; x < width; x++)
  {
    for(int y = 0; y < height; y++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Simple improvement…

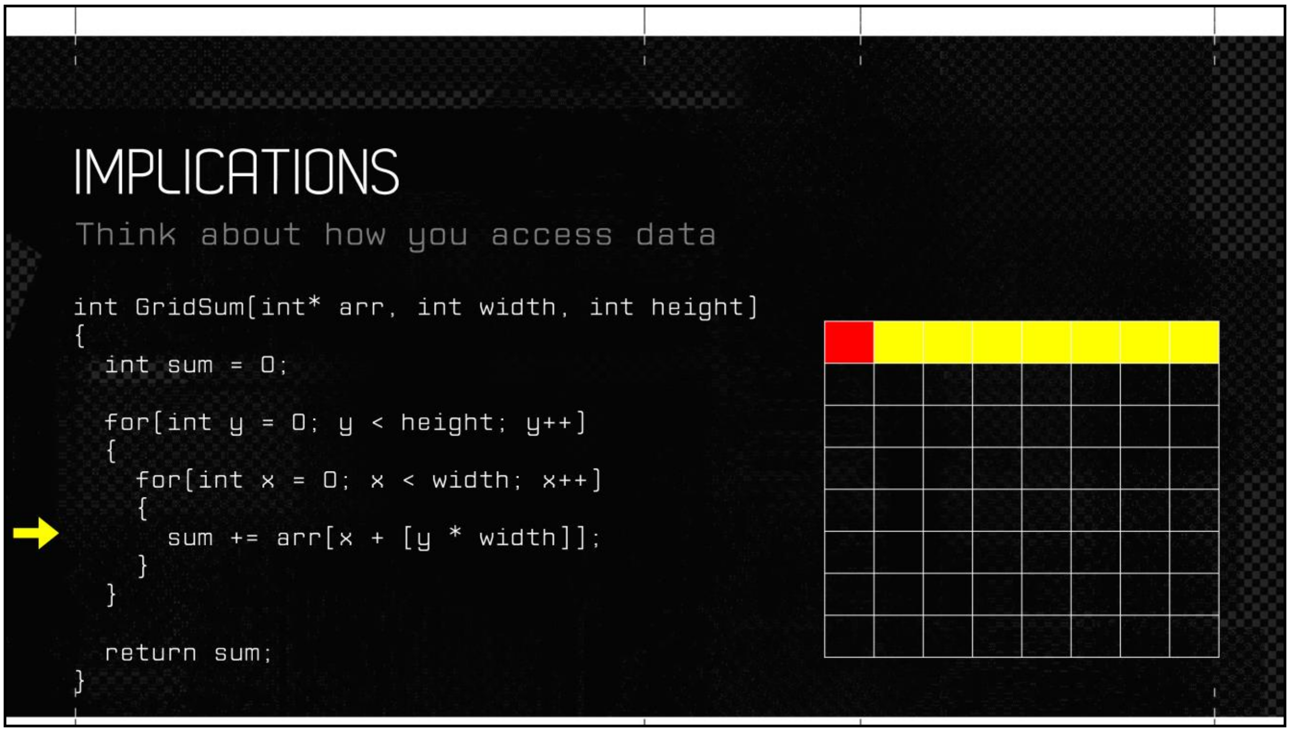


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Switch inner loops!

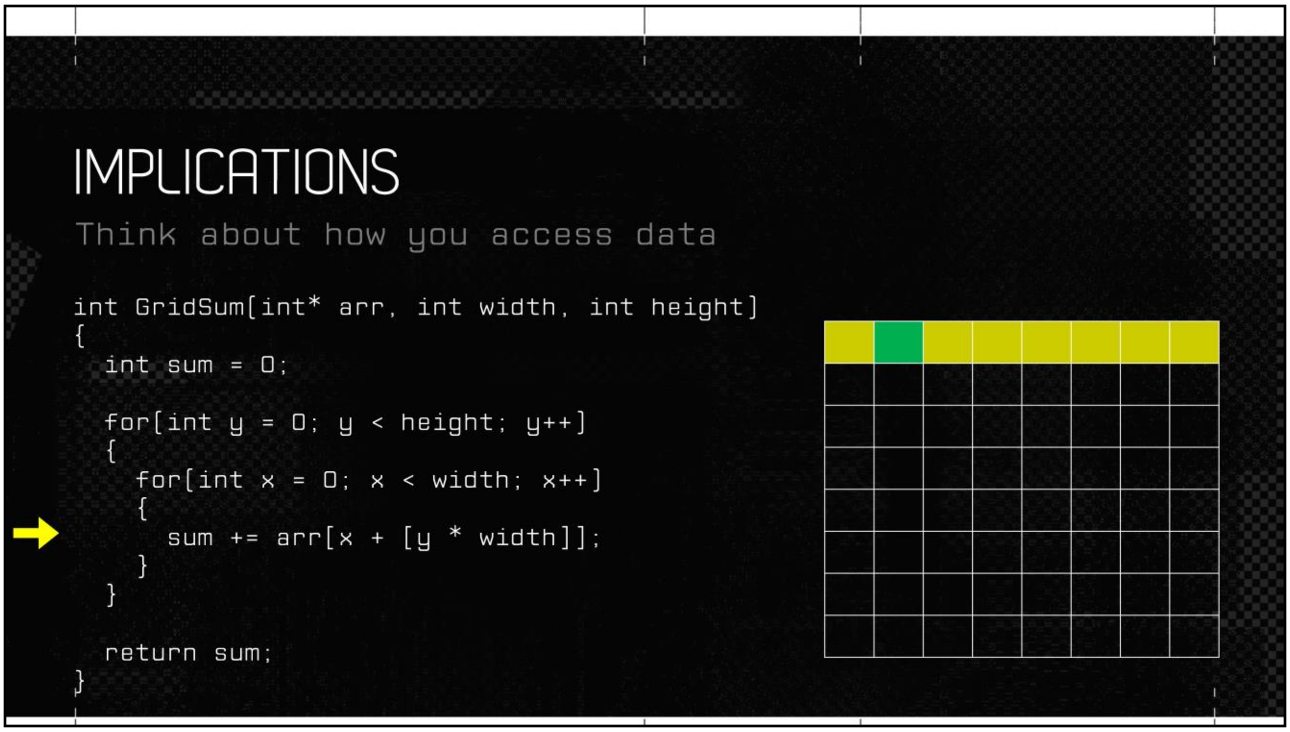


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Now one cache miss…

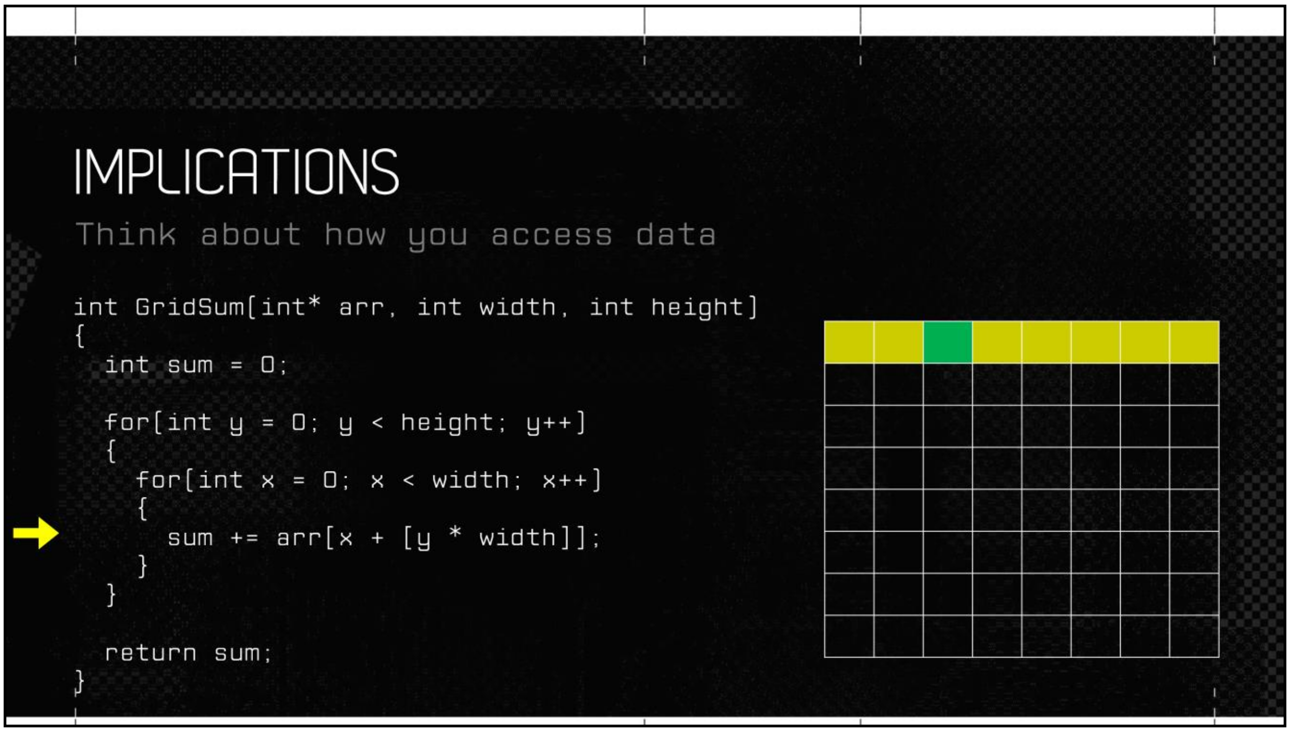


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

…followed by multiple cache hits.

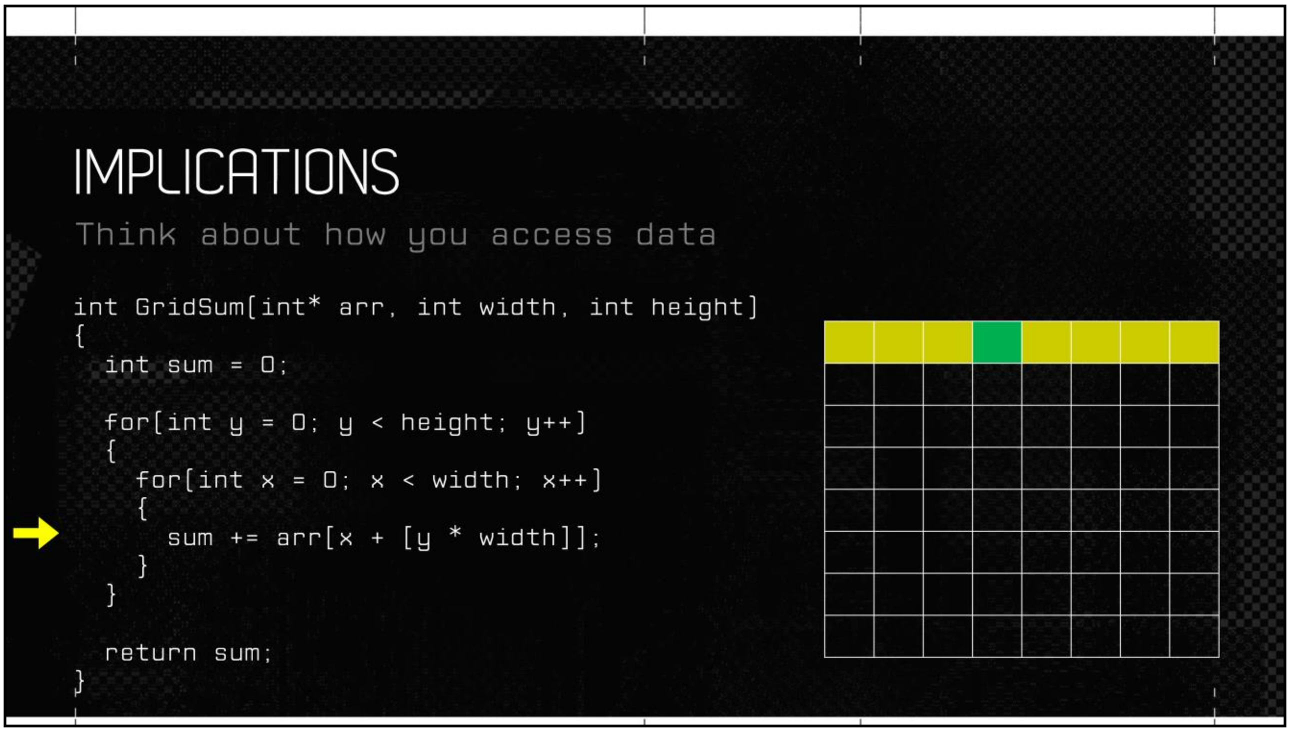


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

…followed by multiple cache hits.

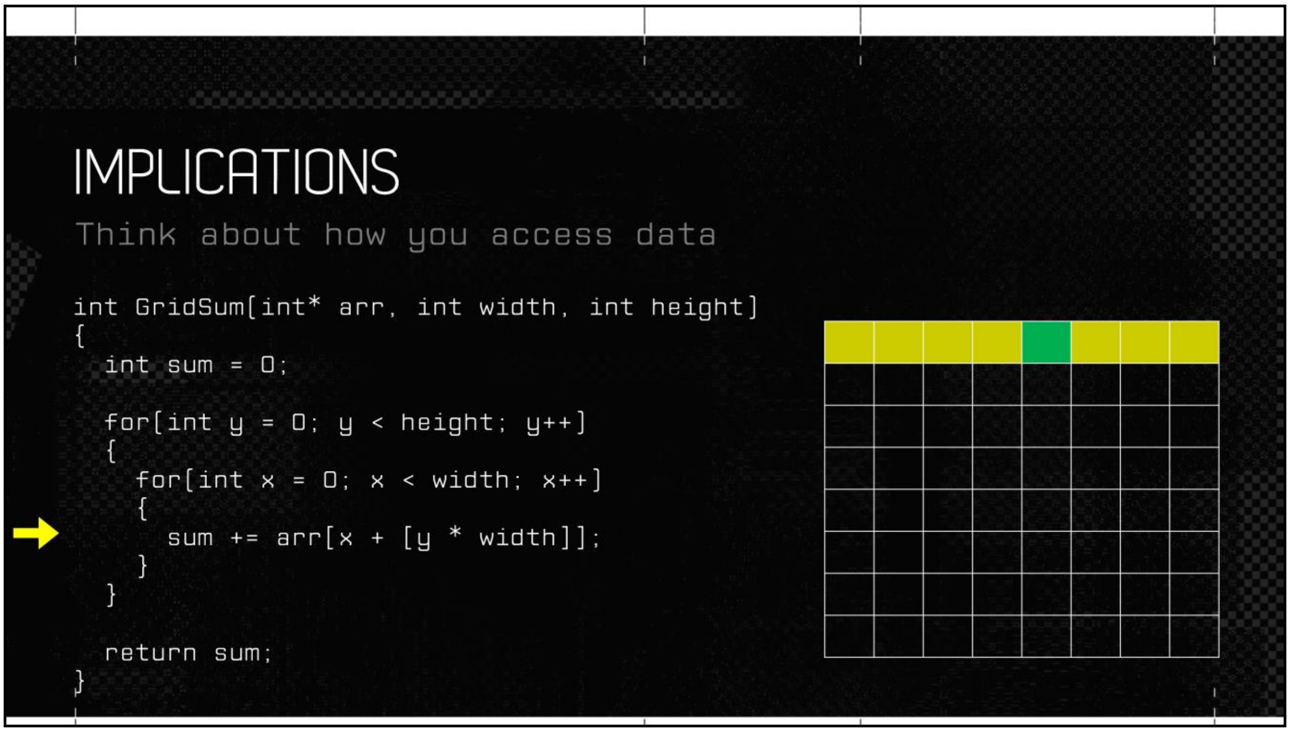


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

…followed by multiple cache hits.

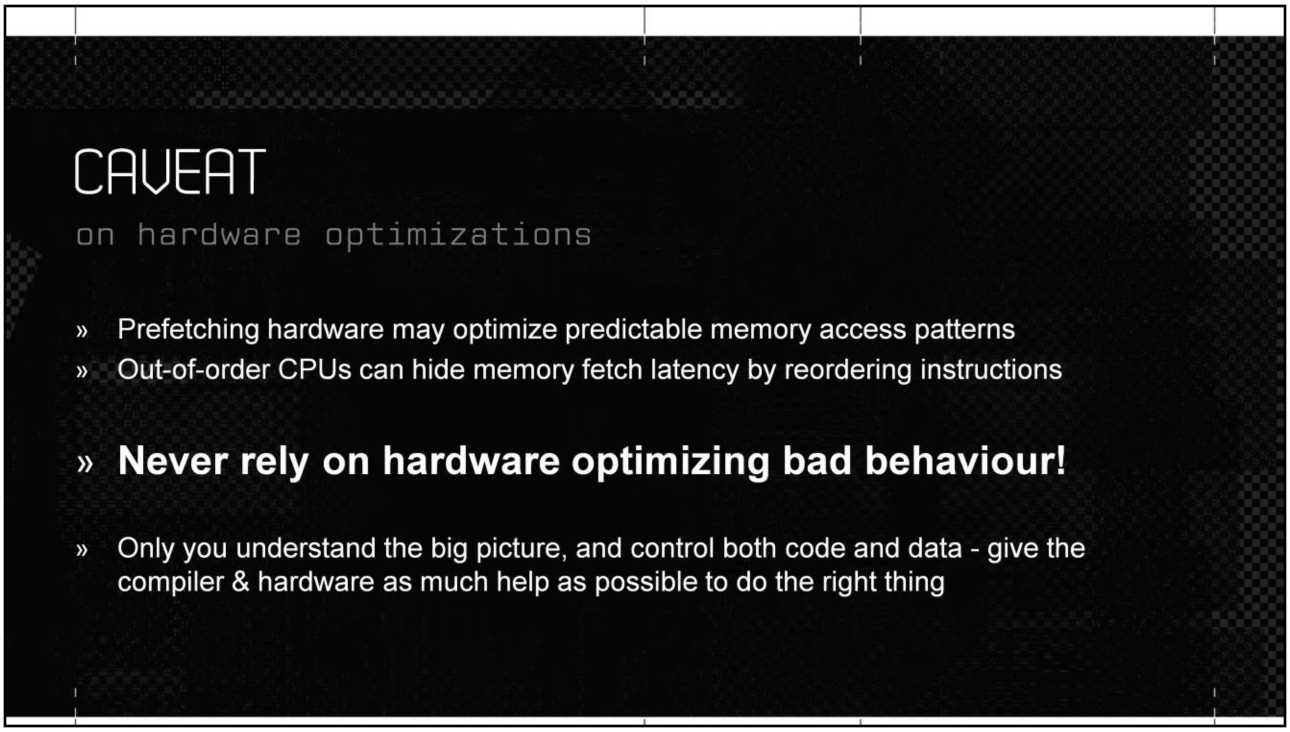


IMPLICATIONS

Think about how you access data

```
int GridSum(int* arr, int width, int height)
{
  int sum = 0;

  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      sum += arr[x + (y * width)];
    }
  }

  return sum;
}
```

Win!



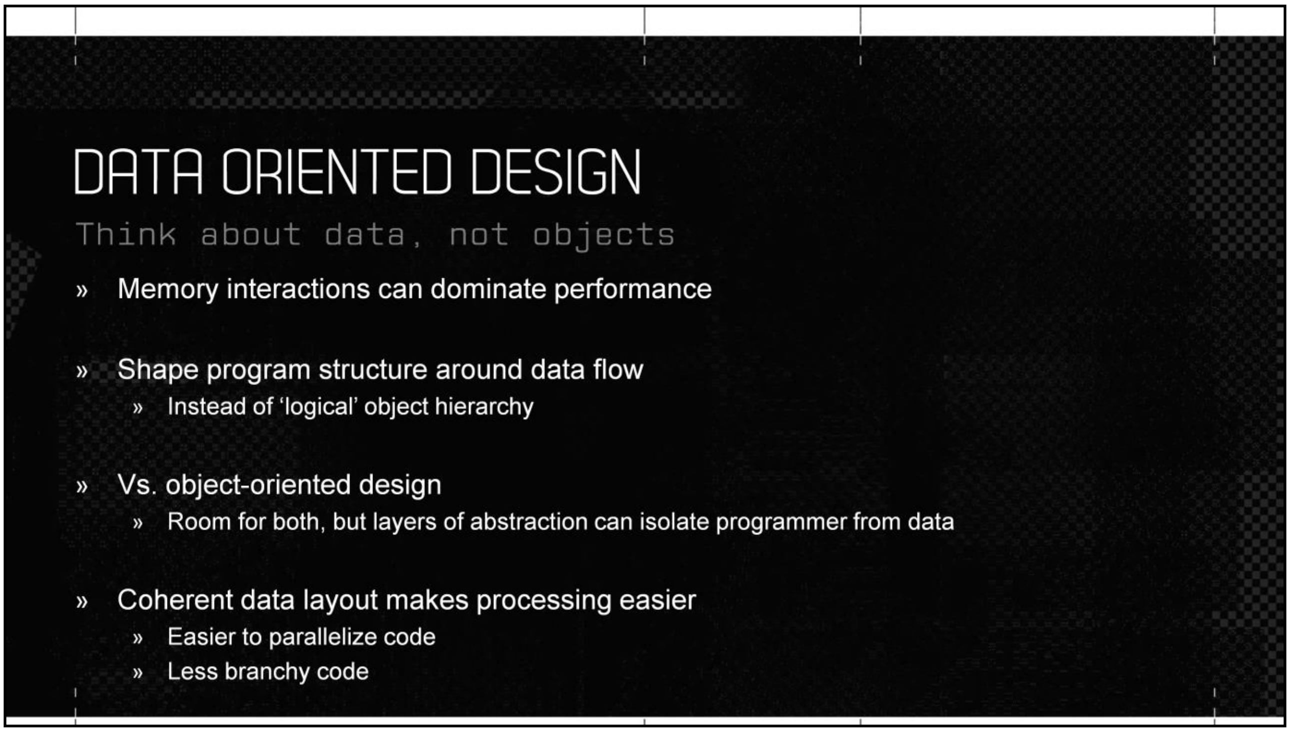
CAVEAT

on hardware optimizations

- » Prefetching hardware may optimize predictable memory access patterns
- » Out-of-order CPUs can hide memory fetch latency by reordering instructions

- » **Never rely on hardware optimizing bad behaviour!**

- » Only you understand the big picture, and control both code and data - give the compiler & hardware as much help as possible to do the right thing



DATA ORIENTED DESIGN

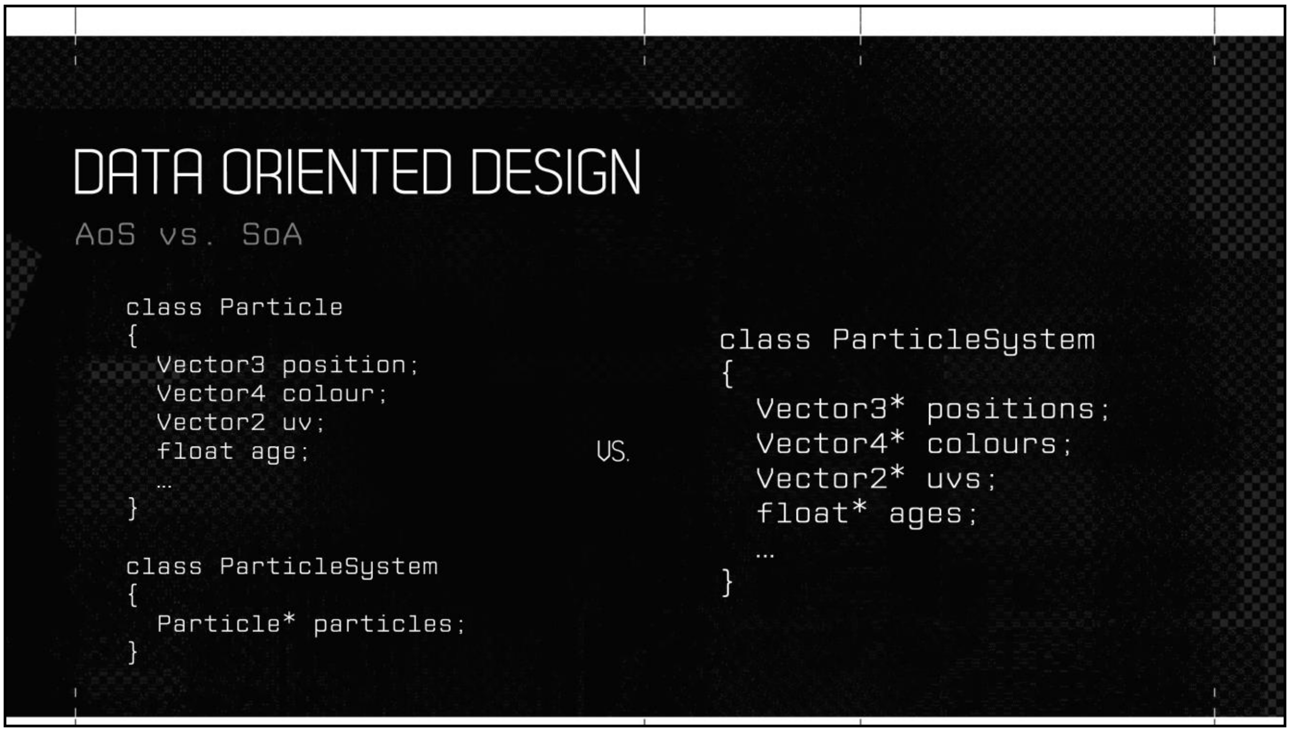
Think about data, not objects

- Memory interactions can dominate performance
- Shape program structure around data flow
  - Instead of 'logical' object hierarchy
- Vs. object-oriented design
  - Room for both, but layers of abstraction can isolate programmer from data
- Coherent data layout makes processing easier
  - Easier to parallelize code
  - Less branchy code

More generally… Data Oriented Design

Realities of memory fetch speeds necessitate a change in thinking

Think about your data and access patterns, not necessarily the 'logical' structure of objects & hierarchies



DATA ORIENTED DESIGN

AoS vs. SoA

```
class Particle
{
  Vector3 position;
  Vector4 colour;
  Vector2 uv;
  float age;
  …
}

class ParticleSystem
{
  Particle* particles;
}
```

VS.

```
class ParticleSystem
{
  Vector3* positions;
  Vector4* colours;
  Vector2* uvs;
  float* ages;
  …
}
```

Simple example: particle systems

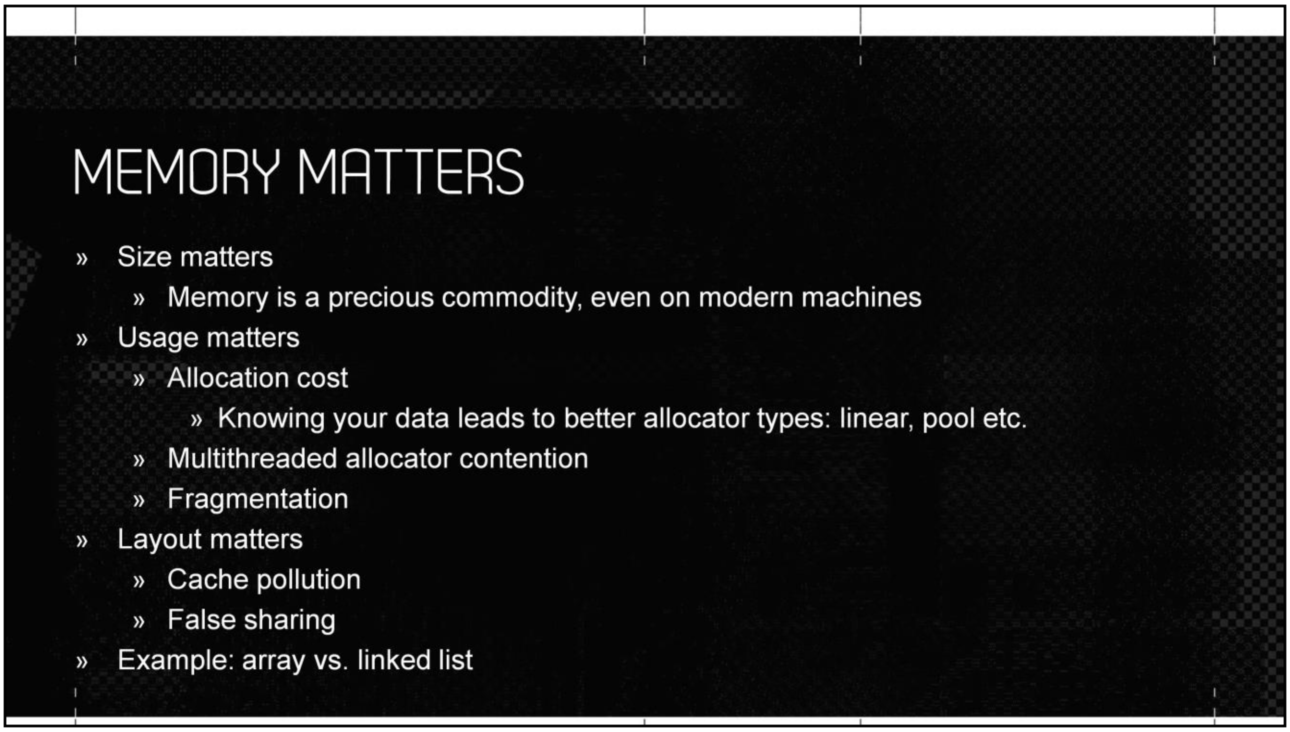
Naïve OO design says the action happens per-particle, so a particle should be an object

But when do we ever work with just a single particle? Consider…

- Per-frame updates (position etc.)
- Building vertex buffers
- Processing dead particles

This is AoS vs. SoA – Arrays of Structures vs Structures of Arrays

Much more to be said about DoD – see references at end (Mike Acton, Tony Albrecht)



MEMORY MATTERS

- Size matters
  - Memory is a precious commodity, even on modern machines
- Usage matters
  - Allocation cost
    - Knowing your data leads to better allocator types: linear, pool etc.
  - Multithreaded allocator contention
  - Fragmentation
- Layout matters
  - Cache pollution
  - False sharing
- Example: array vs. linked list

Overall: memory is an important factor, and always needs to be considered.

Easy to ignore memory if learning gamedev on PC – you're in for a rude awakening.

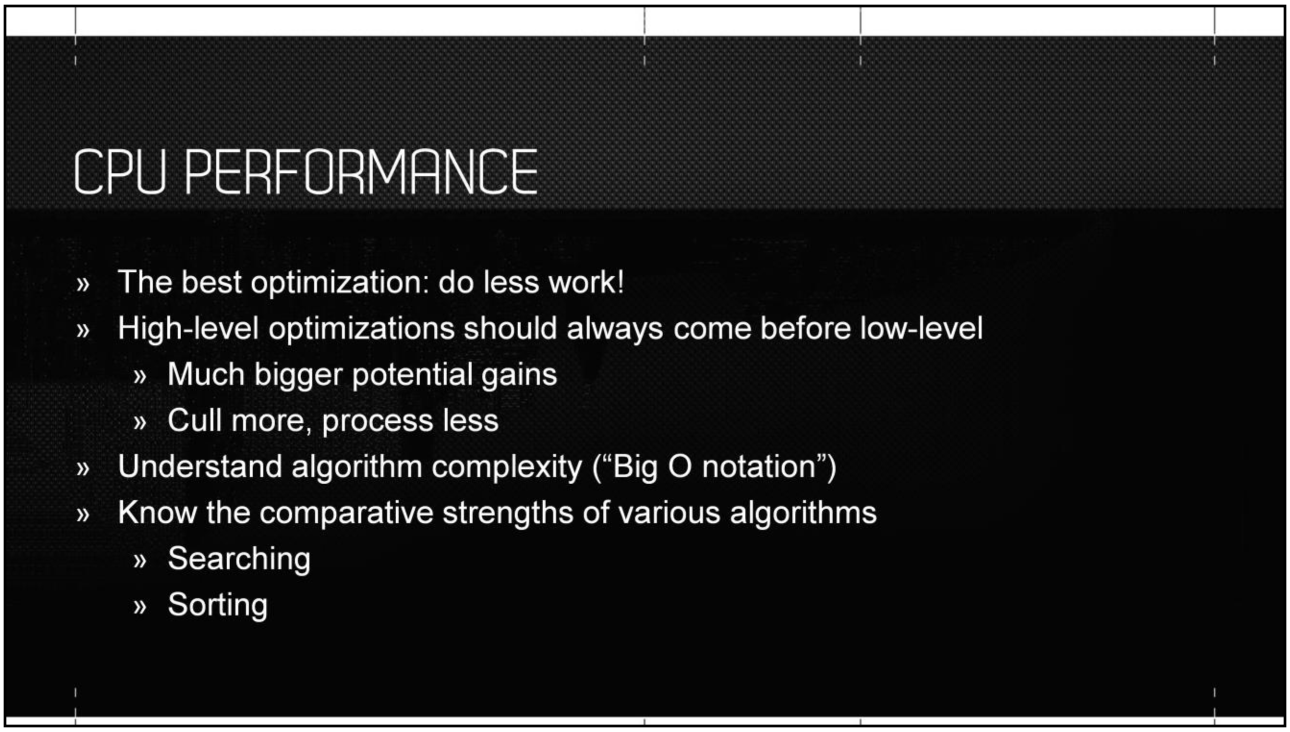
Fragmentation: small heap allocations of varying lifetimes leads to a 'swiss cheese' effect, where there is a lot of free memory available, but in individual chunks too small to be useful.

Cache pollution: pulling unnecessary/unused data into cache, possibly evicting other useful data that then needs to be refetched from memory. Group commonly-used data together in structures, and consider splitting large structures into smaller cacheline-sized structures based on usage.

False sharing: in a multi-processor system, accessing adjacent data form different threads can cause that data to be refetched from memory every time unnecessarily, because both pieces of data would be stored in the same cacheline and so flushed whenever any changes are made by another processor.

All important when deciding on storage structures. Eg: array vs. linked list – very different allocation, access, and cache usage patterns.

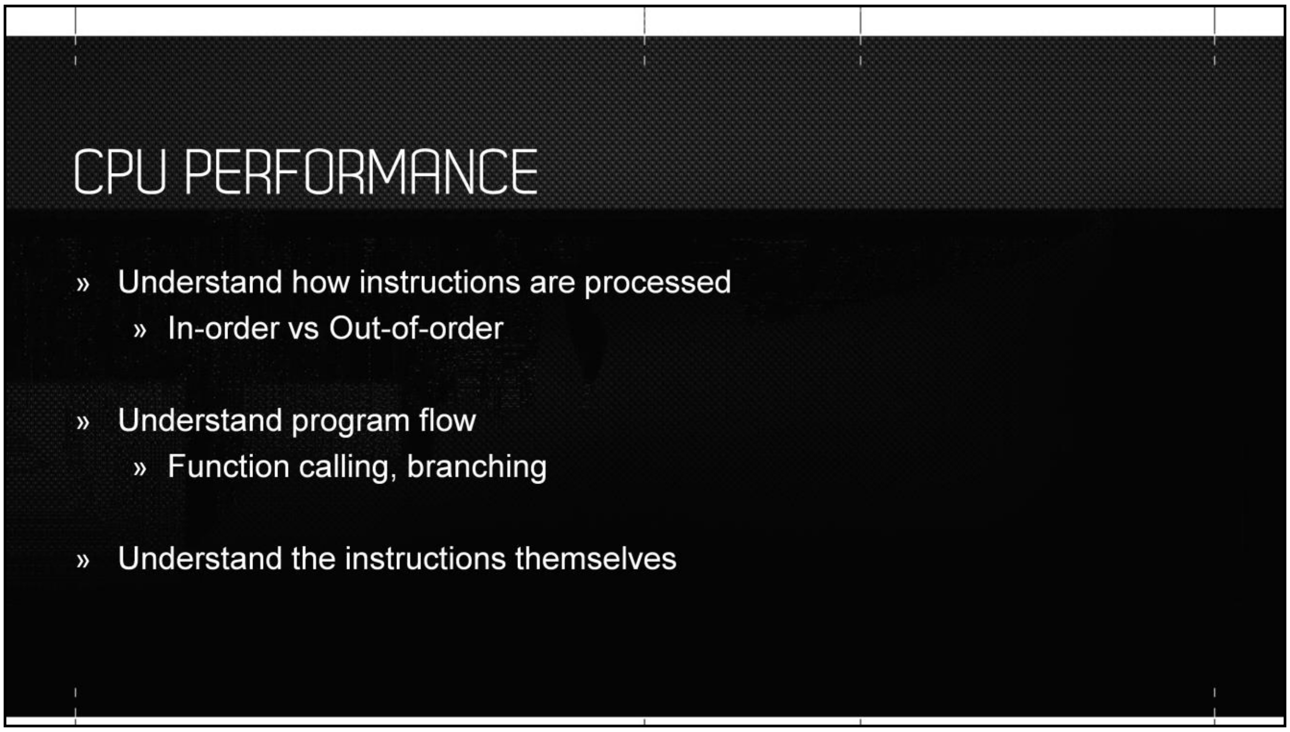
CPU



CPU PERFORMANCE

- The best optimization: do less work!
- High-level optimizations should always come before low-level
  - Much bigger potential gains
  - Cull more, process less
- Understand algorithm complexity (“Big O notation”)
- Know the comparative strengths of various algorithms
  - Searching
  - Sorting

See bigocheatsheet.com



CPU PERFORMANCE

- » Understand how instructions are processed
  - » In-order vs Out-of-order
- » Understand program flow
  - » Function calling, branching
- » Understand the instructions themselves

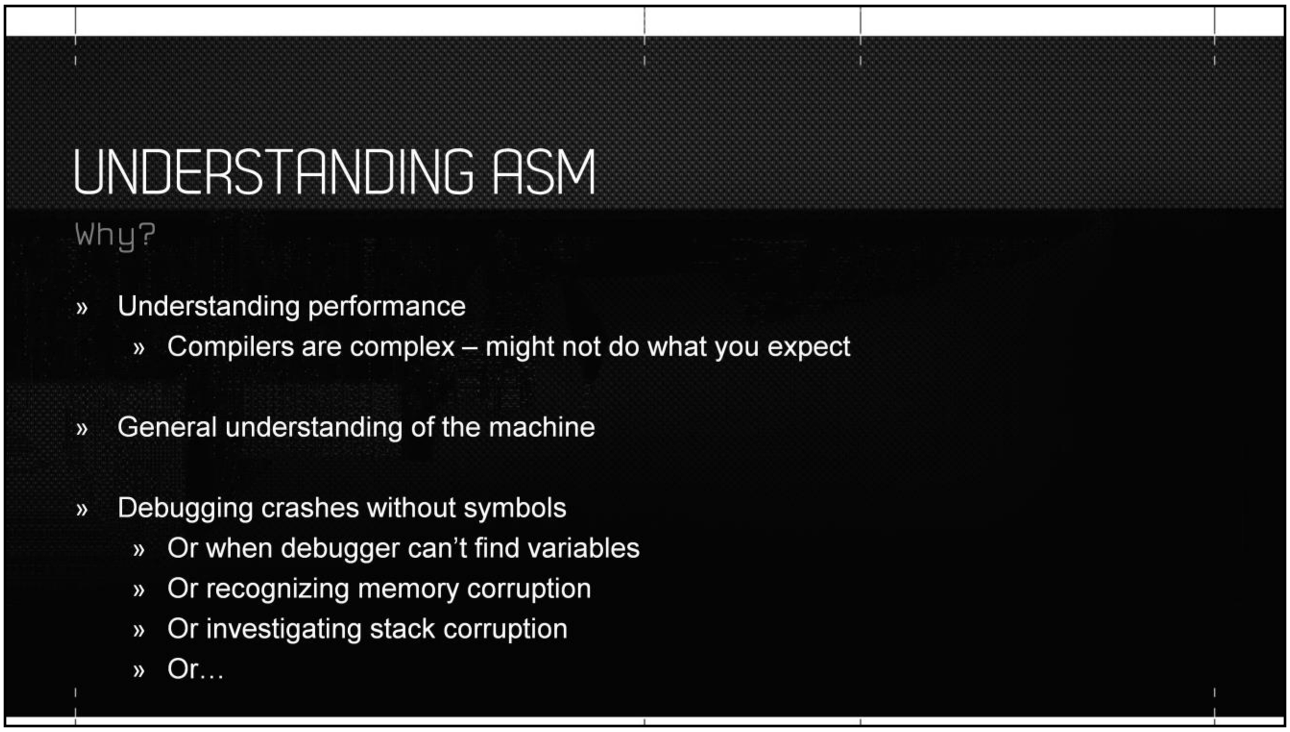
To optimize for the CPU, you need to understand what it's doing in the first place.

In-order processors subject to LHS (Load-Hit-Store), where a piece of data is written to memory and then immediately read back, causing a stall.

Out-of-order can process other independent instructions while waiting for data, possibly hiding the stall with other work. But also less predictable.

Functions: prologue/epilogue overhead; inlining. Virtual function vtable indirection cost.

Branches: prediction hardware ranges in quality, so branches can have varying costs. Potential processor pipeline flush - plan accordingly.



UNDERSTANDING ASM

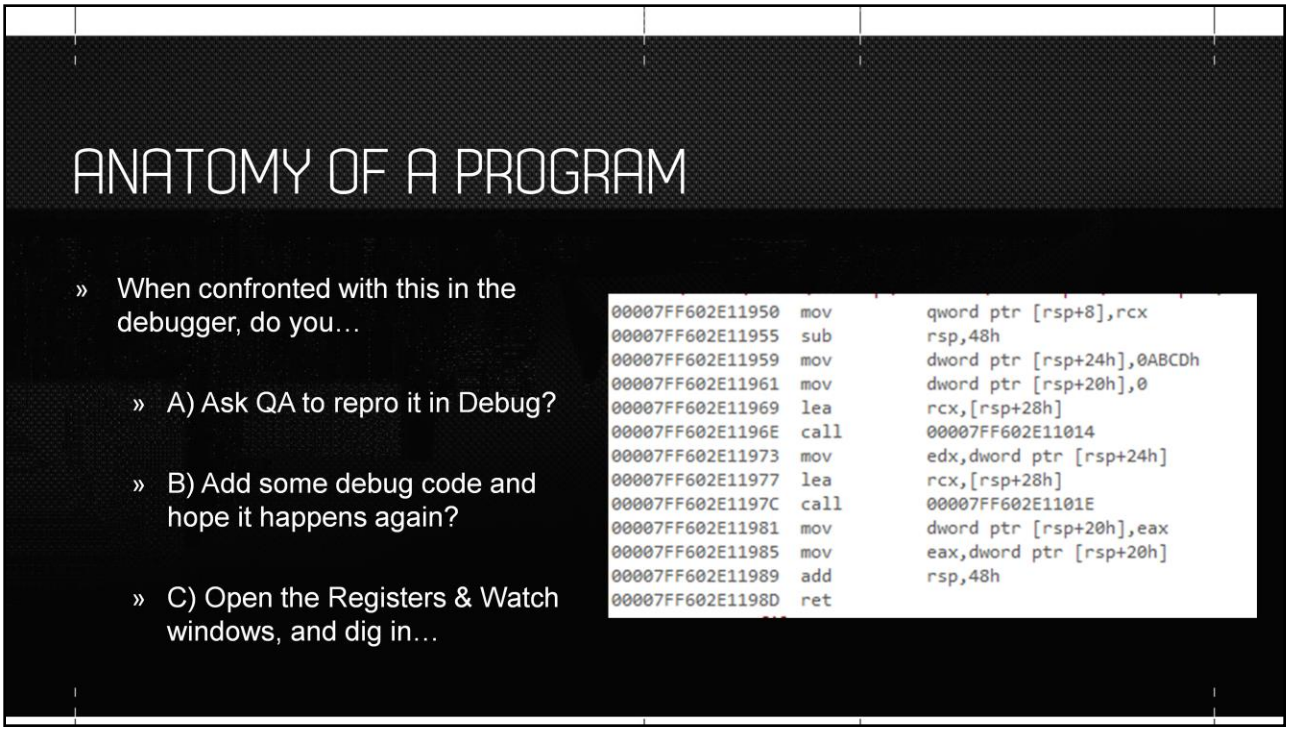
Why?

- Understanding performance
  - Compilers are complex – might not do what you expect
- General understanding of the machine
- Debugging crashes without symbols
  - Or when debugger can't find variables
  - Or recognizing memory corruption
  - Or investigating stack corruption
  - Or…

Aside: understanding assembly is essential for understanding performance, but it's even more important than that

ASM instructions are the building blocks of all the code you write. Every coder should be able to at least read & follow them.

Don't just leave the understanding to "the low-level engine coders".



ANATOMY OF A PROGRAM

- When confronted with this in the debugger, do you…
  - A) Ask QA to repro it in Debug?
  - B) Add some debug code and hope it happens again?
  - C) Open the Registers & Watch windows, and dig in…

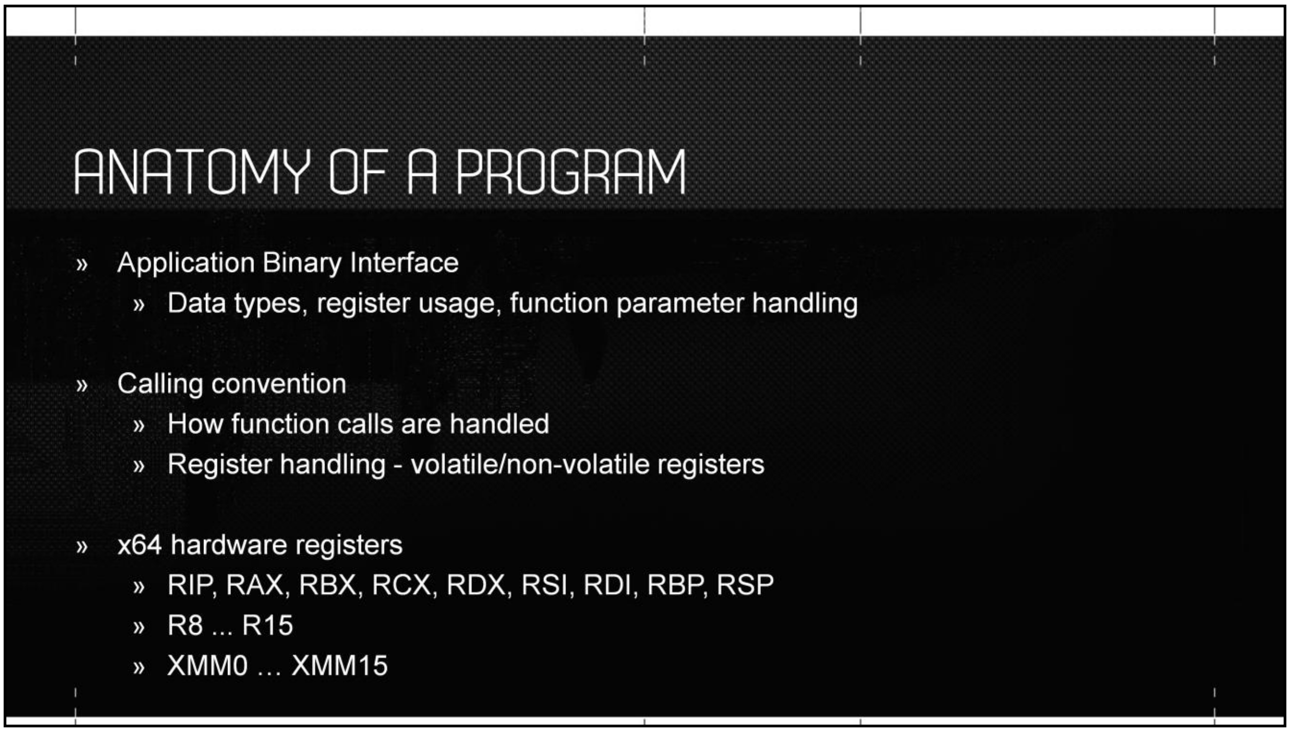
```
00007FF602E11950  mov         qword ptr [rsp+8],rcx
00007FF602E11955  sub         rsp,48h
00007FF602E11959  mov         dword ptr [rsp+24h],0ABCDh
00007FF602E11961  mov         dword ptr [rsp+20h],0
00007FF602E11969  lea         rcx,[rsp+28h]
00007FF602E1196E  call        00007FF602E11014
00007FF602E11973  mov         edx,dword ptr [rsp+24h]
00007FF602E11977  lea         rcx,[rsp+28h]
00007FF602E1197C  call        00007FF602E1101E
00007FF602E11981  mov         dword ptr [rsp+20h],eax
00007FF602E11985  mov         eax,dword ptr [rsp+20h]
00007FF602E11989  add         rsp,48h
00007FF602E1198D  ret
```

Especially towards the end of production, you'll inevitably encounter rare, unreproducible crashes

One crash in 1000 hours of gameplay sounds rare? 2m copies, 10 hours each… 20,000 crashes!

You have to work with what you have, when you have it.

All the information is there, you just have to know how to find it.



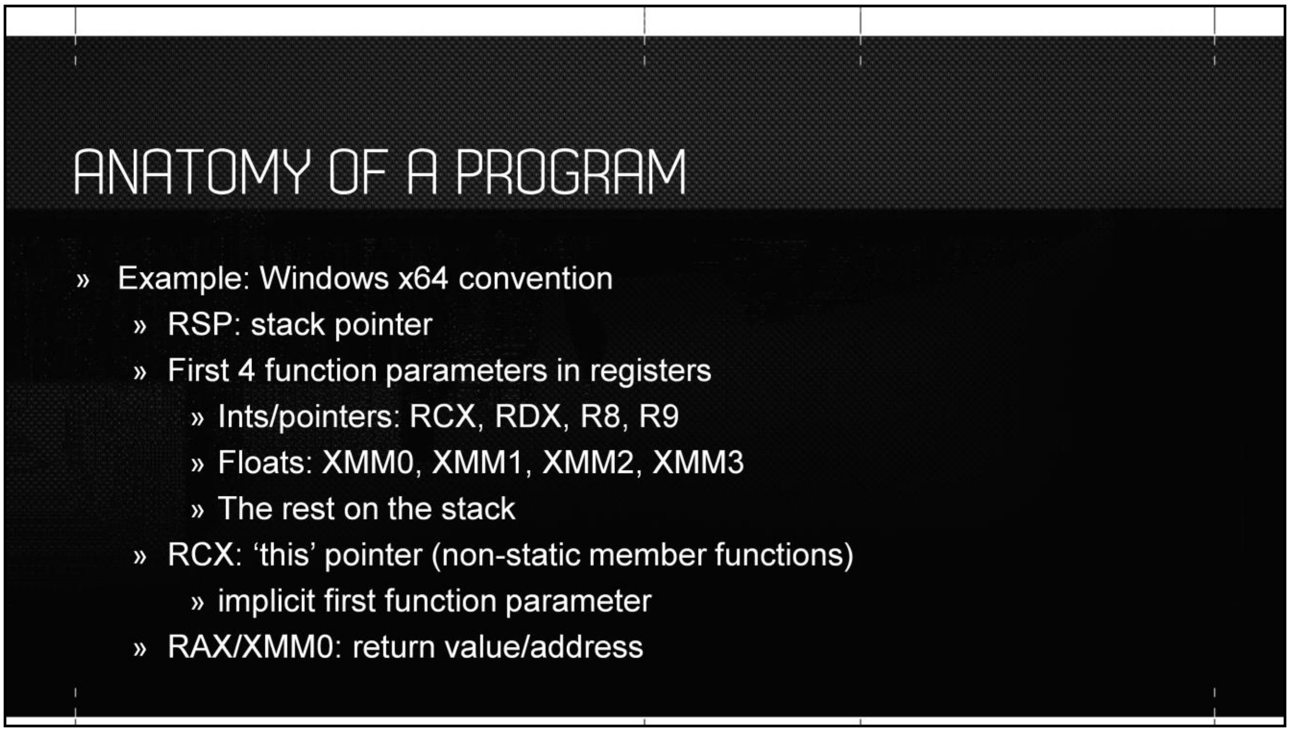
ANATOMY OF A PROGRAM

- Application Binary Interface
  - Data types, register usage, function parameter handling
- Calling convention
  - How function calls are handled
  - Register handling - volatile/non-volatile registers
- x64 hardware registers
  - RIP, RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP
  - R8 ... R15
  - XMM0 … XMM15

Volatile registers: contents may be overwritten inside a function call

Non-volatile registers: values must be saved and restored after a function call

Function prologue/epilogue: reserve stack space, save & restore non-volatile registers, etc.

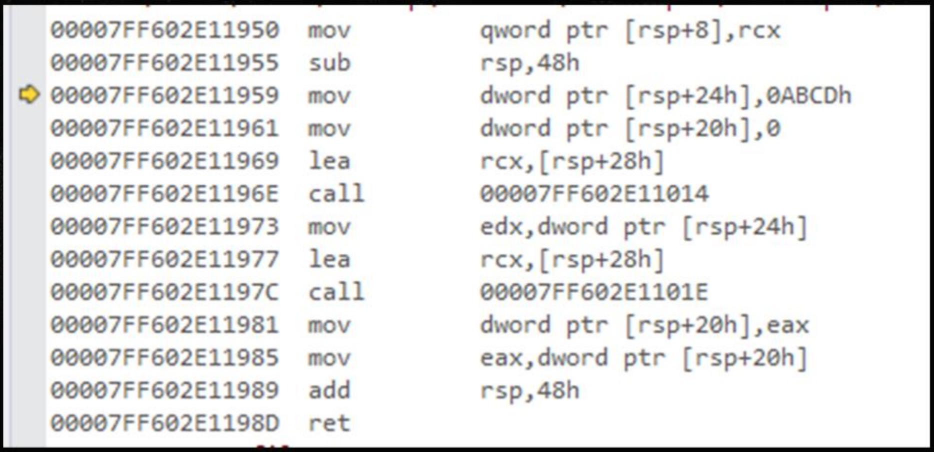


ANATOMY OF A PROGRAM

- Example: Windows x64 convention
  - RSP: stack pointer
  - First 4 function parameters in registers
    - Ints/pointers: RCX, RDX, R8, R9
    - Floats: XMM0, XMM1, XMM2, XMM3
    - The rest on the stack
  - RCX: ‘this’ pointer (non-static member functions)
    - implicit first function parameter
  - RAX/XMM0: return value/address

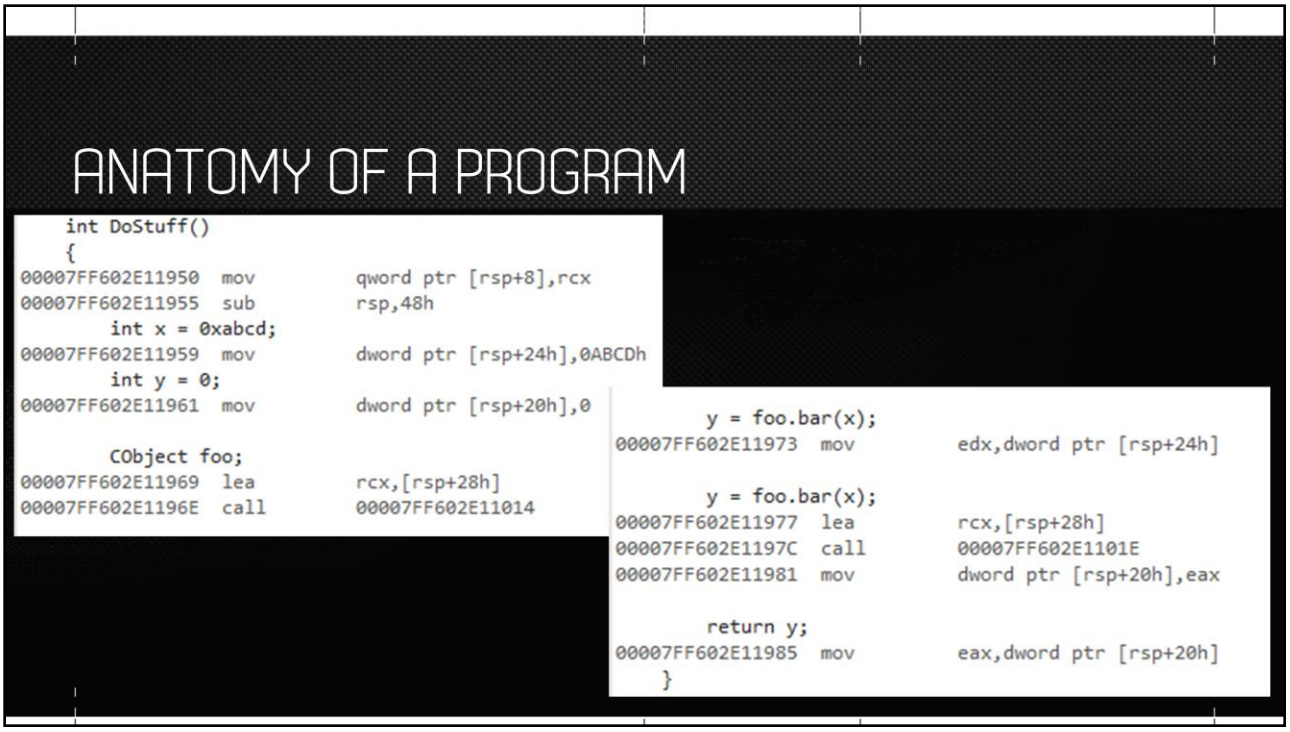
Some basics of x64 hardware register usage and the Windows x64 calling convention

ANATOMY OF A PROGRAM



```
00007FF602E11950  mov          qword ptr [rsp+8],rcx
00007FF602E11955  sub          rsp,48h
00007FF602E11959  mov          dword ptr [rsp+24h],0ABCDh
00007FF602E11961  mov          dword ptr [rsp+20h],0
00007FF602E11969  lea          rcx,[rsp+28h]
00007FF602E1196E  call         00007FF602E11014
00007FF602E11973  mov          edx,dword ptr [rsp+24h]
00007FF602E11977  lea          rcx,[rsp+28h]
00007FF602E1197C  call         00007FF602E1101E
00007FF602E11981  mov          dword ptr [rsp+20h],eax
00007FF602E11985  mov          eax,dword ptr [rsp+20h]
00007FF602E11989  add          rsp,48h
00007FF602E1198D  ret
```

With these few guides, this simple function example becomes a lot clearer…



ANATOMY OF A PROGRAM

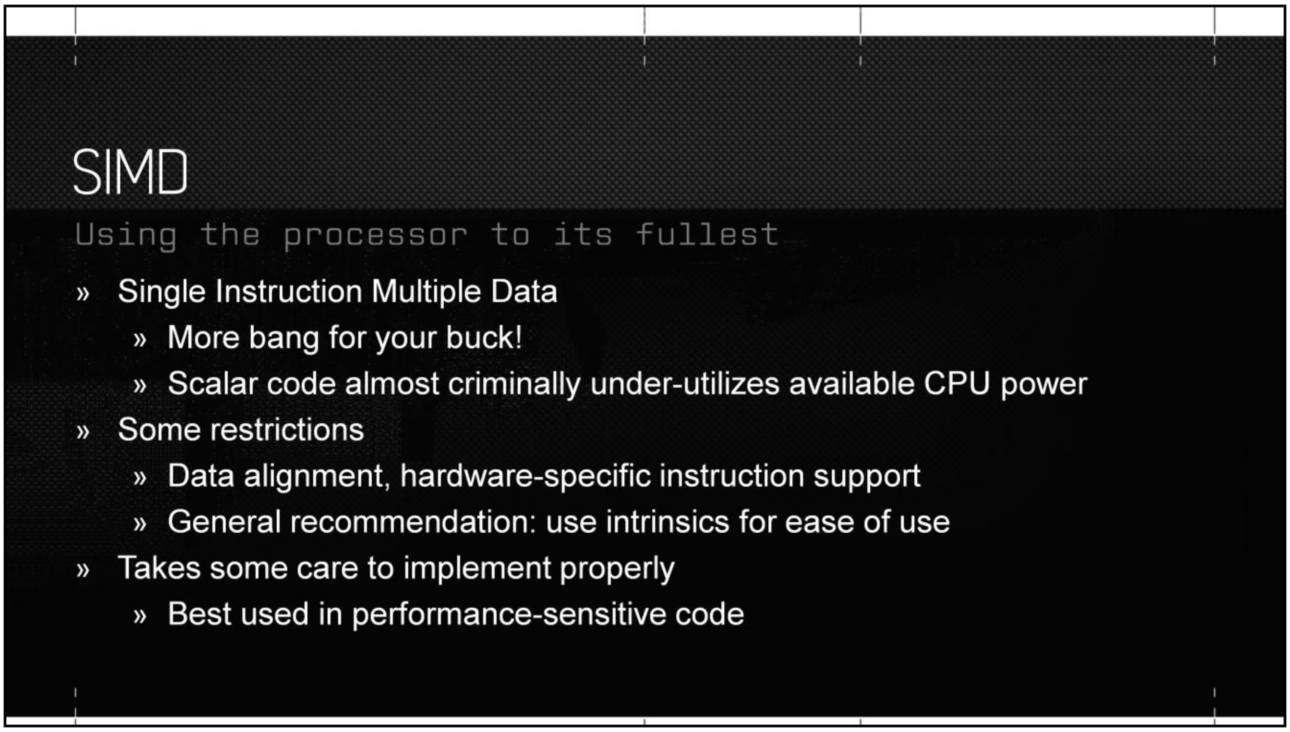
```
    int DoStuff()
    {
00007FF602E11950  mov         qword ptr [rsp+8],rcx
00007FF602E11955  sub         rsp,48h
        int x = 0xabcd;
00007FF602E11959  mov         dword ptr [rsp+24h],0ABCDh
        int y = 0;
00007FF602E11961  mov         dword ptr [rsp+20h],0

        CObject foo;
00007FF602E11969  lea         rcx,[rsp+28h]
00007FF602E1196E  call        00007FF602E11014
```

```
        y = foo.bar(x);
00007FF602E11973  mov         edx,dword ptr [rsp+24h]

        y = foo.bar(x);
00007FF602E11977  lea         rcx,[rsp+28h]
00007FF602E1197C  call        00007FF602E1101E
00007FF602E11981  mov         dword ptr [rsp+20h],eax

        return y;
00007FF602E11985  mov         eax,dword ptr [rsp+20h]
    }
```

Some coders have an irrational fear of assembly. It’s actually very straightforward! Understanding is liberating.

Spend a few hours studying the details and exploring your own programs in the debugger, it will pay off many times over.

See Elan Ruskin’s excellent GDC talk (see references) for more.



SIMD

Using the processor to its fullest

- » Single Instruction Multiple Data
  - » More bang for your buck!
  - » Scalar code almost criminally under-utilizes available CPU power
- » Some restrictions
  - » Data alignment, hardware-specific instruction support
  - » General recommendation: use intrinsics for ease of use
- » Takes some care to implement properly
  - » Best used in performance-sensitive code

Can't talk about CPU performance without mentioning SIMD

A single instruction can perform the same operation on 4 (or more) pieces of data – perfect for the vector operations common in game code

Intrinsics – compiler-specific wrapper around one or more SIMD instructions. Compiler can generally schedule more efficiently given intrinsics rather than bare intructions, and they are easier to write than inline assembly.

Low-level nature and extra restrictions means more effort required to SIMD-ize code, so only use where suitable

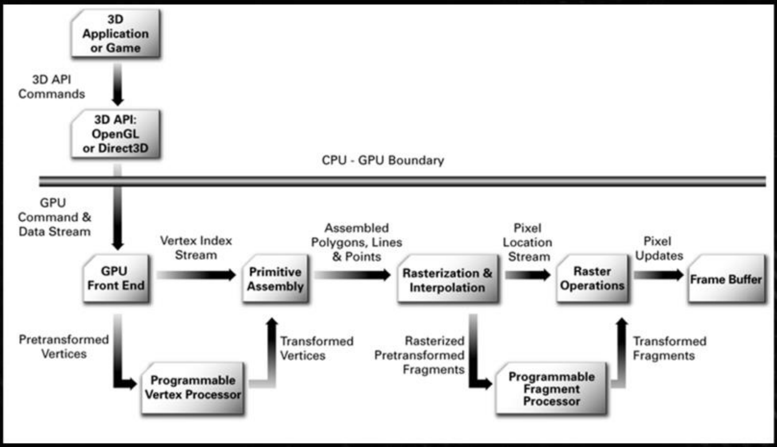
DoD approach fully applies, and code written with DoD in mind can be more easily converted to SIMD

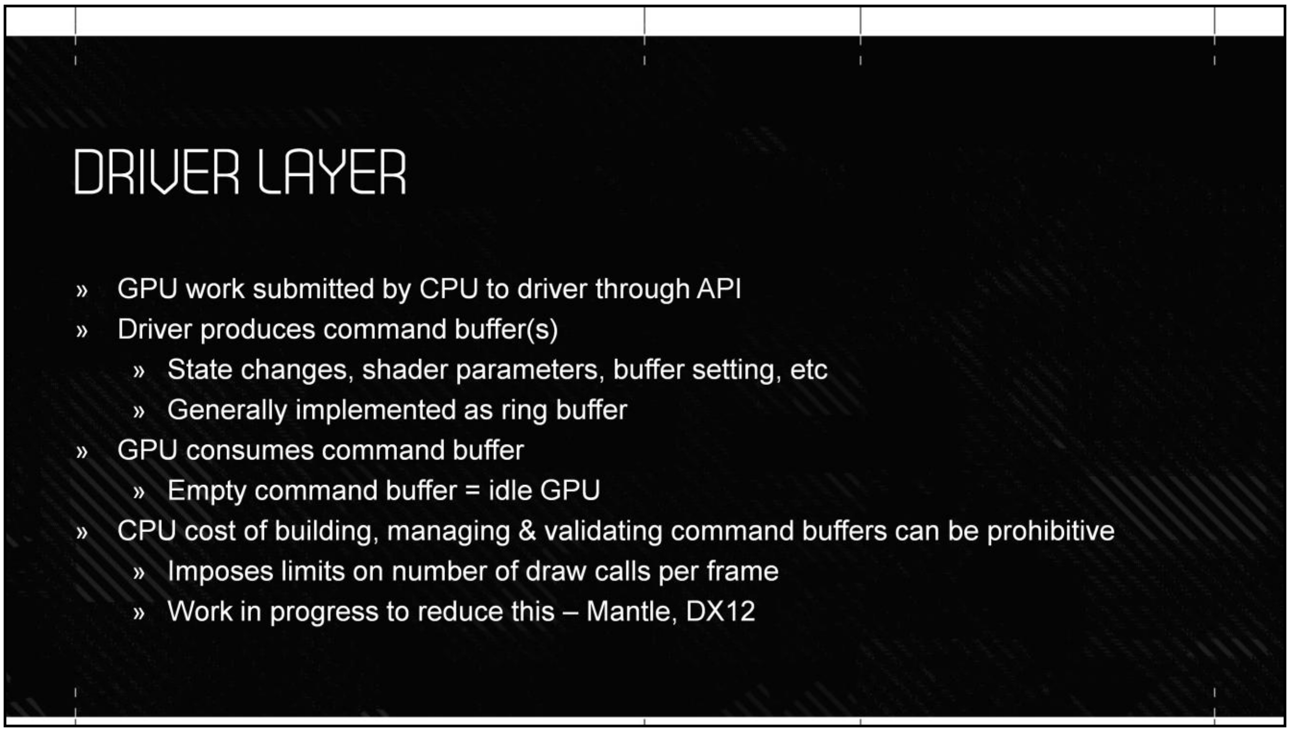
GPU

GPU performance is its own massive subject, will only touch on some high-level concerns here

GPU SYSTEM INTEGRATION

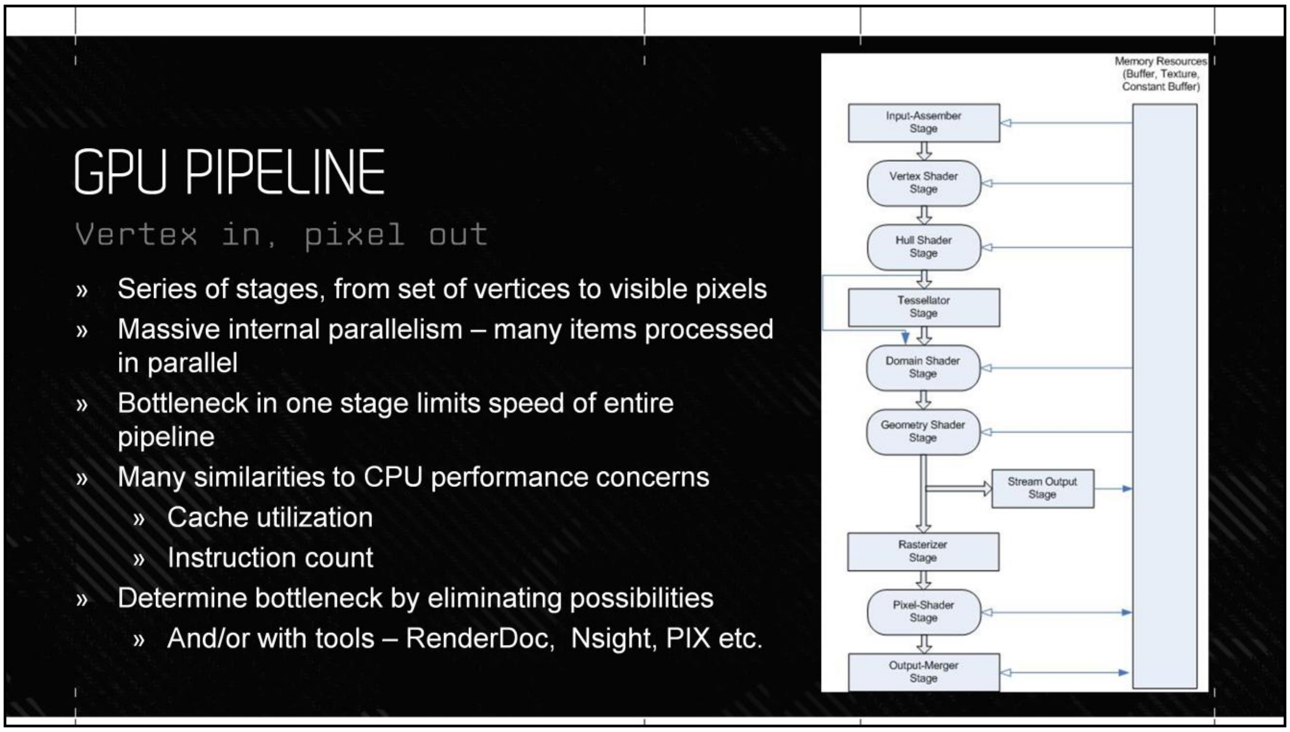
- CPU & GPU are major system components
- Closely coupled – stalling one can stall the other
- Interface through driver
  - Thin or thick





DRIVER LAYER

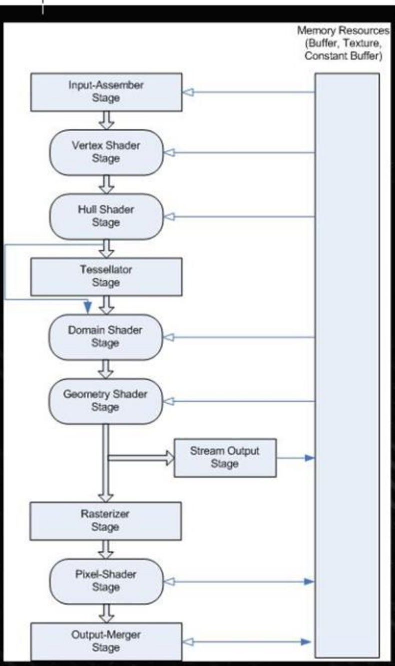
- GPU work submitted by CPU to driver through API
- Driver produces command buffer(s)
  - State changes, shader parameters, buffer setting, etc
  - Generally implemented as ring buffer
- GPU consumes command buffer
  - Empty command buffer = idle GPU
- CPU cost of building, managing & validating command buffers can be prohibitive
  - Imposes limits on number of draw calls per frame
  - Work in progress to reduce this – Mantle, DX12



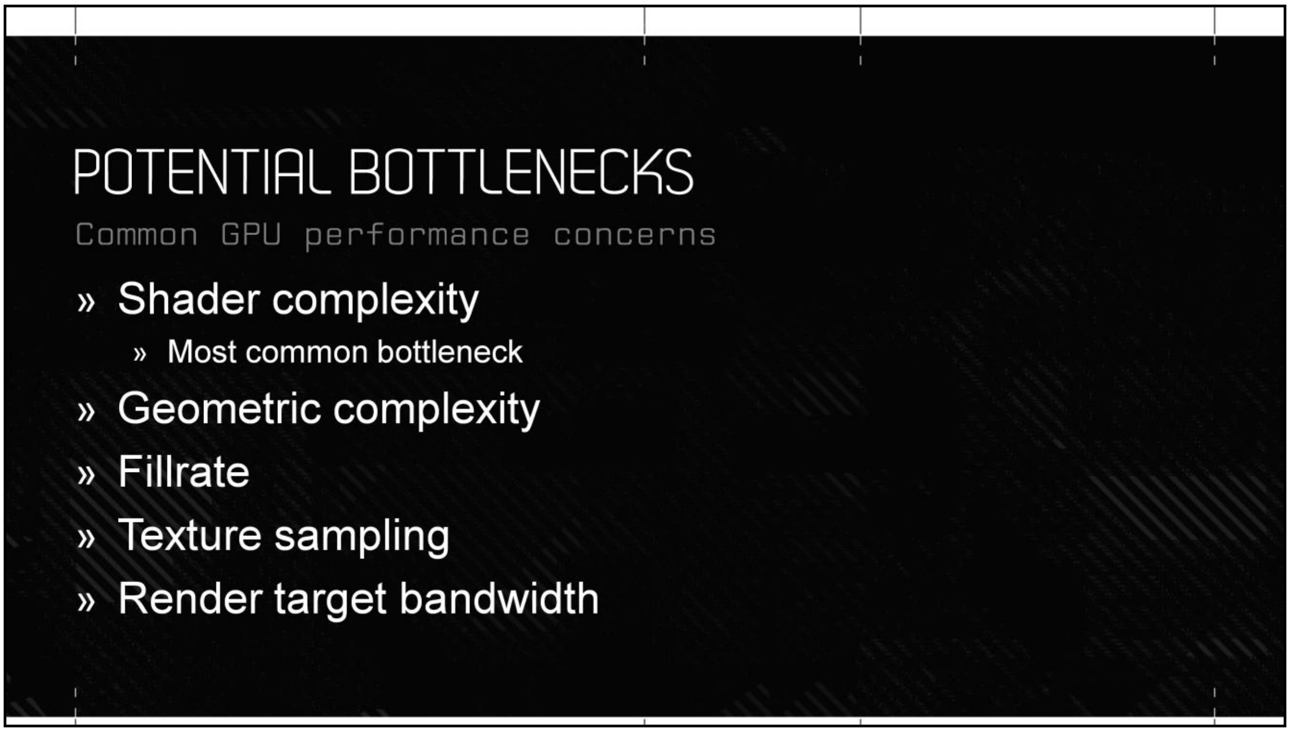
GPU PIPELINE

Vertex in, pixel out

- Series of stages, from set of vertices to visible pixels
- Massive internal parallelism – many items processed in parallel
- Bottleneck in one stage limits speed of entire pipeline
- Many similarities to CPU performance concerns
  - Cache utilization
  - Instruction count
- Determine bottleneck by eliminating possibilities
  - And/or with tools – RenderDoc, Nsight, PIX etc.



Eg. Reduce pixel shader instructions – no change in frame time means the bottleneck is elsewhere



POTENTIAL BOTTLENECKS

Common GPU performance concerns

- » Shader complexity
  - » Most common bottleneck
- » Geometric complexity
- » Fillrate
- » Texture sampling
- » Render target bandwidth

Shader complexity

Different instructions can have different cycle costs – know your architecture! See Emil Persson talks in references.

Geometry complexity

Transform cache – use indexed meshes, reorder for best cache usage (see Tom Forsyth link in references)

Triangle/pixel ratio – use LODs

Fillrate

Overdraw – Use depth buffering, early/hierarchical Z

Esp. problematic with blending

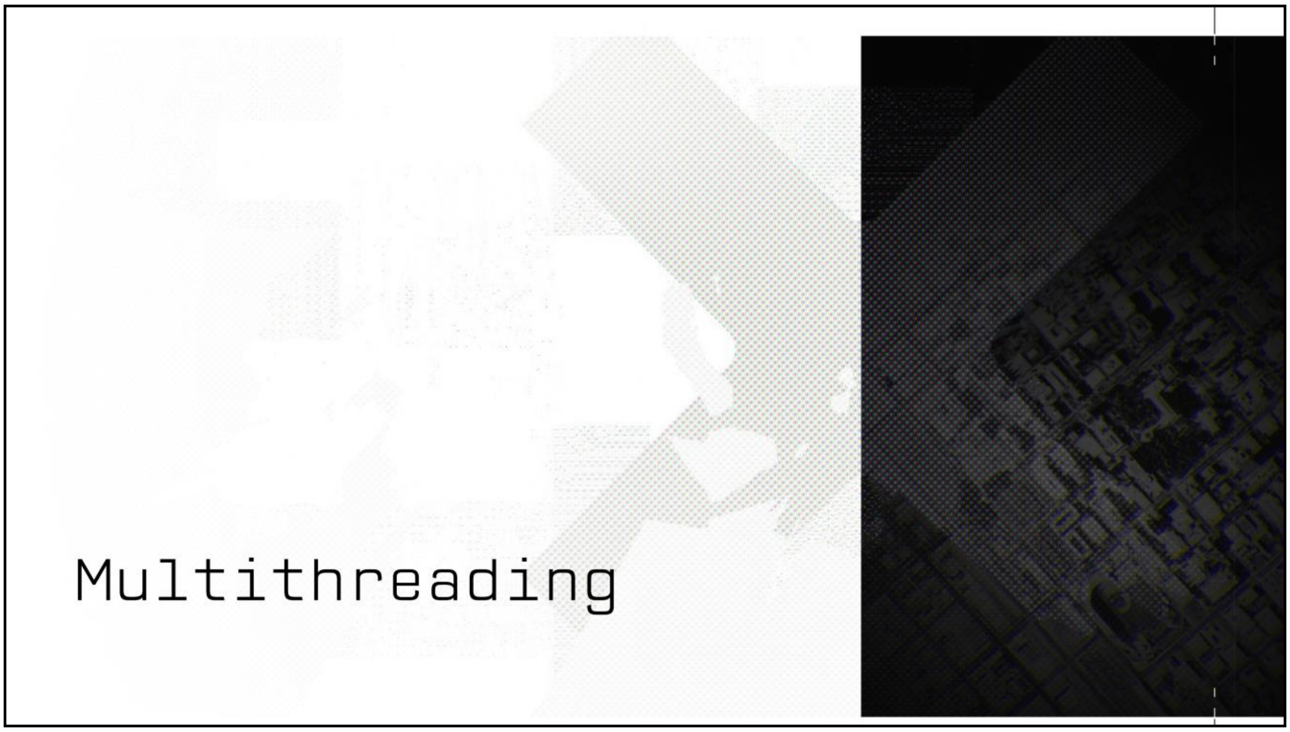
Texture sampling

Cache use - use mipmapping

Filtering cost – use mipmapping, simpler filtering

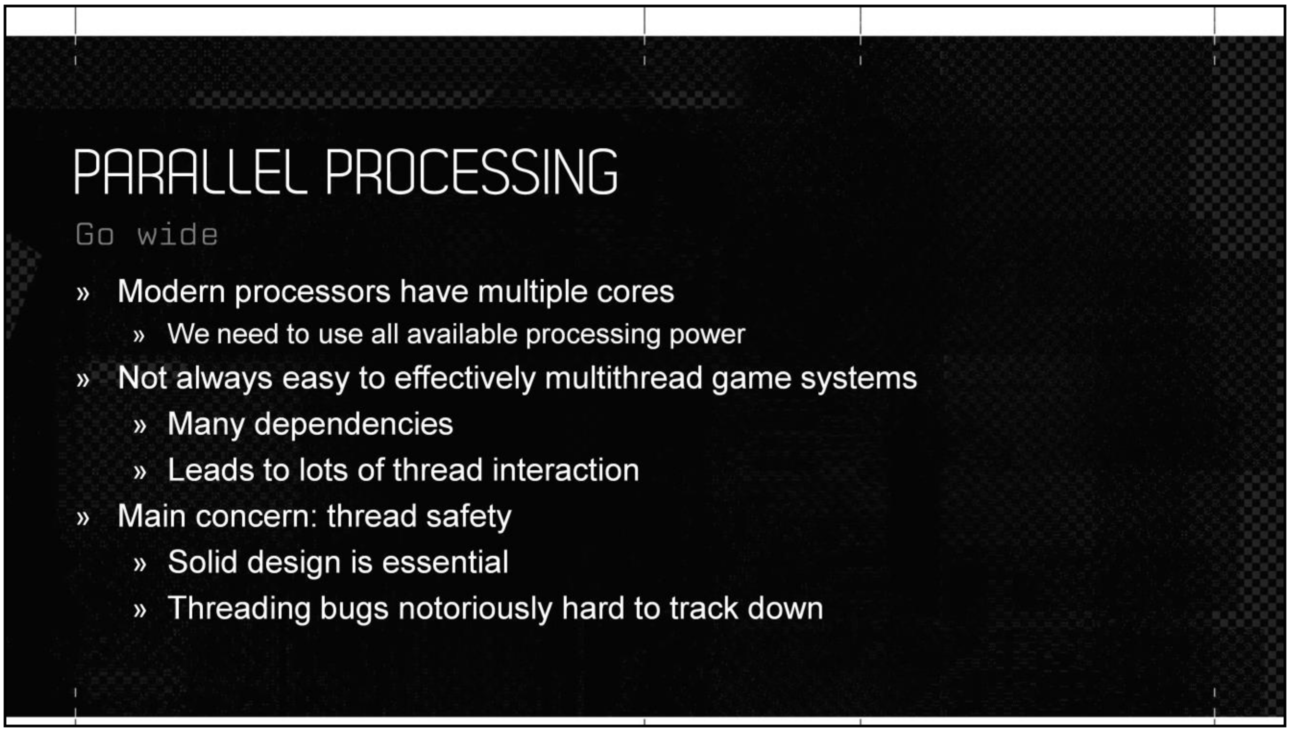
Render target bandwidth

MRTs, bit depth



Multithreading

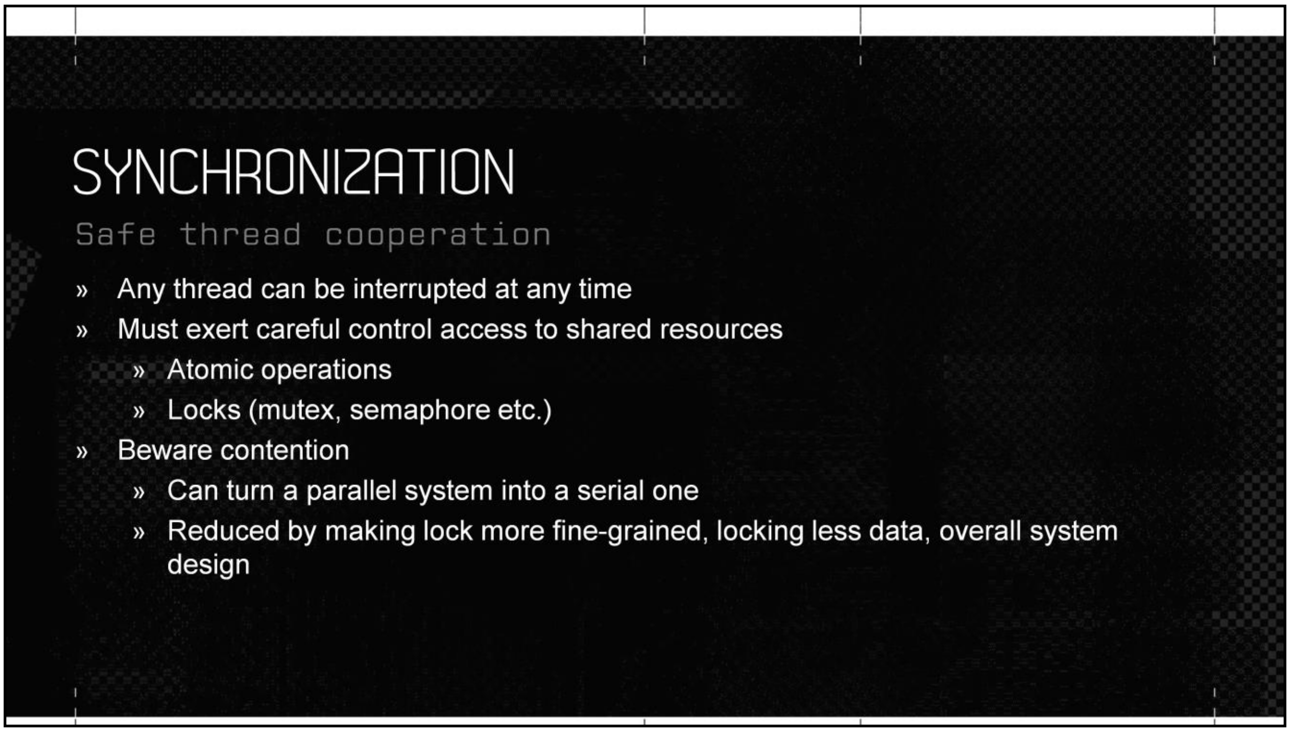
Another large topic, we'll just touch briefly on a few things to be aware of.



PARALLEL PROCESSING

Go wide

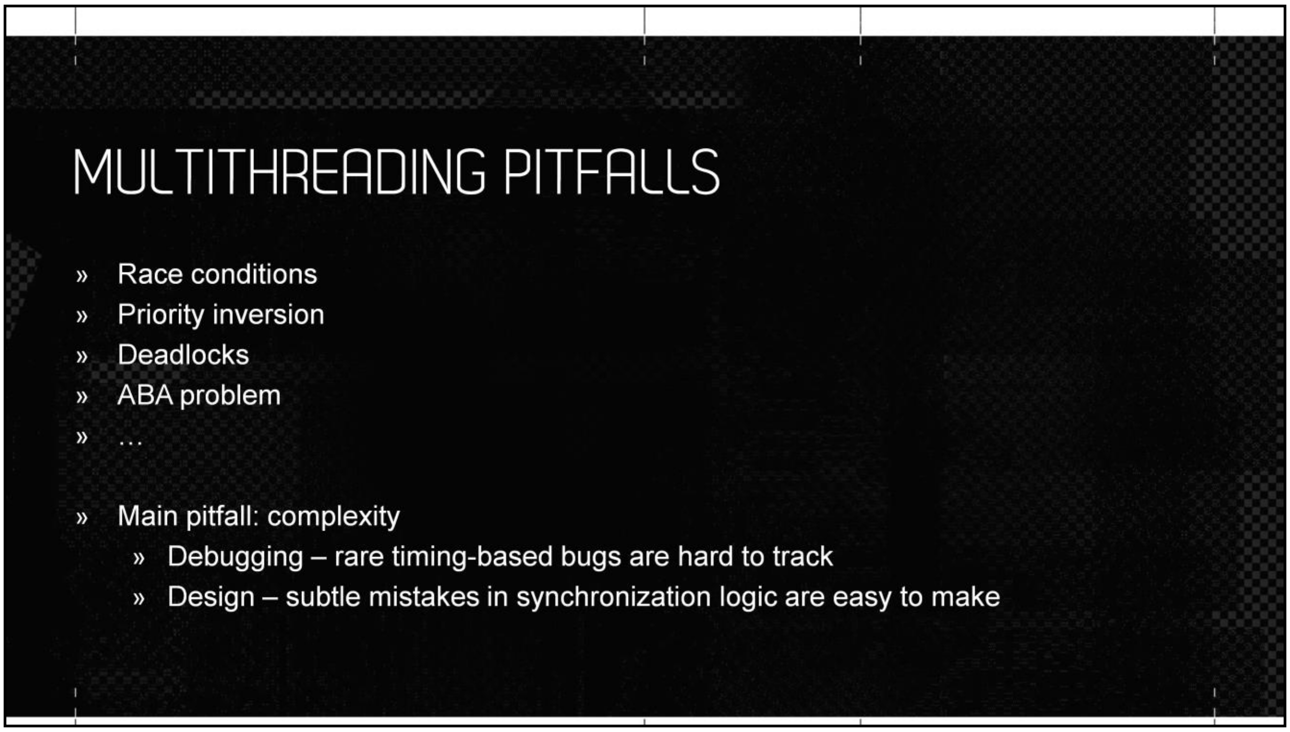
- Modern processors have multiple cores
  - We need to use all available processing power
- Not always easy to effectively multithread game systems
  - Many dependencies
  - Leads to lots of thread interaction
- Main concern: thread safety
  - Solid design is essential
  - Threading bugs notoriously hard to track down



SYNCHRONIZATION

Safe thread cooperation

- Any thread can be interrupted at any time
- Must exert careful control access to shared resources
  - Atomic operations
  - Locks (mutex, semaphore etc.)
- Beware contention
  - Can turn a parallel system into a serial one
  - Reduced by making lock more fine-grained, locking less data, overall system design



MULTITHREADING PITFALLS

- Race conditions
- Priority inversion
- Deadlocks
- ABA problem
- …

- Main pitfall: complexity
  - Debugging – rare timing-based bugs are hard to track
  - Design – subtle mistakes in synchronization logic are easy to make

Just some of the many potential pitfalls

The more complicated/intricate your multithreading setup, the more likely it is to contain subtle insidious bugs

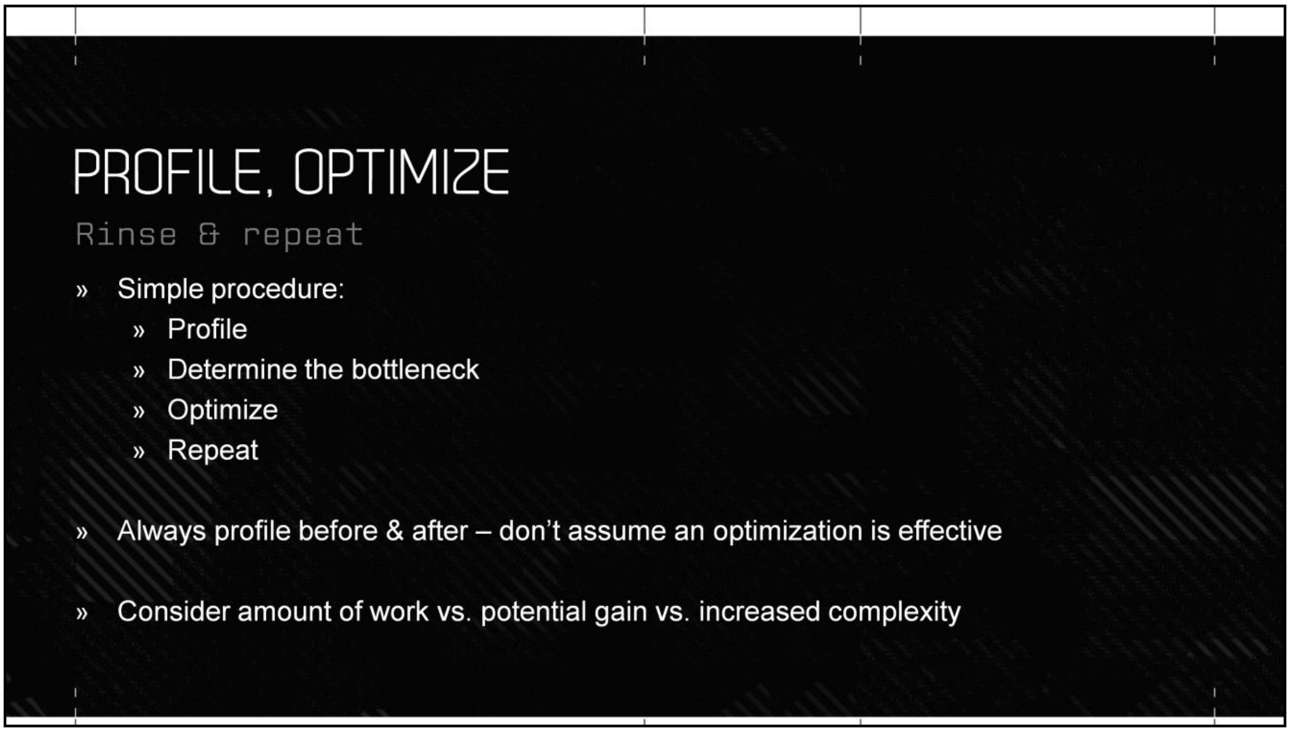
Race condition: Variable results based on order of thread execution

Priority Inversion: Low-priority thread holds lock on resource that high-priority thread needs

Deadlock: Two threads are blocked waiting for locks that the other thread holds

ABA problem: Value changed then changed back to original, but another thread thinks nothing has changed

Optimize effectively



PROFILE, OPTIMIZE

Rinse & repeat

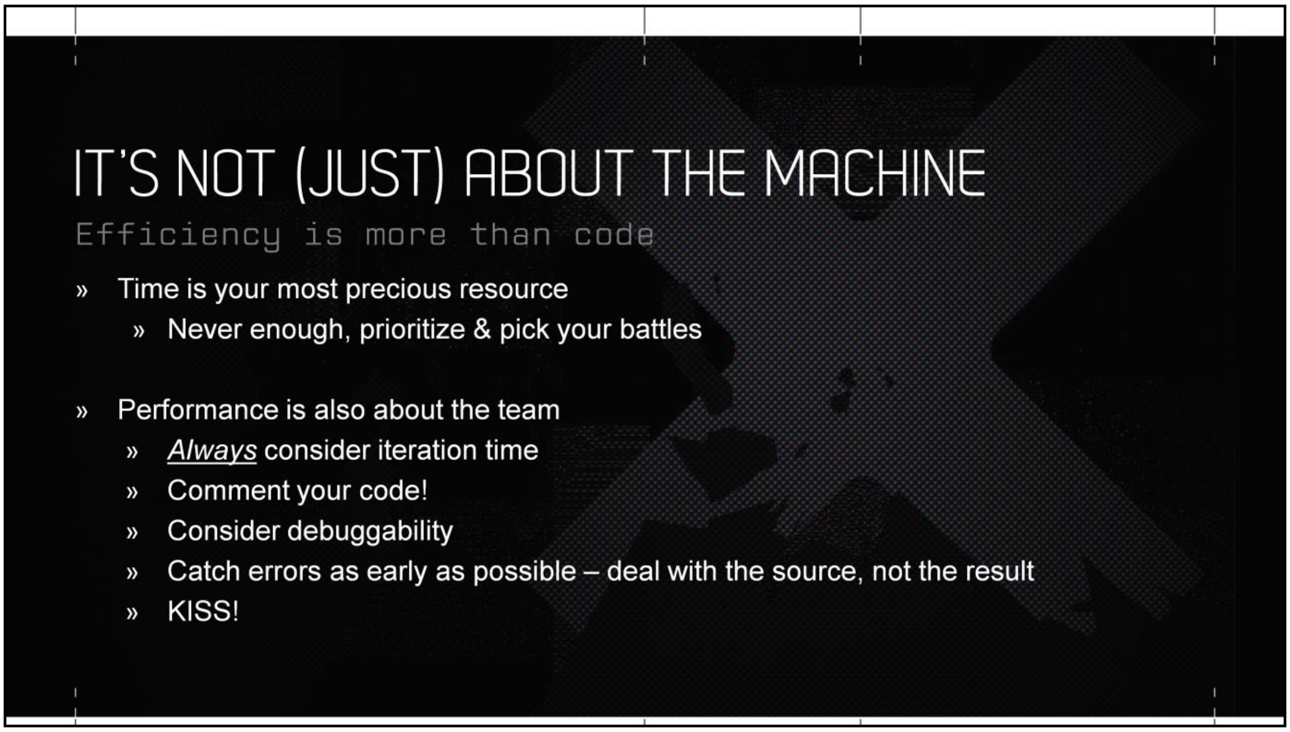
- Simple procedure:
  - Profile
  - Determine the bottleneck
  - Optimize
  - Repeat
- Always profile before & after – don’t assume an optimization is effective
- Consider amount of work vs. potential gain vs. increased complexity

Seems like an obvious point, but it’s easy to mistakenly assume you know where the bottleneck is

Profile before & after – don’t assume that your optimization is a good one

If possible, keep both versions of code functional - makes for much easier profiling.

Other considerations



IT'S NOT (JUST) ABOUT THE MACHINE

Efficiency is more than code

- Time is your most precious resource
  - Never enough, prioritize & pick your battles
- Performance is also about the team
  - *Always* consider iteration time
  - Comment your code!
  - Consider debuggability
  - Catch errors as early as possible – deal with the source, not the result
  - KISS!

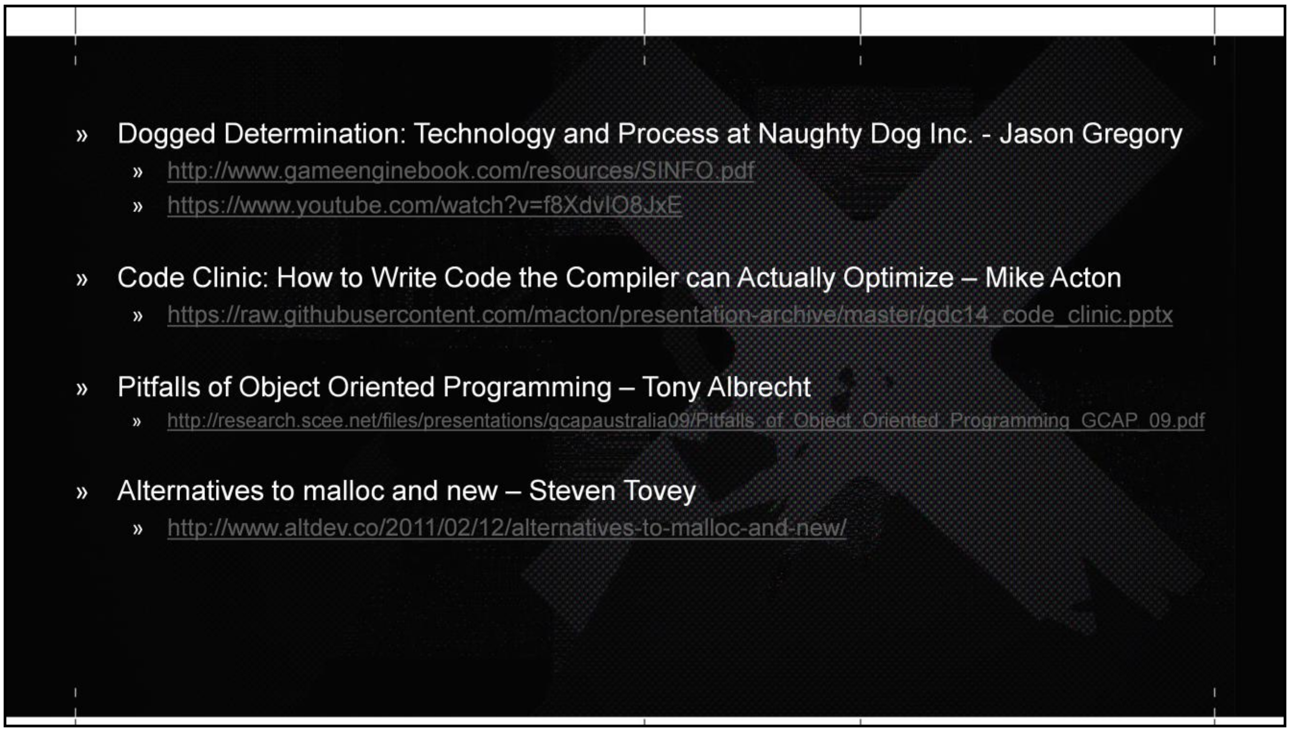
Iteration time is massively important – if a change is going to increase build/export time, slow down artists, or have other negative team-wide impact, weight it up *very* carefully.

Comment your code – not only for your fellow coders, but also for yourself in six months' time

Debuggability – a piece of code may be awesomely vectorized, streamlined, tightly packed and memory efficient, but if something goes wrong how hard is it going to be to pick apart?

Catch errors early – if data can be exported incorrectly, deal with that at export time. Dealing with bad data at load time is much messier, and will lead to errors not being dealt with and piling up.

References



- Dogged Determination: Technology and Process at Naughty Dog Inc. - Jason Gregory
  - http://www.gameenginebook.com/resources/SINFO.pdf
  - https://www.youtube.com/watch?v=f8XdvIO8JxE
- Code Clinic: How to Write Code the Compiler can Actually Optimize – Mike Acton
  - https://raw.githubusercontent.com/macton/presentation-archive/master/gdc14_code_clinic.pptx
- Pitfalls of Object Oriented Programming – Tony Albrecht
  - http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf
- Alternatives to malloc and new – Steven Tovey
  - http://www.altdev.co/2011/02/12/alternatives-to-malloc-and-new/

Dogged Determination: Technology and Process at Naughty Dog Inc. - Jason Gregory

http://www.gameenginebook.com/resources/SINFO.pdf

https://www.youtube.com/watch?v=f8XdvIO8JxE

Code Clinic: How to Write Code the Compiler can Actually Optimize – Mike Acton

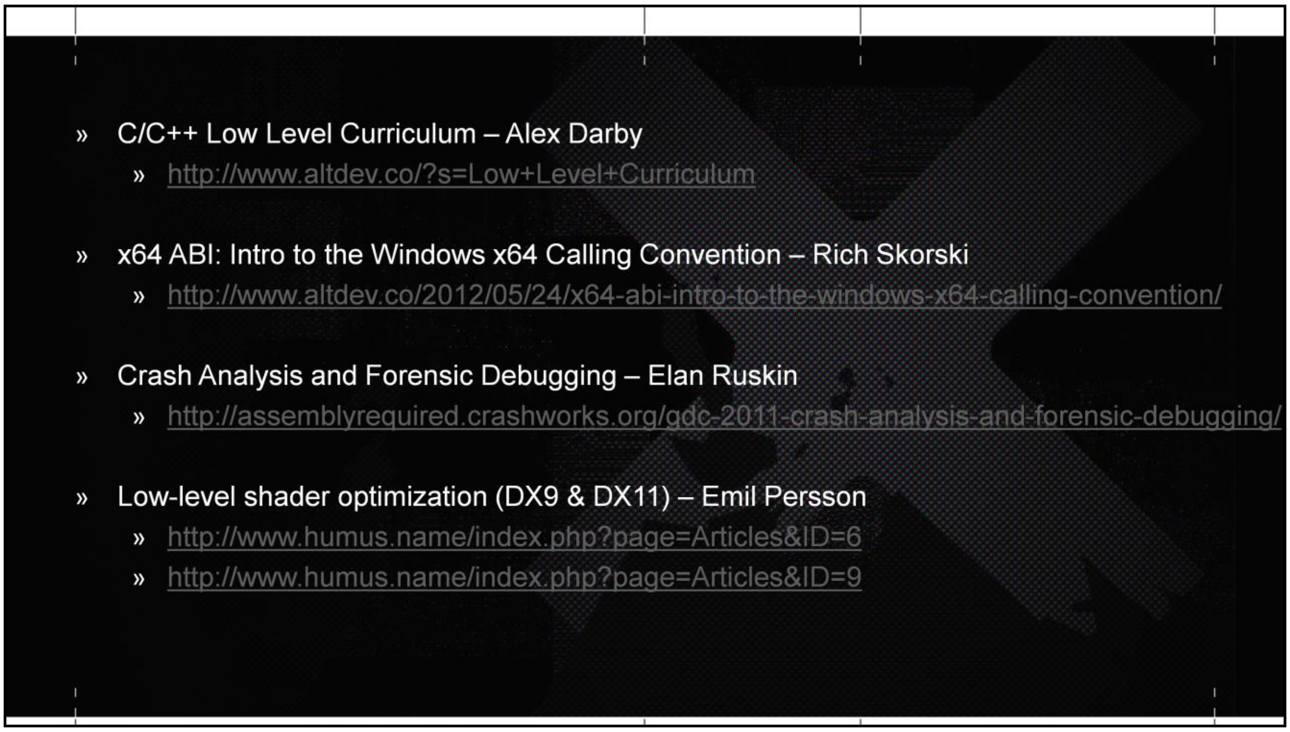
https://raw.githubusercontent.com/macton/presentation-archive/master/gdc14_code_clinic.pptx

Pitfalls of Object Oriented Programming – Tony Albrecht

http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf

Alternatives to malloc and new – Steven Tovey

http://www.altdev.co/2011/02/12/alternatives-to-malloc-and-new/

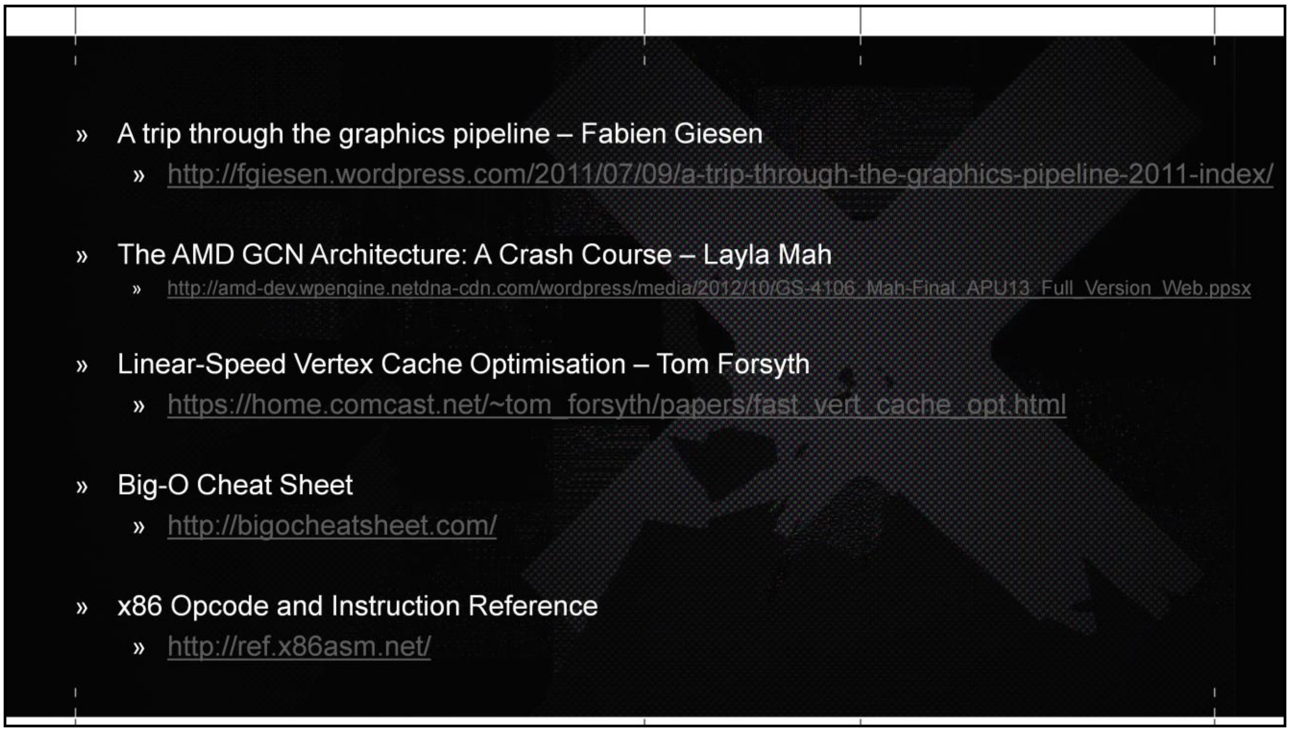

- C/C++ Low Level Curriculum – Alex Darby
  - http://www.altdev.co/?s=Low+Level+Curriculum
- x64 ABI: Intro to the Windows x64 Calling Convention – Rich Skorski
  - http://www.altdev.co/2012/05/24/x64-abi-intro-to-the-windows-x64-calling-convention/
- Crash Analysis and Forensic Debugging – Elan Ruskin
  - http://assemblyrequired.crashworks.org/gdc-2011-crash-analysis-and-forensic-debugging/
- Low-level shader optimization (DX9 & DX11) – Emil Persson
  - http://www.humus.name/index.php?page=Articles&ID=6
  - http://www.humus.name/index.php?page=Articles&ID=9

C/C++ Low Level Curriculum – Alex Darby
http://www.altdev.co/?s=Low+Level+Curriculum

x64 ABI: Intro to the Windows x64 Calling Convention – Rich Skorski
http://www.altdev.co/2012/05/24/x64-abi-intro-to-the-windows-x64-calling-convention/

Crash Analysis and Forensic Debugging – Elan Ruskin
http://assemblyrequired.crashworks.org/gdc-2011-crash-analysis-and-forensic-debugging/

Low-level shader optimization (DX9 & DX11) – Emil Persson
http://www.humus.name/index.php?page=Articles&ID=6
http://www.humus.name/index.php?page=Articles&ID=9



» A trip through the graphics pipeline – Fabien Giesen
  » http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/
» The AMD GCN Architecture: A Crash Course – Layla Mah
  » http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/GS-4106_Mah-Final_APU13_Full_Version_Web.ppsx
» Linear-Speed Vertex Cache Optimisation – Tom Forsyth
  » https://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html
» Big-O Cheat Sheet
  » http://bigocheatsheet.com/
» x86 Opcode and Instruction Reference
  » http://ref.x86asm.net/

A trip through the graphics pipeline – Fabien Giesen

http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/

The AMD GCN Architecture: A Crash Course – Layla Mah

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/GS-4106_Mah-Final_APU13_Full_Version_Web.ppsx

Linear-Speed Vertex Cache Optimisation – Tom Forsyth

https://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html

Big-O Cheat Sheet

http://bigocheatsheet.com/

x86 Opcode and Instruction Reference

http://ref.x86asm.net/