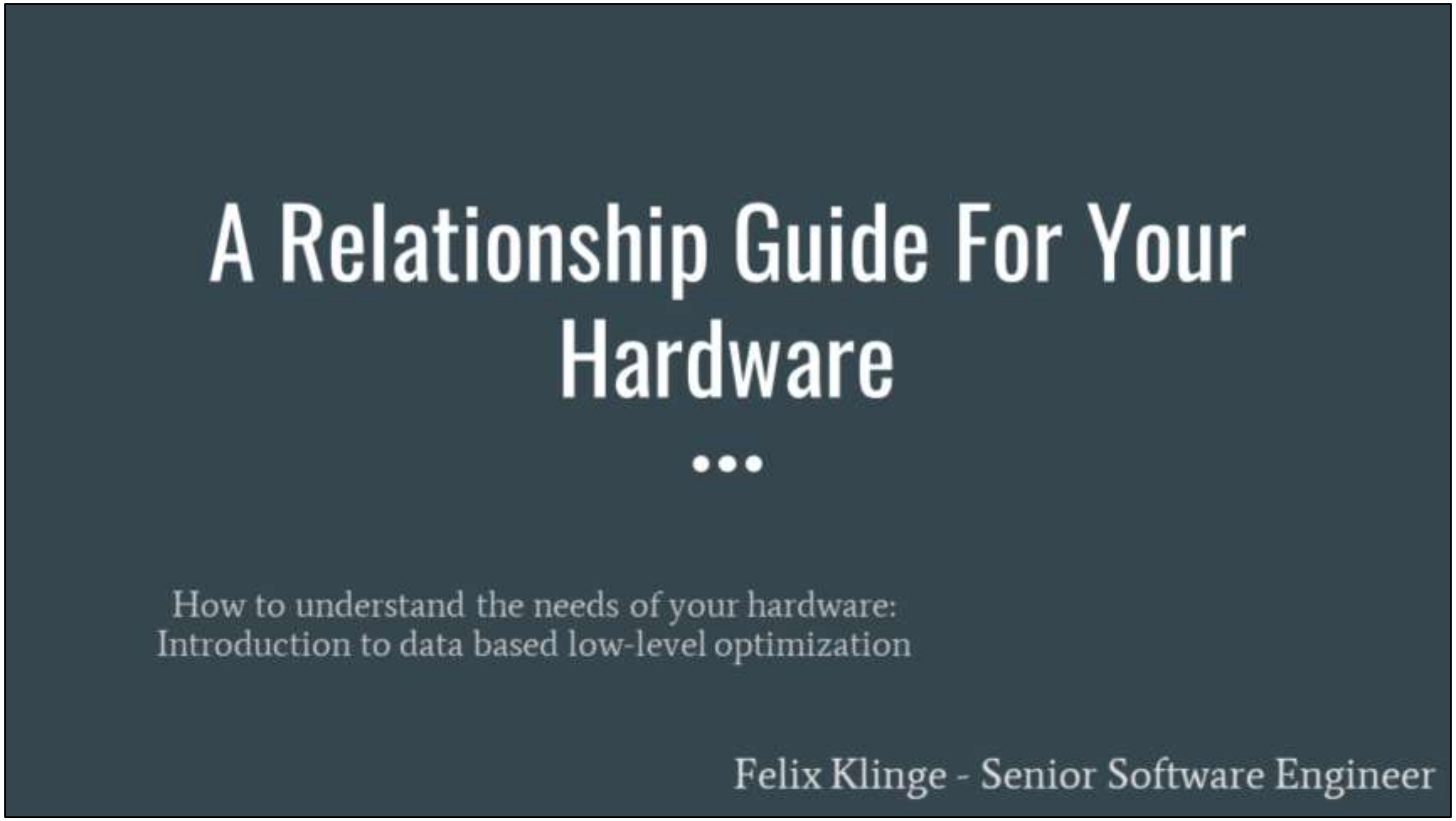


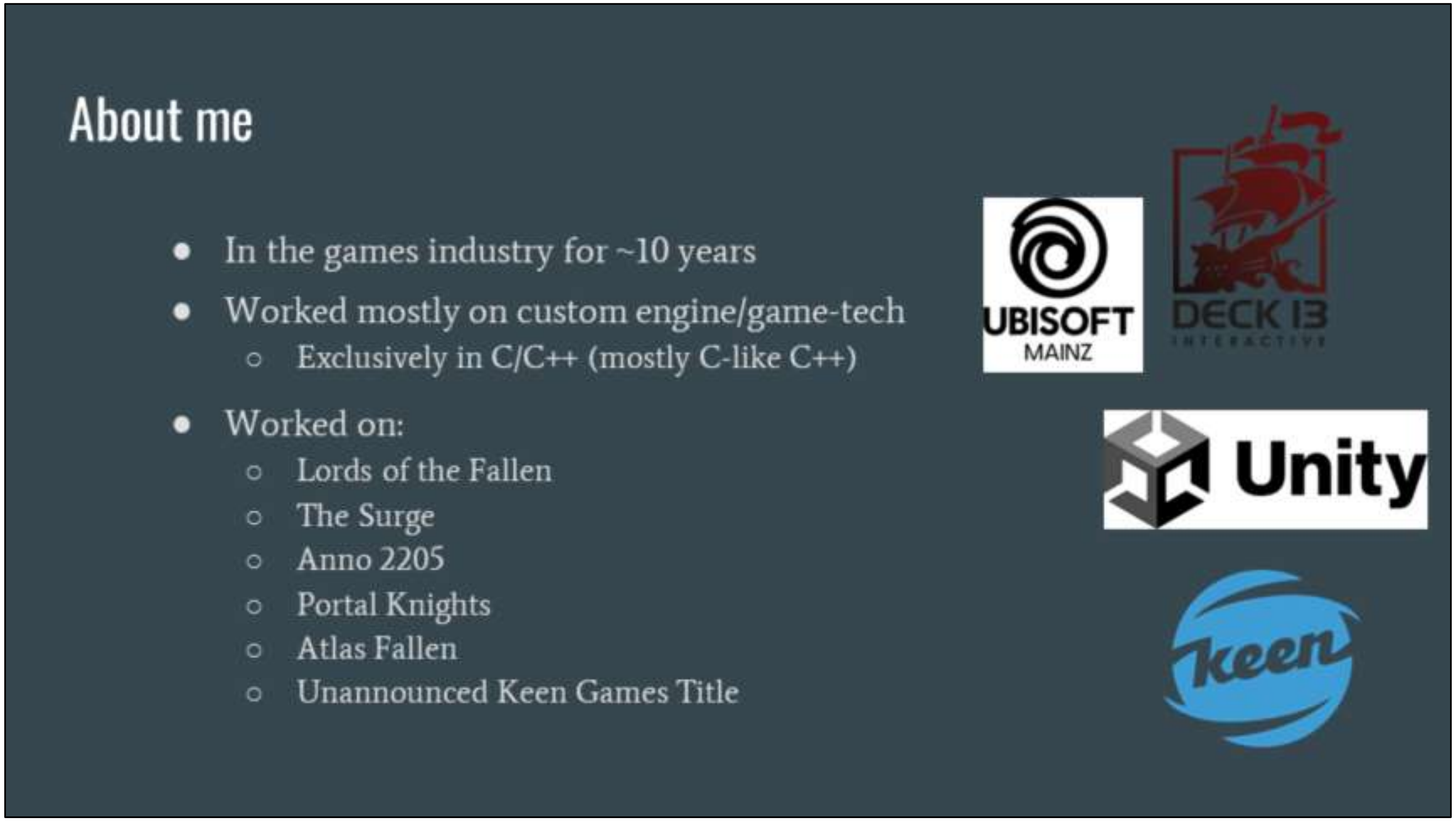
A Relationship Guide For Your Hardware

How to understand the needs of your hardware:
Introduction to data based low-level optimization

Felix Klinge - Senior Software Engineer

Agenda

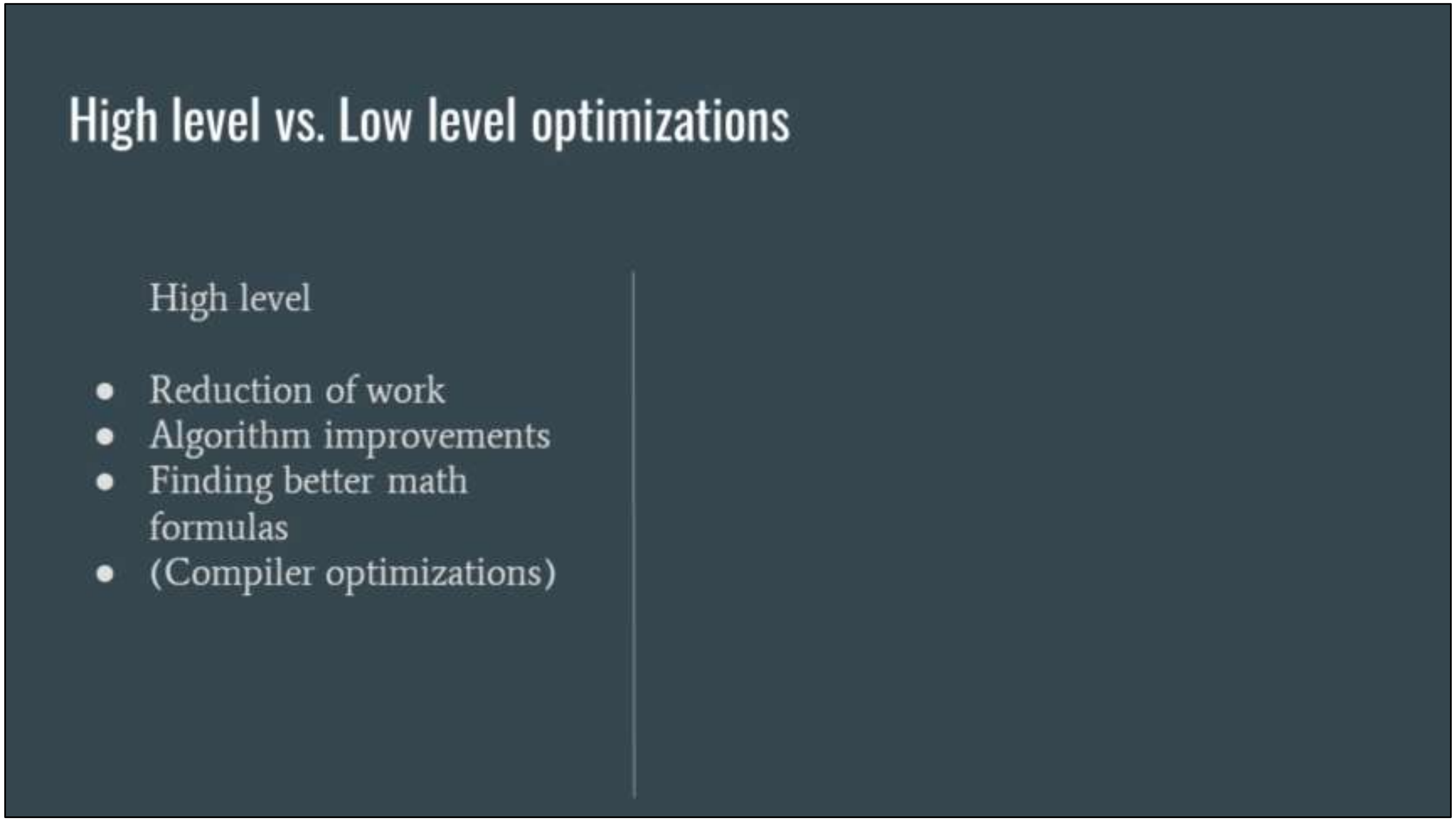
- About me
- High Level vs Low Level optimizations
- Why should I care about low level stuff?
- Example: 2D bitmap rotation
- Naive Implementation
- Optimization 1: Better Execution Unit Utilization
- Optimization 2: Loop Blocking
- Optimization 3: Multithreading
- Optimization 4: SIMD
- Conclusion



About me

- In the games industry for ~10 years
- Worked mostly on custom engine/game-tech
  - Exclusively in C/C++ (mostly C-like C++)
- Worked on:
  - Lords of the Fallen
  - The Surge
  - Anno 2205
  - Portal Knights
  - Atlas Fallen
  - Unannounced Keen Games Title

Let me introduce me first.
Hi, I'm Felix and I've been a programmer in the games industry for roughly 10 years.
During my time in the industry, I've been mostly working on custom game- and engine tech for various companies like Ubisoft, Deck13, Keen Games and now, Unity.



High level vs. Low level optimizations

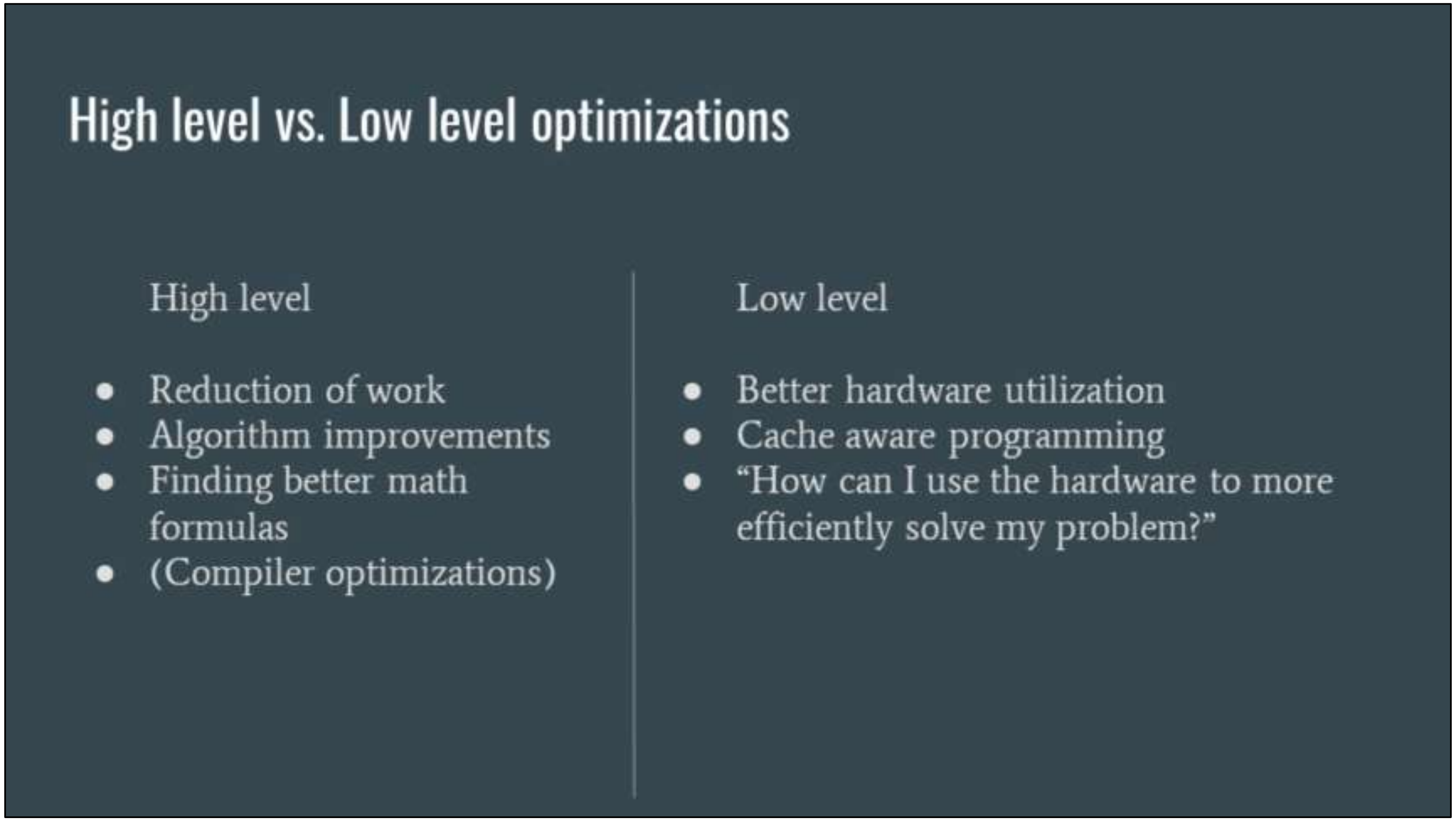
High level

- Reduction of work
- Algorithm improvements
- Finding better math formulas
- (Compiler optimizations)

Let's first draw a clear line between what actually is “high level” and “low level” optimizations to focus on what I'll be talking about in this talk.

High level optimizations are optimizations that don't take the hardware into account. That could be things like reducing the amount of work needed (shortening an algorithm), improving upon an existing algorithm or using a different algorithm that better fits the problem space or finding better math formulas/shortening existing math formulas. Compiler optimizations could be considered high level optimizations but they ultimivately will end up in the low-level optimization territory.

Low Level optimizations on the other hand are optimizations that take the hardware into account to achieve better hardware utlization by incorporating cache aware programming. Basically you ask yourself the question “how can I use the hardware to more efficiently solve my problem?”


High level vs. Low level optimizations

High level

- Reduction of work
- Algorithm improvements
- Finding better math formulas
- (Compiler optimizations)

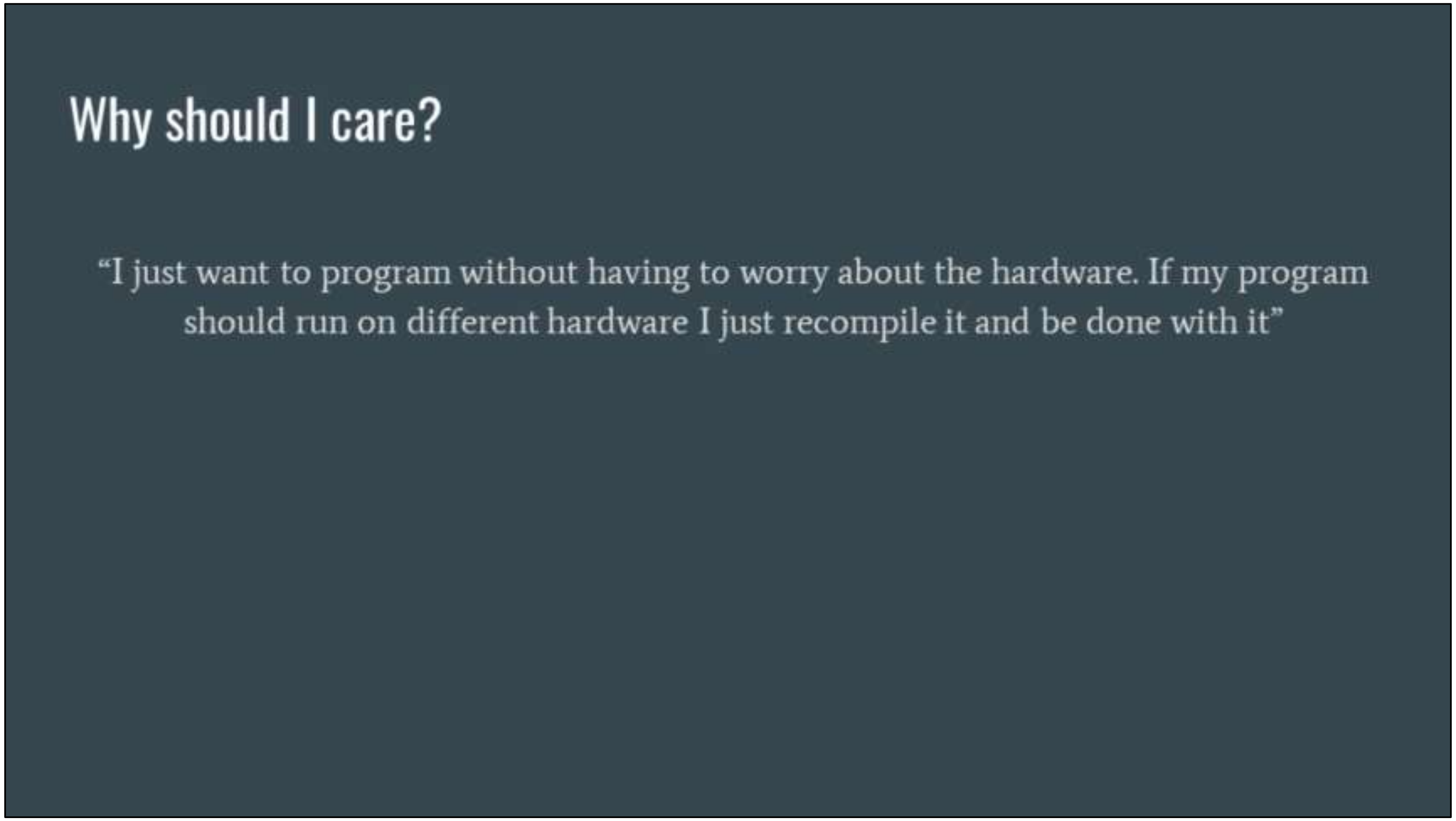
Low level

- Better hardware utilization
- Cache aware programming
- “How can I use the hardware to more efficiently solve my problem?”

Let’s first draw a clear line between what actually is “high level” and “low level” optimizations to focus on what I’ll be talking about in this talk.

High level optimizations are optimizations that don’t take the hardware into account. That could be things like reducing the amount of work needed (shortening an algorithm), improving upon an existing algorithm or using a different algorithm that better fits the problem space or finding better math formulas/shortening existing math formulas. Compiler optimizations could be considered high level optimizations but they ultimivately will end up in the low-level optimization territory.

Low Level optimizations on the other hand are optimizations that take the hardware into account to achieve better hardware utlization by incorporating cache aware programming. Basically you ask yourself the question “how can I use the hardware to more efficiently solve my problem?”



Why should I care?

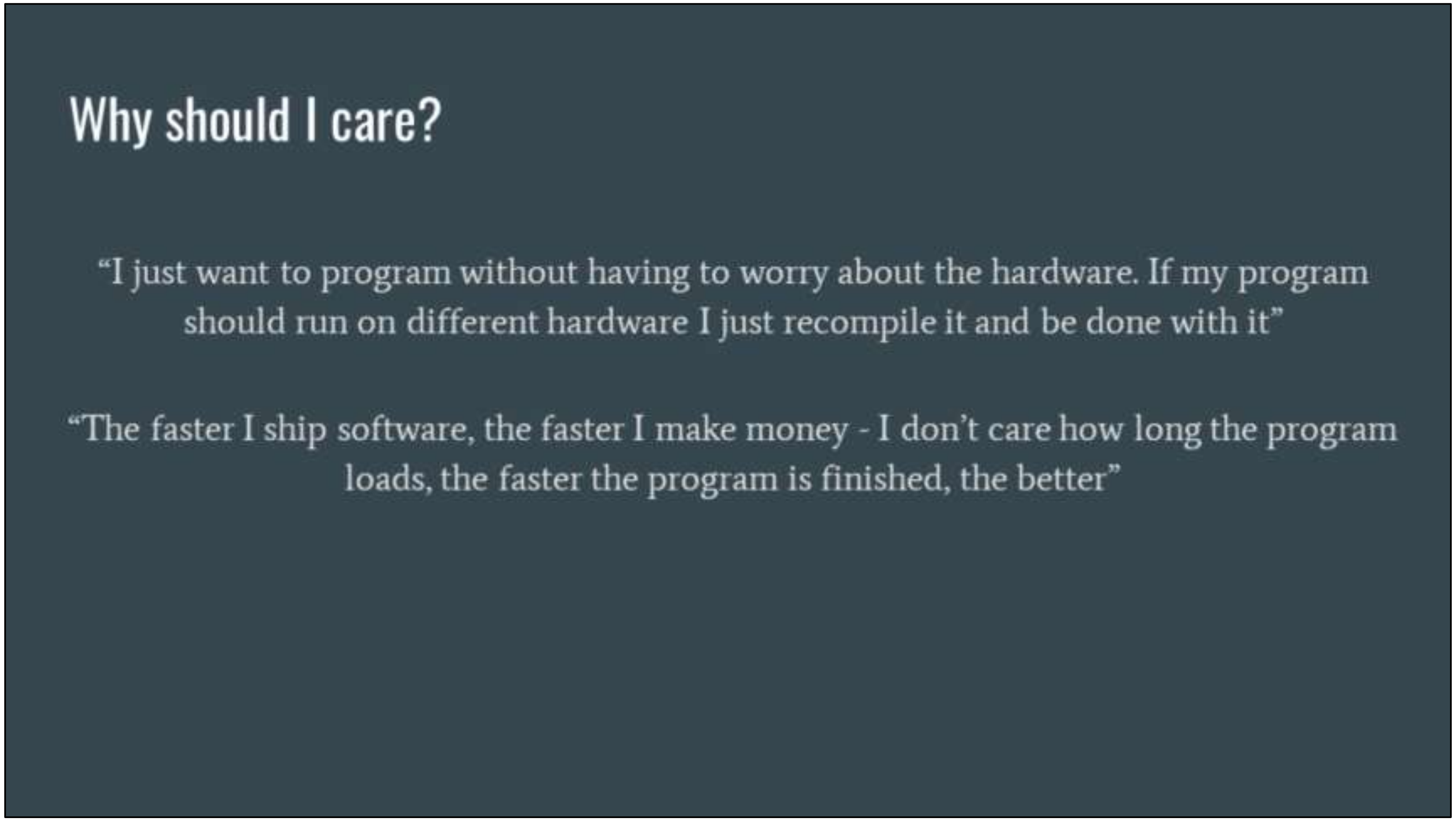
"I just want to program without having to worry about the hardware. If my program should run on different hardware I just recompile it and be done with it"

Before going into low level optimization I want to emphasize why you should care about it.

I've read statements over the years that go like "yeah, I don't really need to know, I just want my program to work" or "time is money, investing time into optimizations (even high level) is a waste of time if it means that the programmer has to spent more time on the project". Especially the last one is a real world problem.

The problem with these mindsets however is that software is getting slower and slower. I don't necessarily mean video games but software in general. Your Word, Browsers, Text Editors etc. Everything is so far abstracted from the hardware that nobody has any idea anymore what actual instructions are actually being processed by the CPU.

This twitter video encapsulates perfectly what I'm talking about.



Why should I care?

"I just want to program without having to worry about the hardware. If my program should run on different hardware I just recompile it and be done with it"

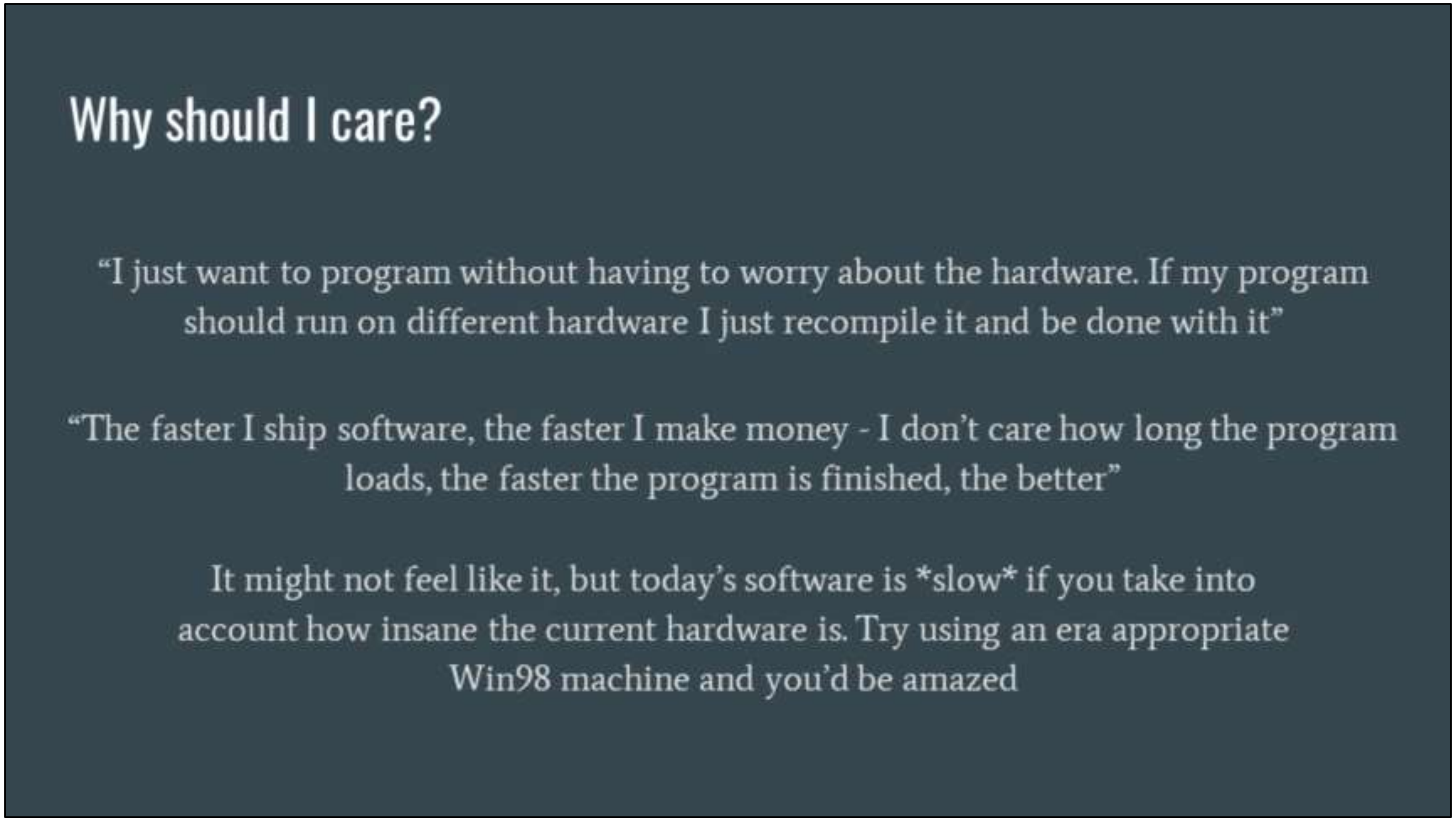
"The faster I ship software, the faster I make money - I don't care how long the program loads, the faster the program is finished, the better"

Before going into low level optimization I want to emphasize why you should care about it.

I've read statements over the years that go like "yeah, I don't really need to know, I just want my program to work" or "time is money, investing time into optimizations (even high level) is a waste of time if it means that the programmer has to spent more time on the project". Especially the last one is a real world problem.

The problem with these mindsets however is that software is getting slower and slower. I don't necessarily mean video games but software in general. Your Word, Browsers, Text Editors etc. Everything is so far abstracted from the hardware that nobody has any idea anymore what actual instructions are actually being processed by the CPU.

This twitter video encapsulates perfectly what I'm talking about.



Why should I care?

“I just want to program without having to worry about the hardware. If my program should run on different hardware I just recompile it and be done with it”

“The faster I ship software, the faster I make money - I don’t care how long the program loads, the faster the program is finished, the better”

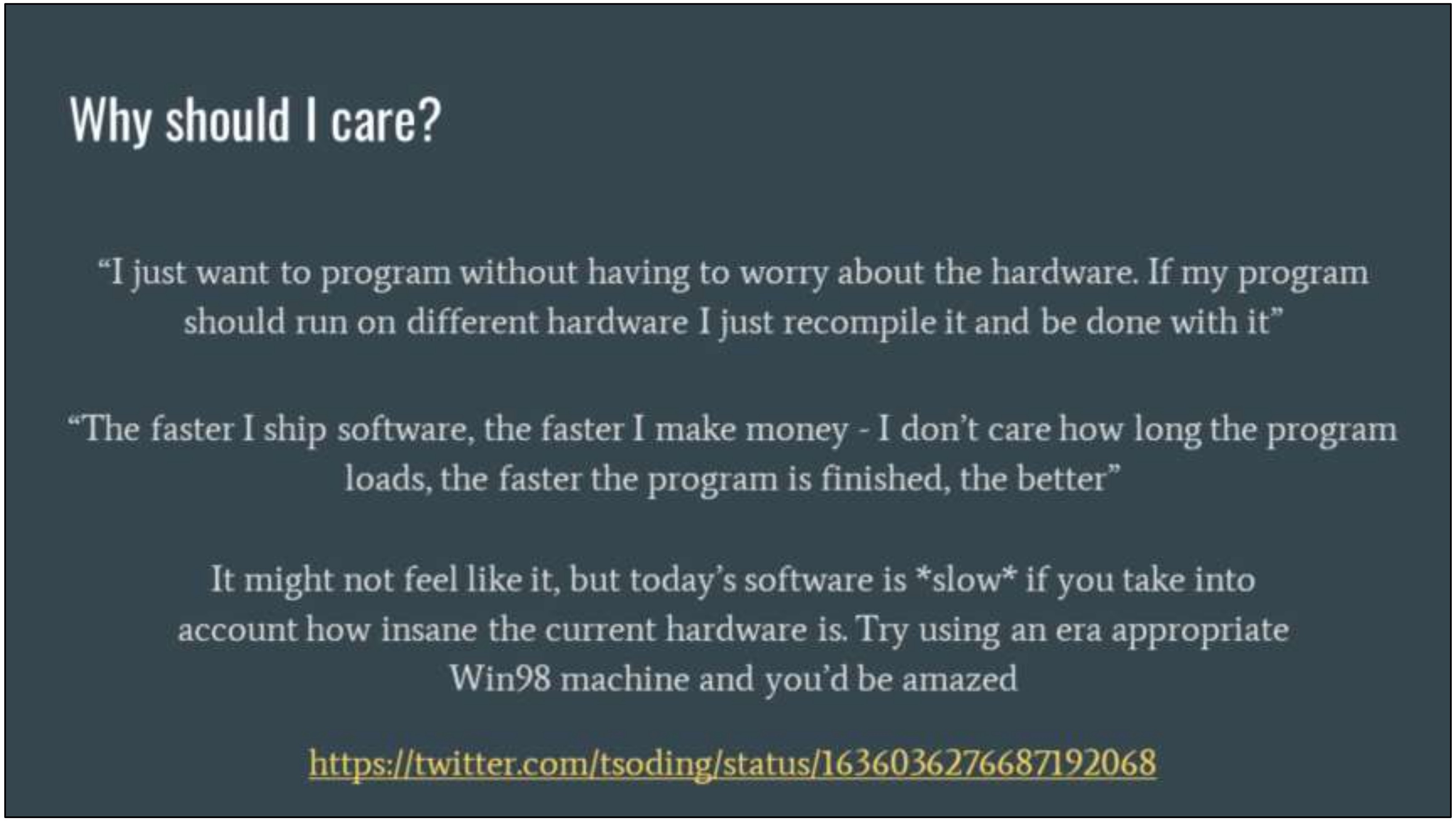
It might not feel like it, but today’s software is *slow* if you take into account how insane the current hardware is. Try using an era appropriate Win98 machine and you’d be amazed

Before going into low level optimization I want to emphasize why you should care about it.

I’ve read statements over the years that go like “yeah, I don’t really need to know, I just want my program to work” or “time is money, investing time into optimizations (even high level) is a waste of time if it means that the programmer has to spent more time on the project”. Especially the last one is a real world problem.

The problem with these mindsets however is that software is getting slower and slower. I don’t necessarily mean video games but software in general. Your Word, Browsers, Text Editors etc. Everything is so far abstracted from the hardware that nobody has any idea anymore what actual instructions are actually being processed by the CPU.

This twitter video encapsulates perfectly what I’m talking about.



Why should I care?

"I just want to program without having to worry about the hardware. If my program should run on different hardware I just recompile it and be done with it"

"The faster I ship software, the faster I make money - I don't care how long the program loads, the faster the program is finished, the better"

It might not feel like it, but today's software is *slow* if you take into account how insane the current hardware is. Try using an era appropriate Win98 machine and you'd be amazed

https://twitter.com/tsoding/status/1636036276687192068

Before going into low level optimization I want to emphasize why you should care about it.

I've read statements over the years that go like "yeah, I don't really need to know, I just want my program to work" or "time is money, investing time into optimizations (even high level) is a waste of time if it means that the programmer has to spent more time on the project". Especially the last one is a real world problem.

The problem with these mindsets however is that software is getting slower and slower. I don't necessarily mean video games but software in general. Your Word, Browsers, Text Editors etc. Everything is so far abstracted from the hardware that nobody has any idea anymore what actual instructions are actually being processed by the CPU.

This twitter video encapsulates perfectly what I'm talking about.

Why should I care?

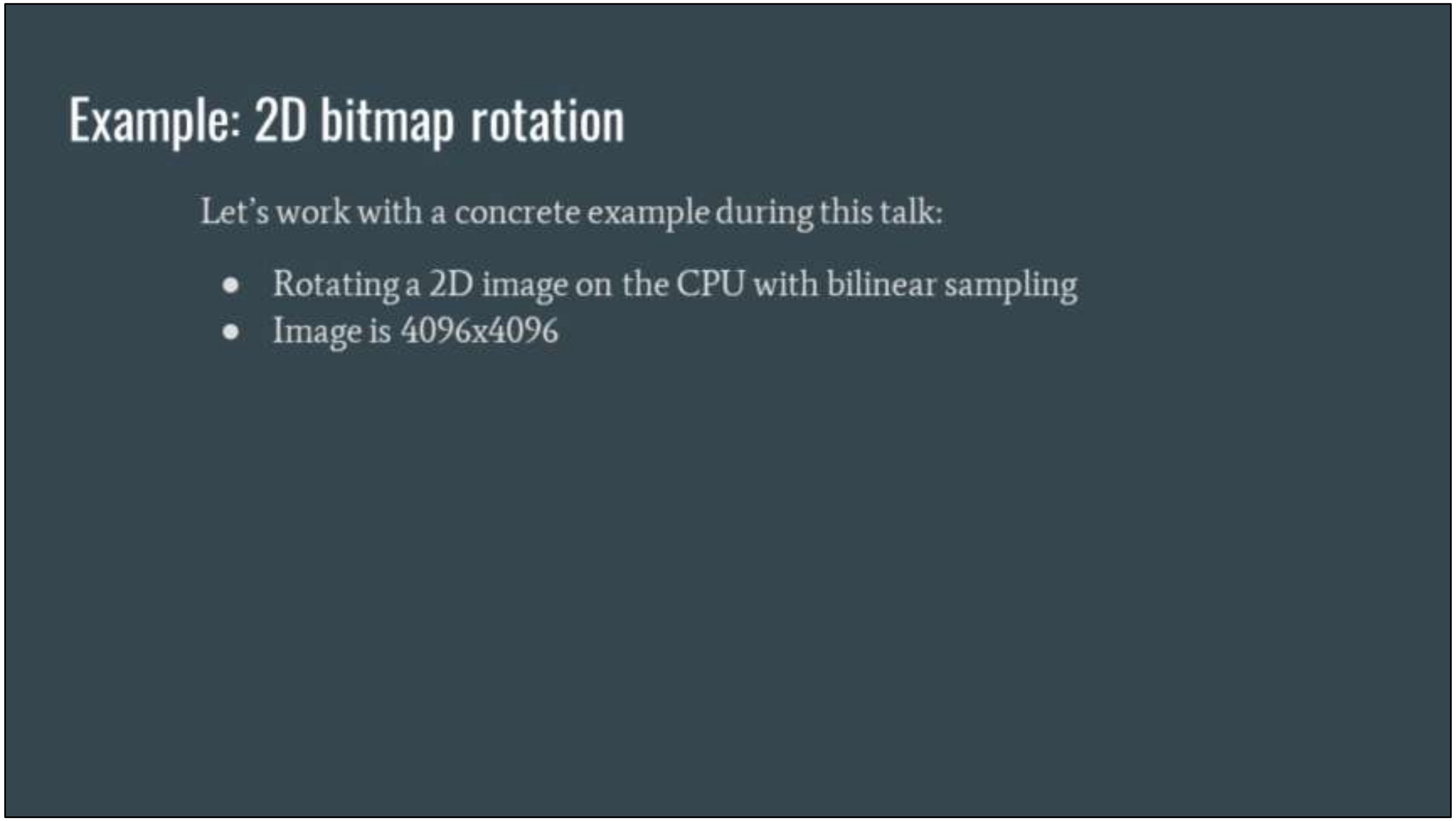
This is where we're at right now (this is *not* a parody):

http://www.youtube.com/watch?v=CT7nnXej2K4

I also wanted to show this video which recently has been shared on Twitter.

This is a video of Microsoft Teams, a Slack-like communication program. In this video they present new performance improvements.
I'll only show the first improvement they show in this video, namely a faster startup time.

In this video they show that the startup time went from 22.2s to 9.1s.
I'll let you draw your own conclusions but I want to throw in here that the laptop that I'm showing this presentation on reboots in about 8s.



Example: 2D bitmap rotation

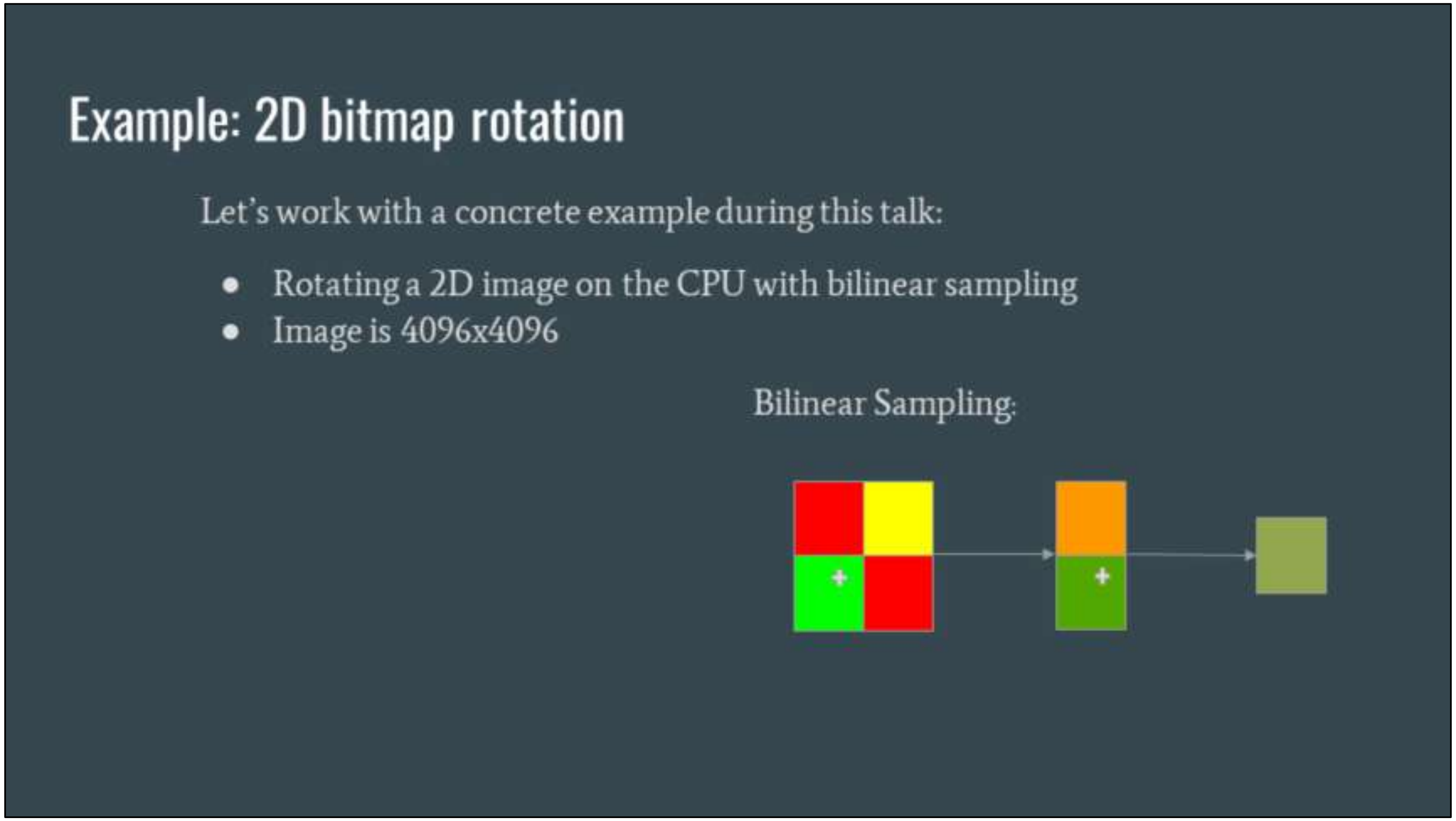
Let's work with a concrete example during this talk:

- Rotating a 2D image on the CPU with bilinear sampling
- Image is 4096x4096

Lets define a concrete example that we'll use as our problem to optimize in this talk.

For this talk we'll be looking at an algorithm that rotates a 2D bitmap by an arbitrary angle.
Since some sampling has to be done a bilinear sampling should be implemented.

The input for our algorithm will be an image and a transformation matrix that represents the rotation
The output will be the rotated image

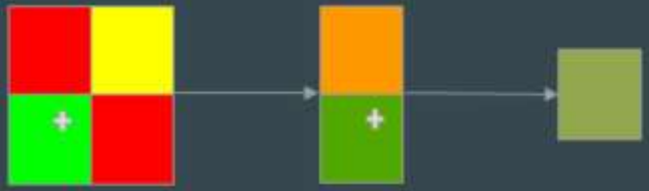


Example: 2D bitmap rotation

Let's work with a concrete example during this talk:

- Rotating a 2D image on the CPU with bilinear sampling
- Image is 4096x4096

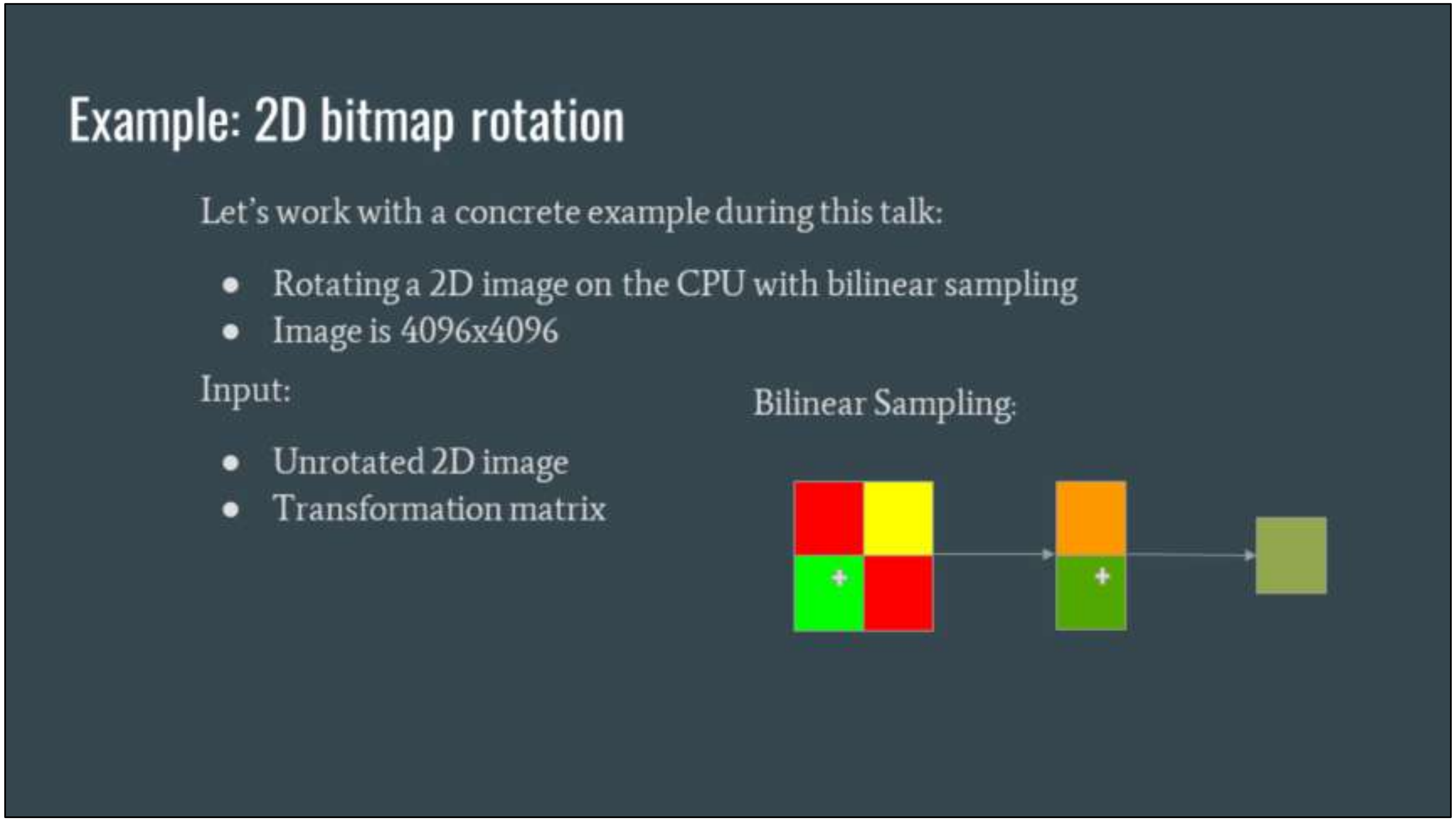
Bilinear Sampling:



Lets define a concrete example that we'll use as our problem to optimize in this talk.

For this talk we'll be looking at an algorithm that rotates a 2D bitmap by an arbitrary angle.
Since some sampling has to be done a bilinear sampling should be implemented.

The input for our algorithm will be an image and a transformation matrix that represents the rotation
The output will be the rotated image



Example: 2D bitmap rotation

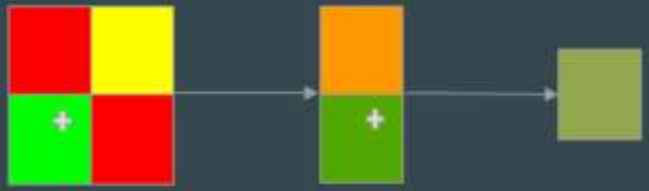
Let's work with a concrete example during this talk:

- Rotating a 2D image on the CPU with bilinear sampling
- Image is 4096x4096

Input:

- Unrotated 2D image
- Transformation matrix

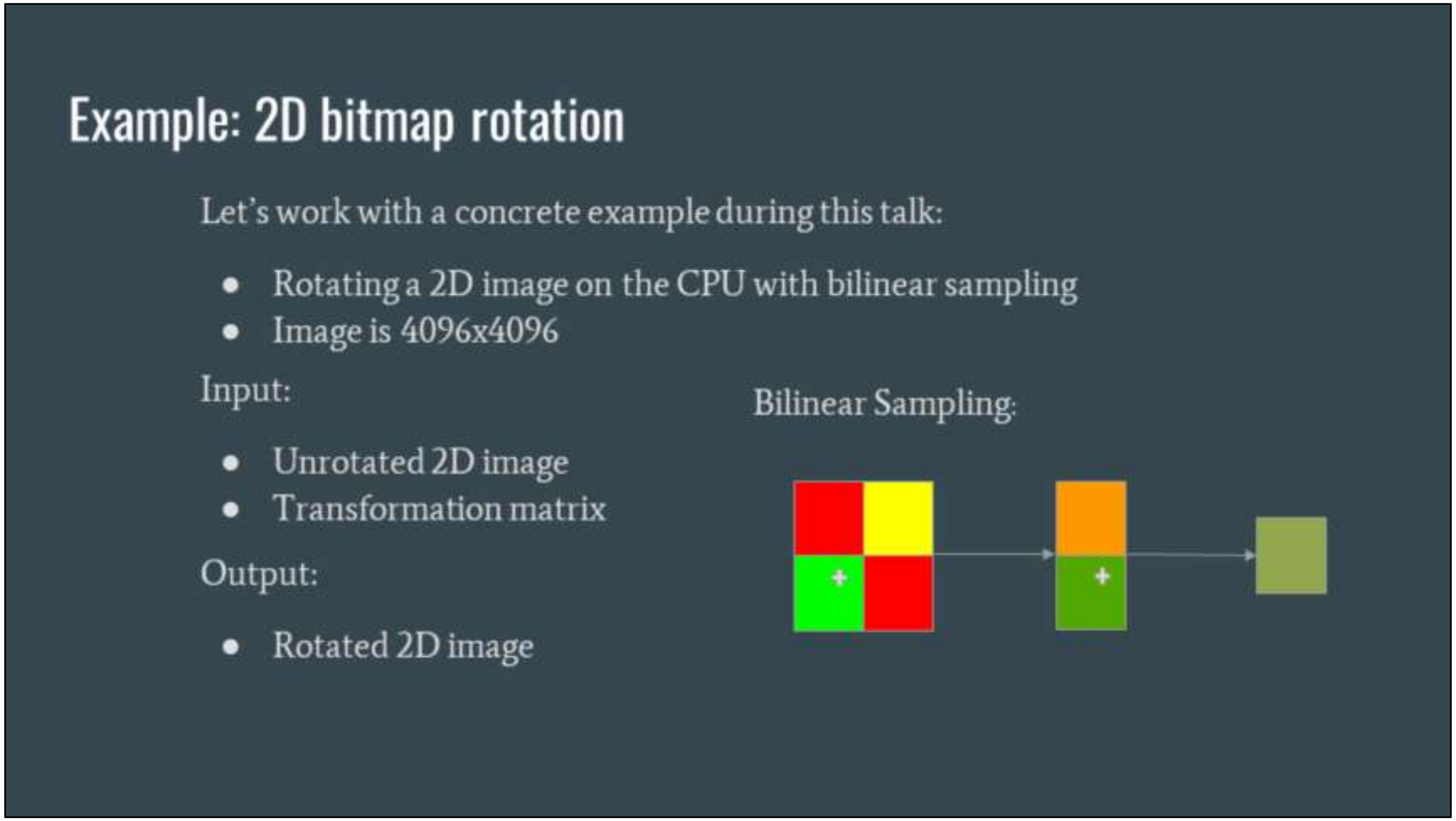
Bilinear Sampling:



Lets define a concrete example that we'll use as our problem to optimize in this talk.

For this talk we'll be looking at an algorithm that rotates a 2D bitmap by an arbitrary angle.
Since some sampling has to be done a bilinear sampling should be implemented.

The input for our algorithm will be an image and a transformation matrix that represents the rotation
The output will be the rotated image



Example: 2D bitmap rotation

Let's work with a concrete example during this talk:

- Rotating a 2D image on the CPU with bilinear sampling
- Image is 4096x4096

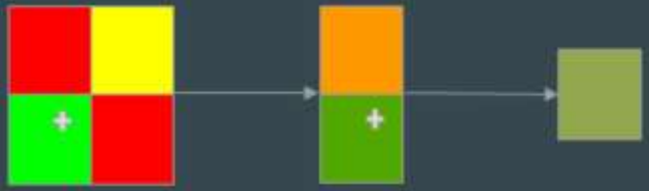
Input:

- Unrotated 2D image
- Transformation matrix

Output:

- Rotated 2D image

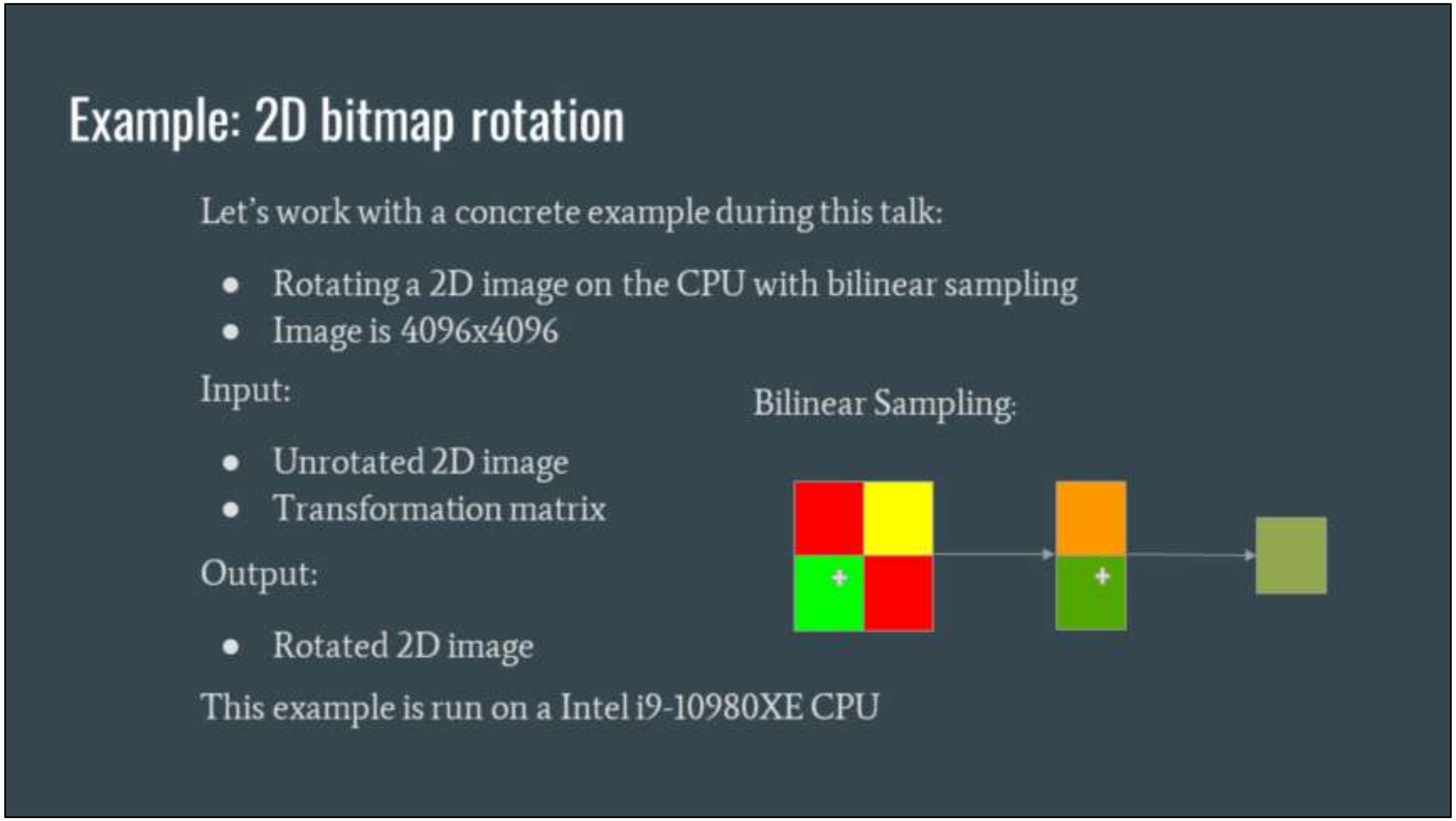
Bilinear Sampling:



Lets define a concrete example that we'll use as our problem to optimize in this talk.

For this talk we'll be looking at an algorithm that rotates a 2D bitmap by an arbitrary angle.
Since some sampling has to be done a bilinear sampling should be implemented.

The input for our algorithm will be an image and a transformation matrix that represents the rotation
The output will be the rotated image



Example: 2D bitmap rotation

Let's work with a concrete example during this talk:

- Rotating a 2D image on the CPU with bilinear sampling
- Image is 4096x4096

Input:

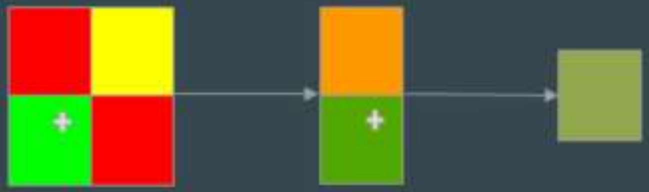
- Unrotated 2D image
- Transformation matrix

Output:

- Rotated 2D image

This example is run on a Intel i9-10980XE CPU

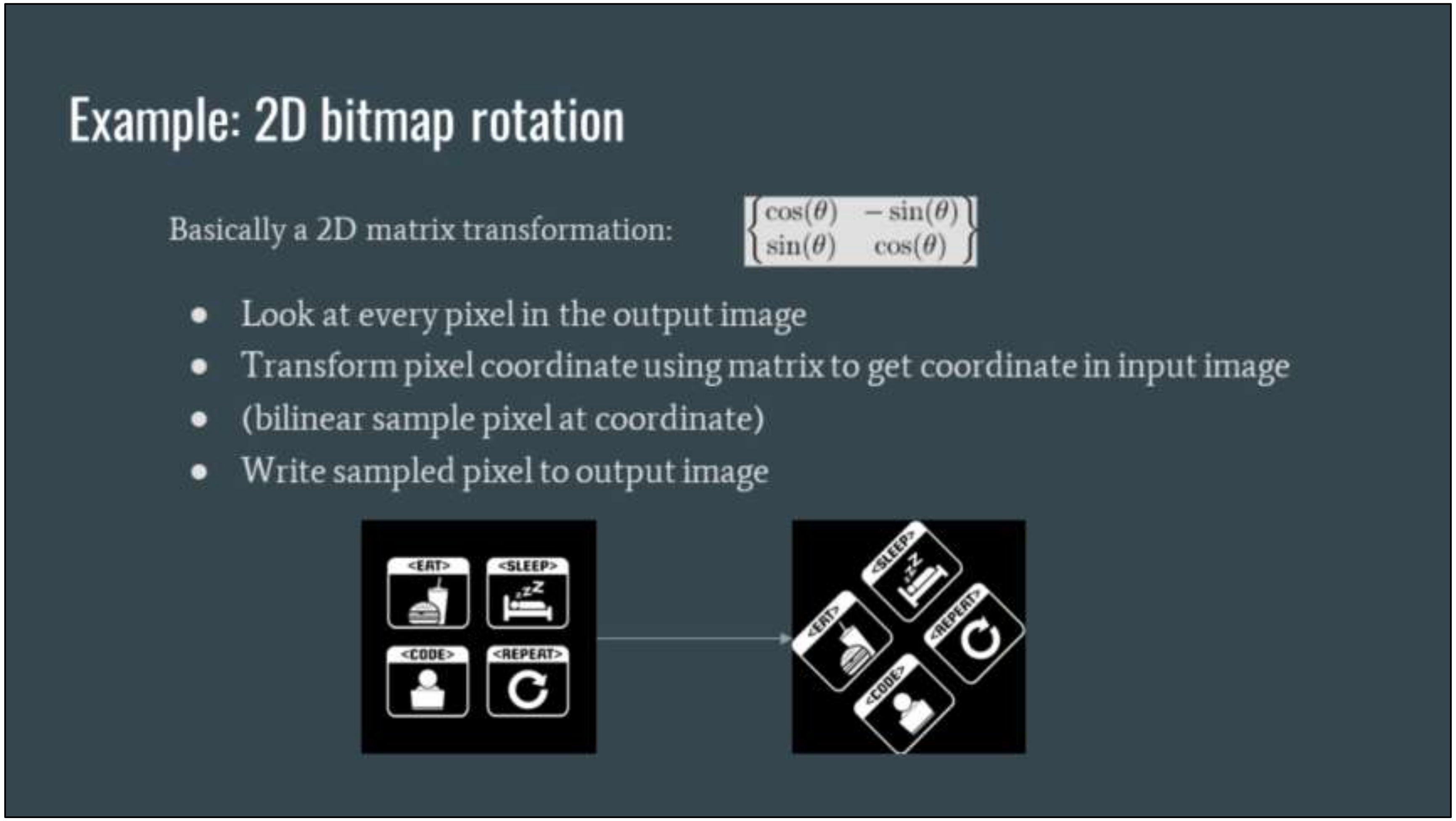
Bilinear Sampling:



Lets define a concrete example that we'll use as our problem to optimize in this talk.

For this talk we'll be looking at an algorithm that rotates a 2D bitmap by an arbitrary angle.
Since some sampling has to be done a bilinear sampling should be implemented.

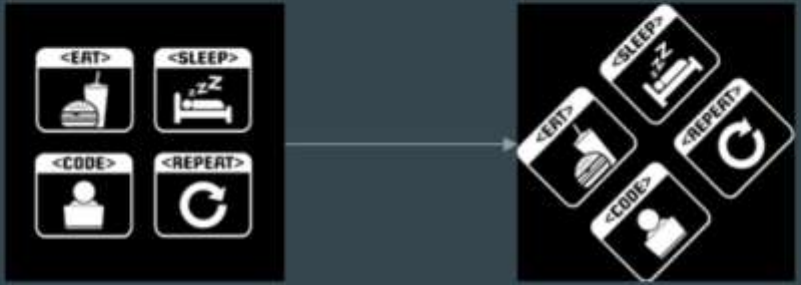
The input for our algorithm will be an image and a transformation matrix that represents the rotation
The output will be the rotated image



Example: 2D bitmap rotation

Basically a 2D matrix transformation: $\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$

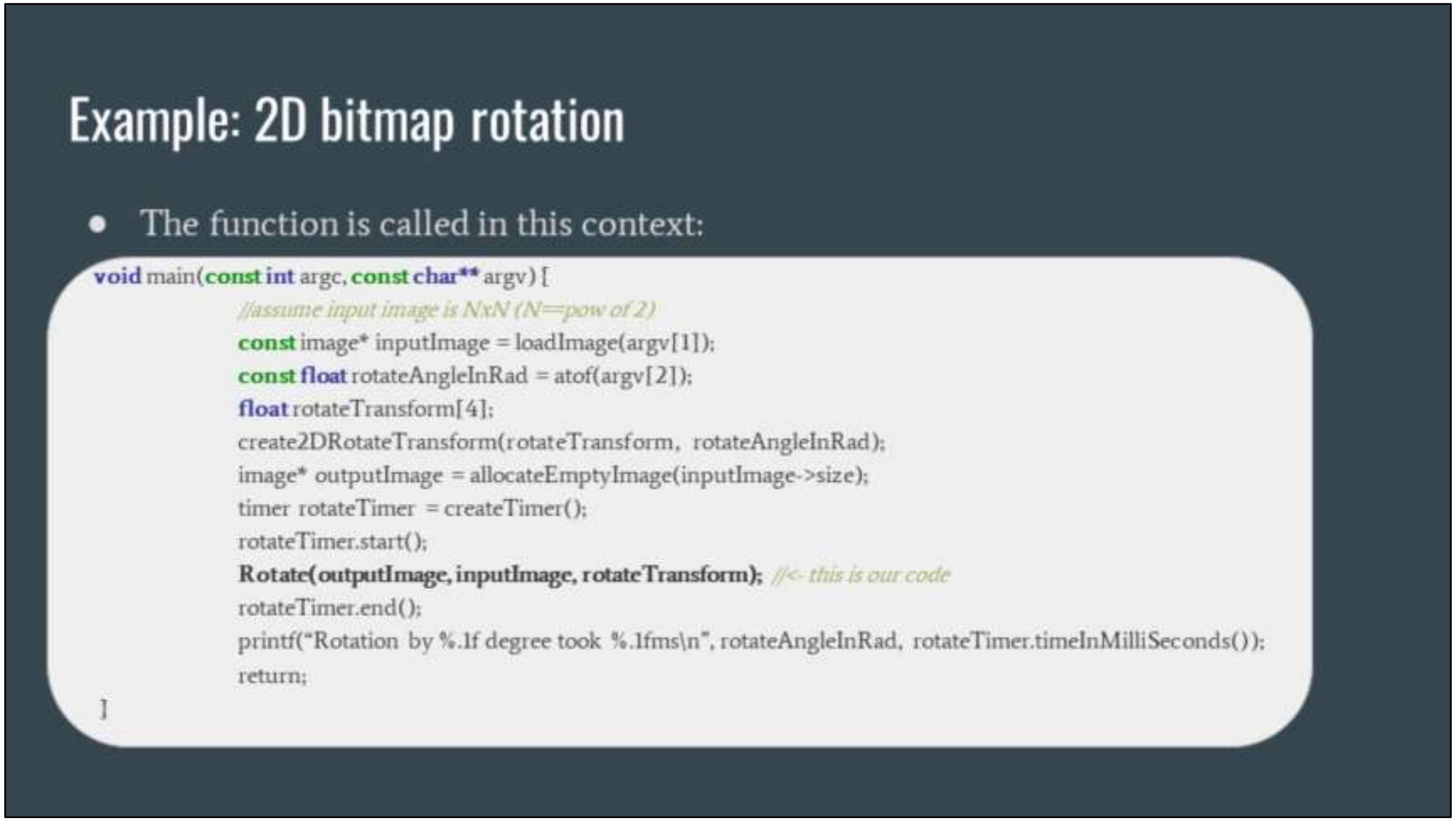
- Look at every pixel in the output image
- Transform pixel coordinate using matrix to get coordinate in input image
- (bilinear sample pixel at coordinate)
- Write sampled pixel to output image



The algorithm itself is basically a 2D matrix transform
And the logic is:

Look at each pixel,
Transform the coordinates
Sample pixel from input image using transformed coordinates
Write sample to output pixel

The expected result is shown in the picture



Example: 2D bitmap rotation

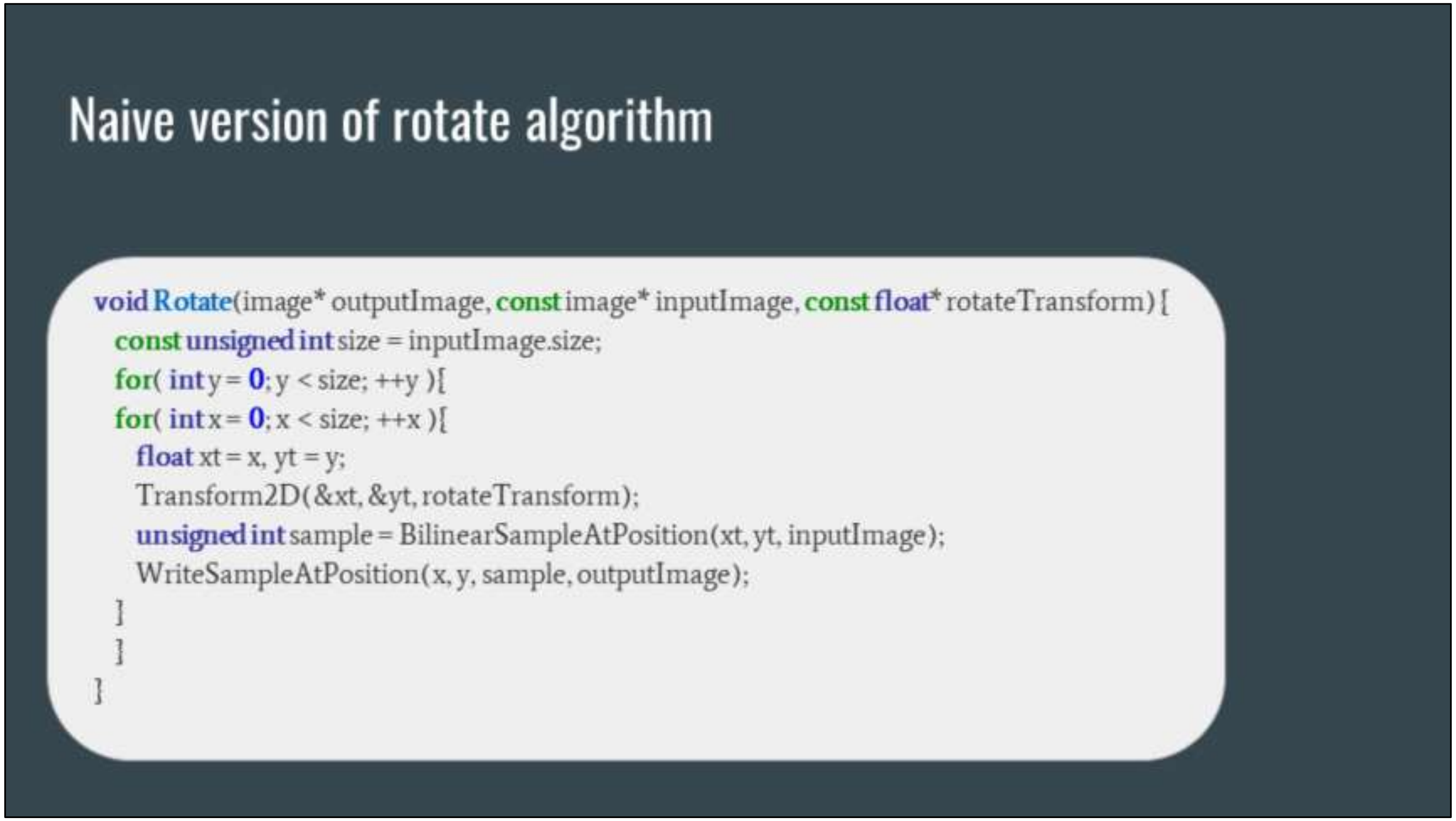
- The function is called in this context:

```
void main(const int argc, const char** argv) {
        //assume input image is NxN (N==pow of 2)
        const image* inputImage = loadImage(argv[1]);
        const float rotateAngleInRad = atof(argv[2]);
        float rotateTransform[4];
        create2DRotateTransform(rotateTransform, rotateAngleInRad);
        image* outputImage = allocateEmptyImage(inputImage->size);
        timer rotateTimer = createTimer();
        rotateTimer.start();
        Rotate(outputImage, inputImage, rotateTransform); //<- this is our code
        rotateTimer.end();
        printf("Rotation by %.1f degree took %.1fms\n", rotateAngleInRad, rotateTimer.timeInMilliSeconds());
        return;
}
```

This will be the context in which our function is called.
The image that we'll be rotating will be loaded from disc and the 2D rotation matrix will be computed based on the angle that we should rotate the image by.

After that a function named Rotate() is being called.
This will be the function that we'll implement and optimize during this talk.

To get some performance data, we'll time this function and print the duration in milliseconds.



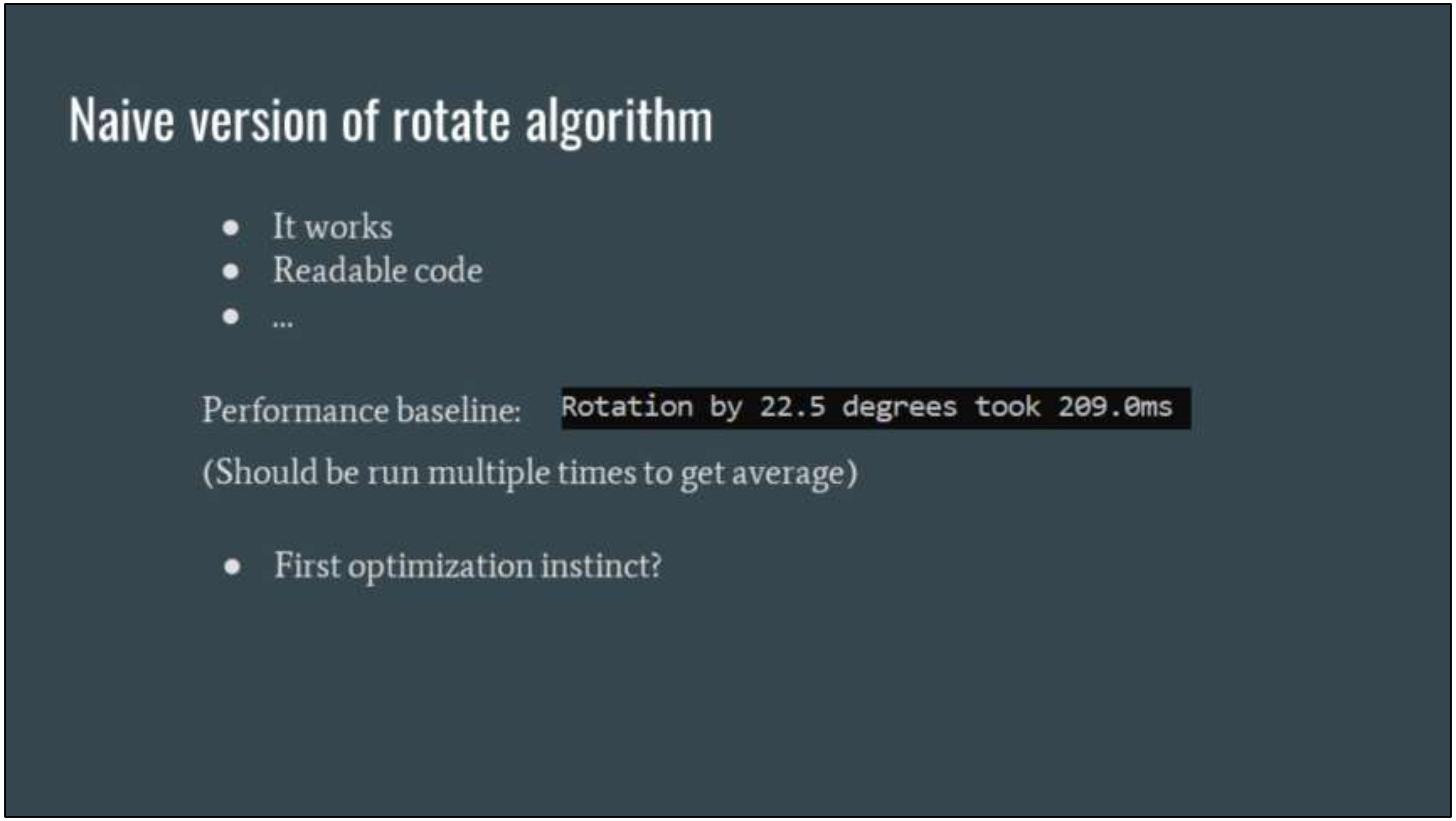
Naive version of rotate algorithm

```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform) {
  const unsigned int size = inputImage.size;
  for( int y = 0; y < size; ++y ){
  for( int x = 0; x < size; ++x ){
    float xt = x, yt = y;
    Transform2D(&xt, &yt, rotateTransform);
    unsigned int sample = BilinearSampleAtPosition(xt, yt, inputImage);
    WriteSampleAtPosition(x, y, sample, outputImage);
  }
  }
}
```

Let's first start with a naive version of the algorithm.

To keep things simple I assume that Transform2D, BilinearSampleAtPosition and WriteSampleAtPosition are all given and are being inlined by the compiler.
The implementation details of those function doesn't *really* matter for the first couple of optimizations that we'll apply.

For completeness:
Transform2D transforms a 2D coordinate using the given 2x2 transformation matrix
BilinearSampleAtPosition performs a bilinear sampling at the given coordinate (+ performance mirror addressing if the coordinate is out of bounds)
WriteSampleAtPosition writes the sample at the given coordinate to the image



Naive version of rotate algorithm

- It works
- Readable code
- ...

Performance baseline:

```
Rotation by 22.5 degrees took 209.0ms
```

(Should be run multiple times to get average)

- First optimization instinct?

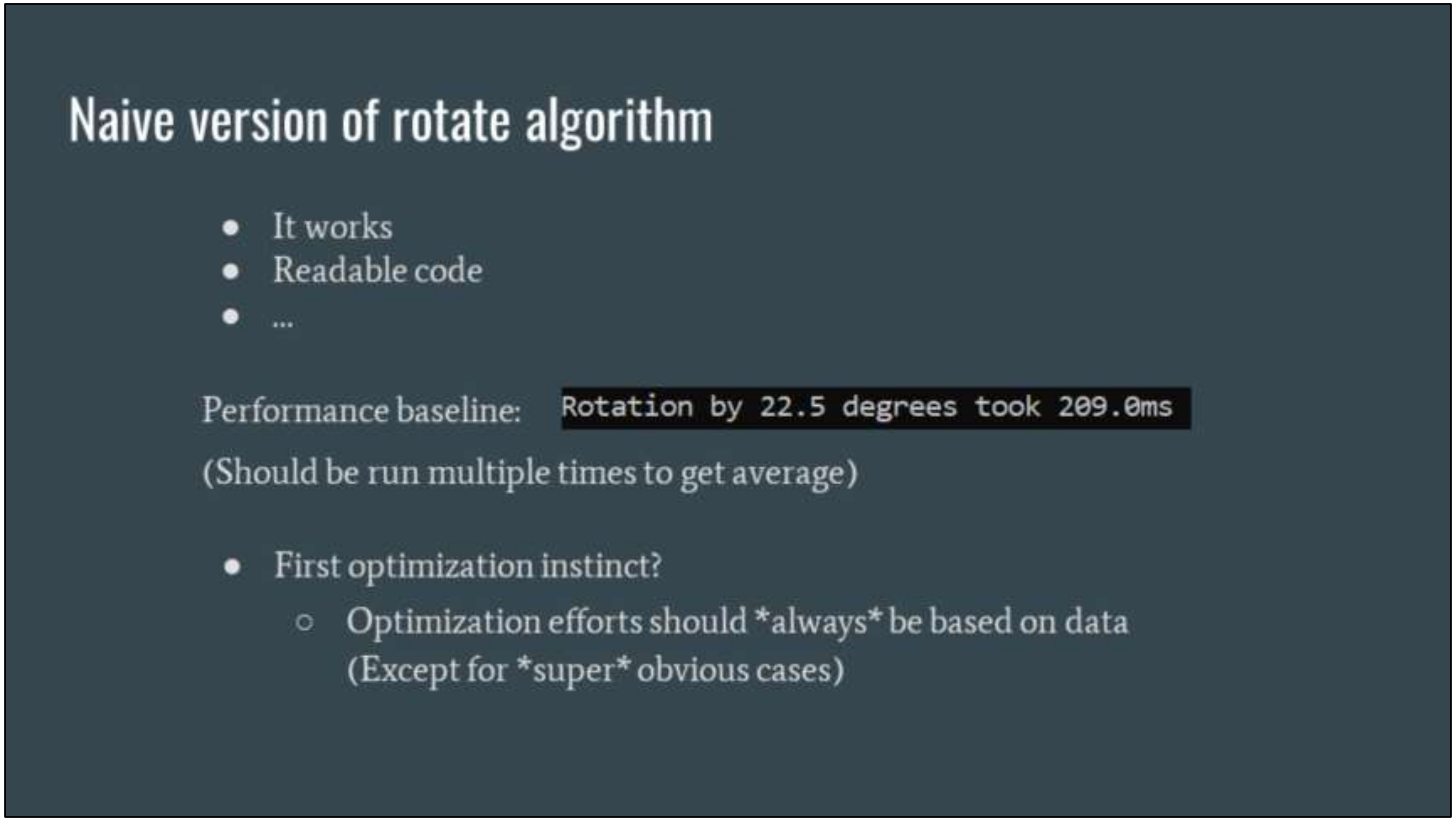
Let’s review this approach:

It works
I would argue that the code is readable and easy to follow
But other than that, that’s about it

This is our current performance baseline - around 205ms. A far cry from real time (real time would be if this function executes faster than 16.6ms for 60hz or 33.3ms for 30hz).
Since there will always be some variation in the timing, this should be run multiple times - the resources of your CPU are shared by all running processes so variations here are expected.

Question to the audience - What would be your first optimization instinct?
discuss with audience

This is all fine and good but in general all optimization efforts should always be backed up by data - except for super obvious cases like iterating over a 2d array in a column major order.



Naive version of rotate algorithm

- It works
- Readable code
- ...

Performance baseline:

```
Rotation by 22.5 degrees took 209.0ms
```

(Should be run multiple times to get average)

- First optimization instinct?
    - Optimization efforts should *always* be based on data (Except for *super* obvious cases)

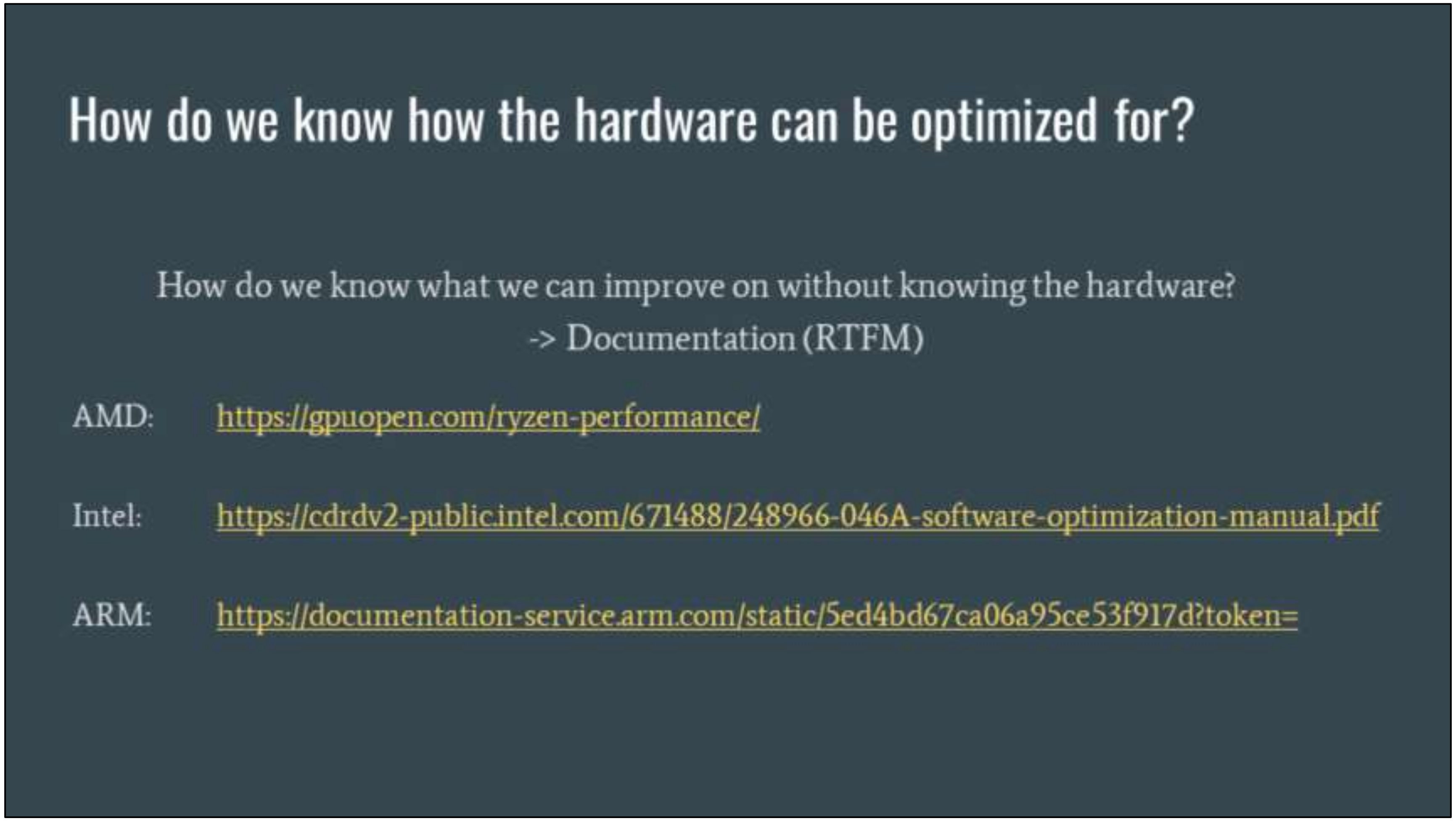
Let’s review this approach:

It works
I would argue that the code is readable and easy to follow
But other than that, that’s about it

This is our current performance baseline - around 205ms. A far cry from real time (real time would be if this function executes faster than 16.6ms for 60hz or 33.3ms for 30hz).
Since there will always be some variation in the timing, this should be run multiple times - the resources of your CPU are shared by all running processes so variations here are expected.

Question to the audience - What would be your first optimization instinct?
discuss with audience

This is all fine and good but in general all optimization efforts should always be backed up by data - except for super obvious cases like iterating over a 2d array in a column major order.



How do we know how the hardware can be optimized for?

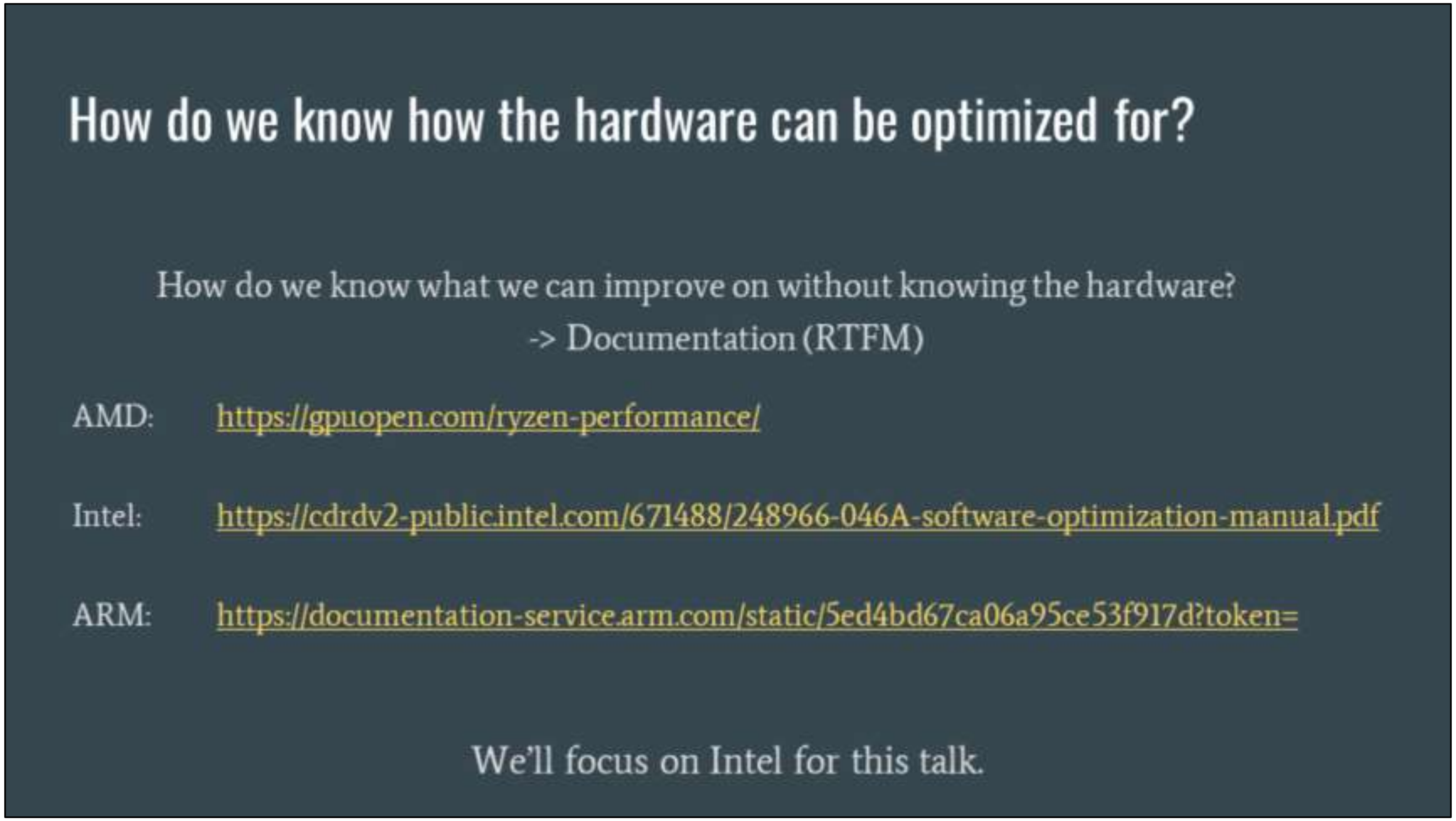
How do we know what we can improve on without knowing the hardware?
-> Documentation (RTFM)

AMD:	https://gpuopen.com/ryzen-performance/
Intel:	https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf
ARM:	https://documentation-service.arm.com/static/5ed4bd67ca06a95ce53f917d?token=

But how do we know how we can utilize the hardware to get better runtime performance?
Well, I hate to break it to you but at the core you'll have to read the documentation *pull out printed copy of Intel Software Optimization Guide*

Yes, this big book - but don't be discourage by this, you really don't need to know *all* of it and there are some talks out there that talk about some
Of the informations in here - I'll reference some of them at the end of this presentation.

Generally these are the links that you'd want to use if you're looking for the documentation for a specific CPU vendor.
For this talk we'll focus on intel because it's what I'm most familiar with.



How do we know how the hardware can be optimized for?

How do we know what we can improve on without knowing the hardware?
-> Documentation (RTFM)

AMD:	https://gpuopen.com/ryzen-performance/
Intel:	https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf
ARM:	https://documentation-service.arm.com/static/5ed4bd67ca06a95ce53f917d?token=

We'll focus on Intel for this talk.

But how do we know how we can utilize the hardware to get better runtime performance?
Well, I hate to break it to you but at the core you'll have to read the documentation *pull out printed copy of Intel Software Optimization Guide*

Yes, this big book - but don't be discourage by this, you really don't need to know *all* of it and there are some talks out there that talk about some
Of the informations in here - I'll reference some of them at the end of this presentation.

Generally these are the links that you'd want to use if you're looking for the documentation for a specific CPU vendor.
For this talk we'll focus on intel because it's what I'm most familiar with.

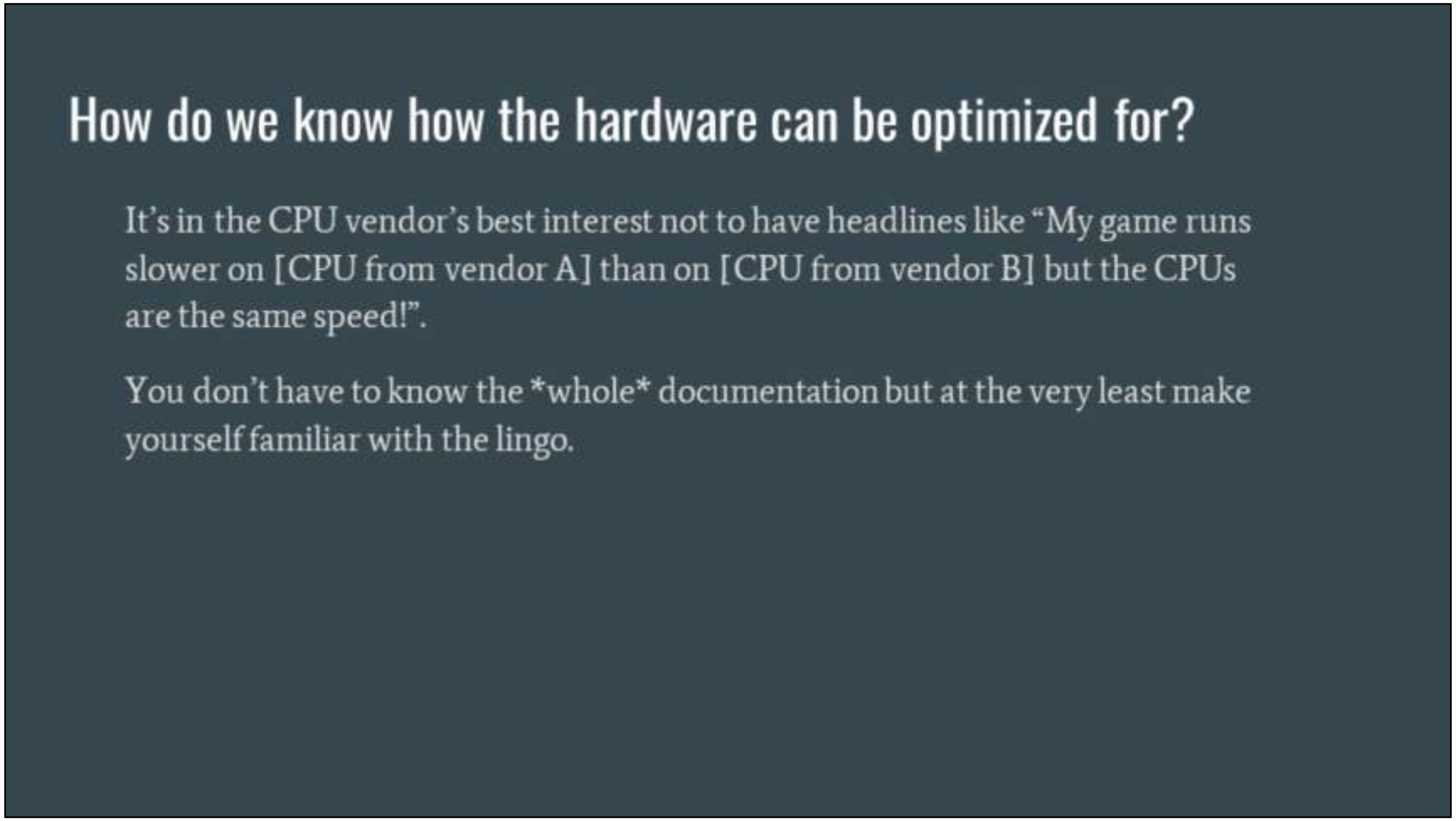
How do we know how the hardware can be optimized for?

It's in the CPU vendor's best interest not to have headlines like "My game runs slower on [CPU from vendor A] than on [CPU from vendor B] but the CPUs are the same speed!".

The reason these documentations are so big is of course because the CPU vendor doesn't want to see headlines like this.
And again, you don't need to know the *whole* documentation but you should familiarize yourself with the lingo.

Knowing the hardware will let you make more subtle changes to your code to reach your performance goals without having to pull out the SIMD hammer as your first instinct.

To get more in-depth hardware base performance data you can use one of these vendor specific profilers.



How do we know how the hardware can be optimized for?

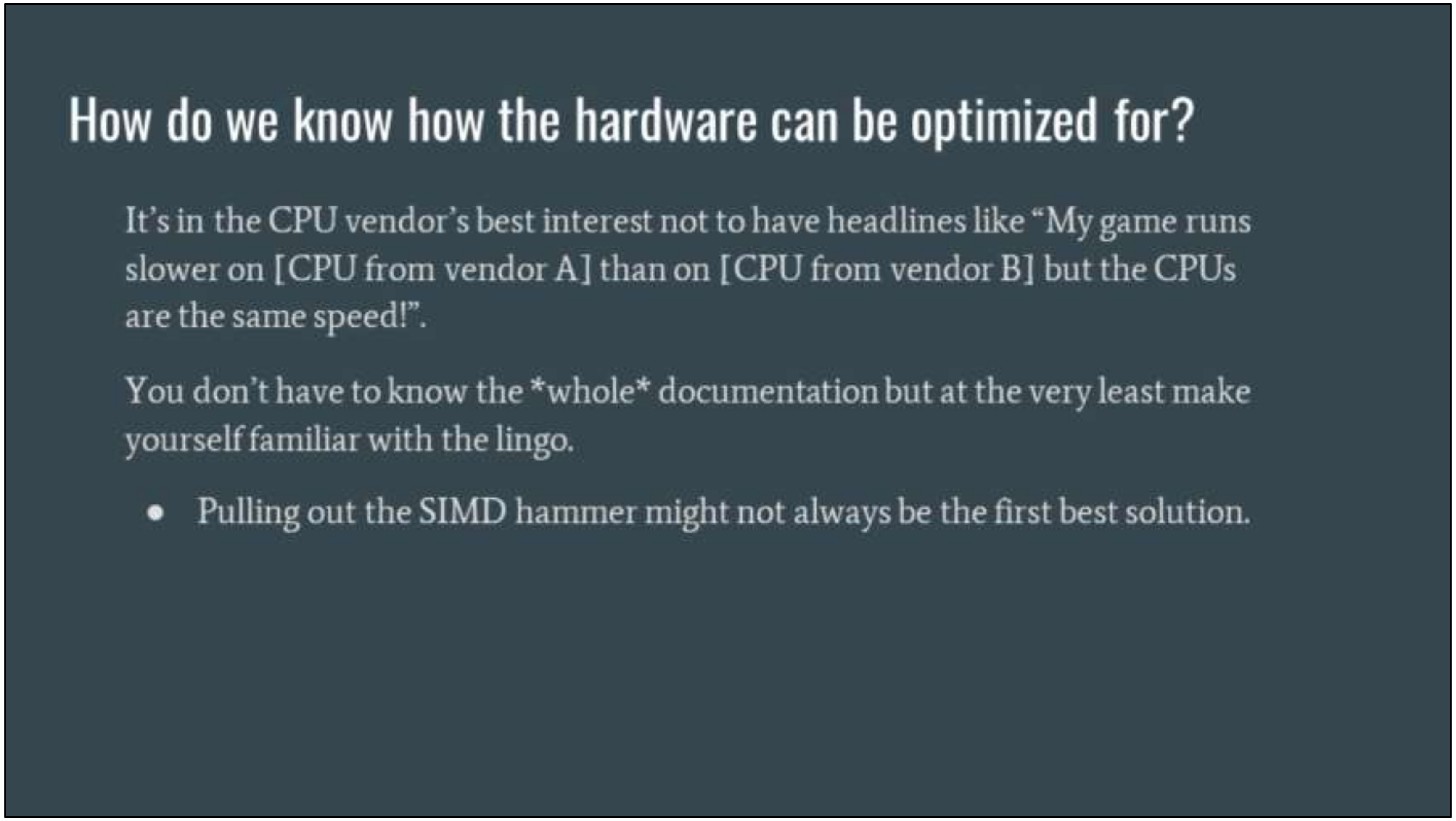
It's in the CPU vendor's best interest not to have headlines like "My game runs slower on [CPU from vendor A] than on [CPU from vendor B] but the CPUs are the same speed!".

You don't have to know the *whole* documentation but at the very least make yourself familiar with the lingo.

The reason these documentations are so big is of course because the CPU vendor doesn't want to see headlines like this.
And again, you don't need to know the *whole* documentation but you should familiarize yourself with the lingo.

Knowing the hardware will let you make more subtle changes to your code to reach your performance goals without having to pull out the SIMD hammer as your first instinct.

To get more in-depth hardware base performance data you can use one of these vendor specific profilers.



How do we know how the hardware can be optimized for?

It's in the CPU vendor's best interest not to have headlines like "My game runs slower on [CPU from vendor A] than on [CPU from vendor B] but the CPUs are the same speed!".

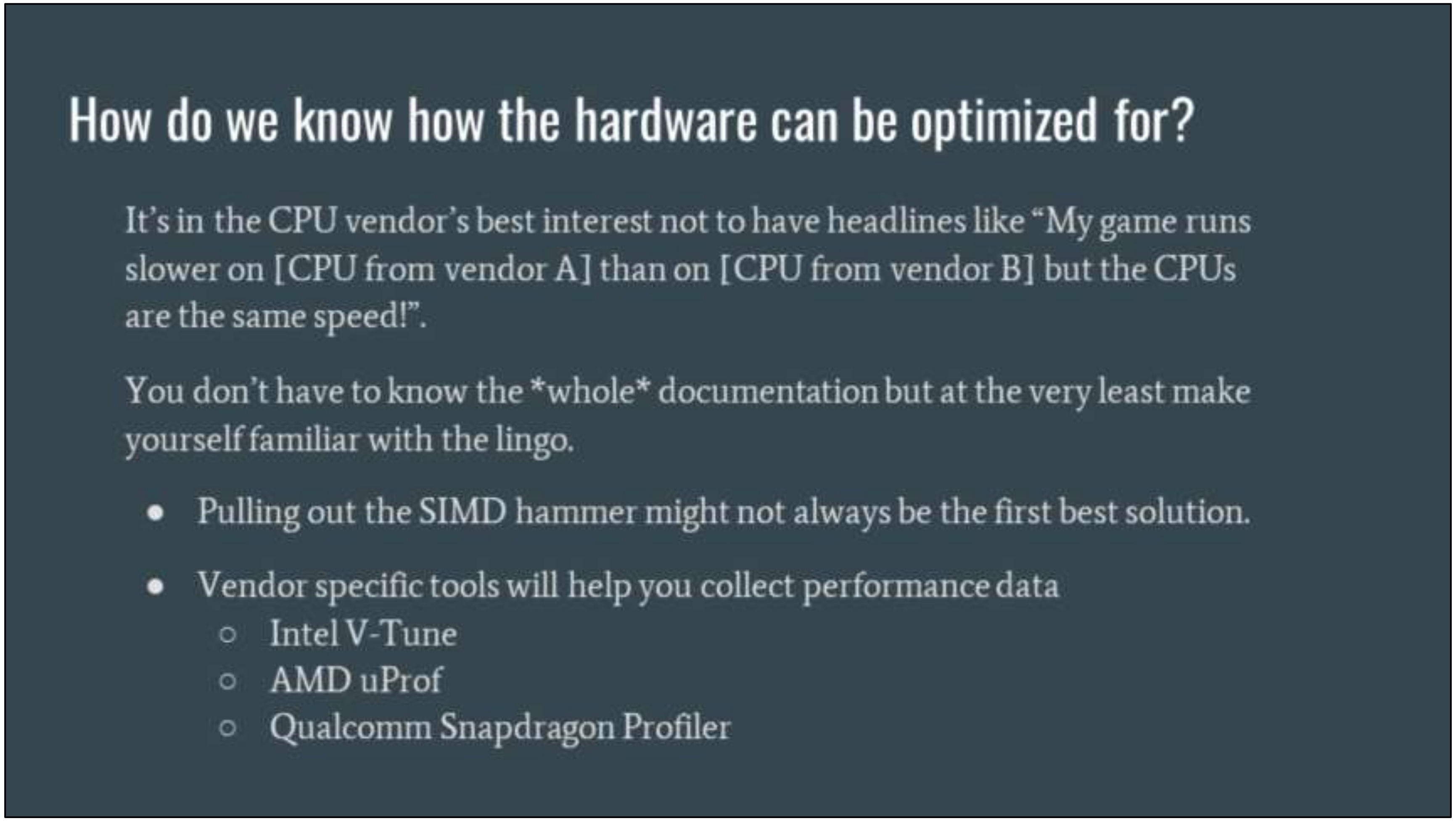
You don't have to know the *whole* documentation but at the very least make yourself familiar with the lingo.

- Pulling out the SIMD hammer might not always be the first best solution.

The reason these documentations are so big is of course because the CPU vendor doesn't want to see headlines like this.
And again, you don't need to know the *whole* documentation but you should familiarize yourself with the lingo.

Knowing the hardware will let you make more subtle changes to your code to reach your performance goals without having to pull out the SIMD hammer as your first instinct.

To get more in-depth hardware base performance data you can use one of these vendor specific profilers.



How do we know how the hardware can be optimized for?

It's in the CPU vendor's best interest not to have headlines like "My game runs slower on [CPU from vendor A] than on [CPU from vendor B] but the CPUs are the same speed!".

You don't have to know the *whole* documentation but at the very least make yourself familiar with the lingo.

- Pulling out the SIMD hammer might not always be the first best solution.
- Vendor specific tools will help you collect performance data
  - Intel V-Tune
  - AMD uProf
  - Qualcomm Snapdragon Profiler

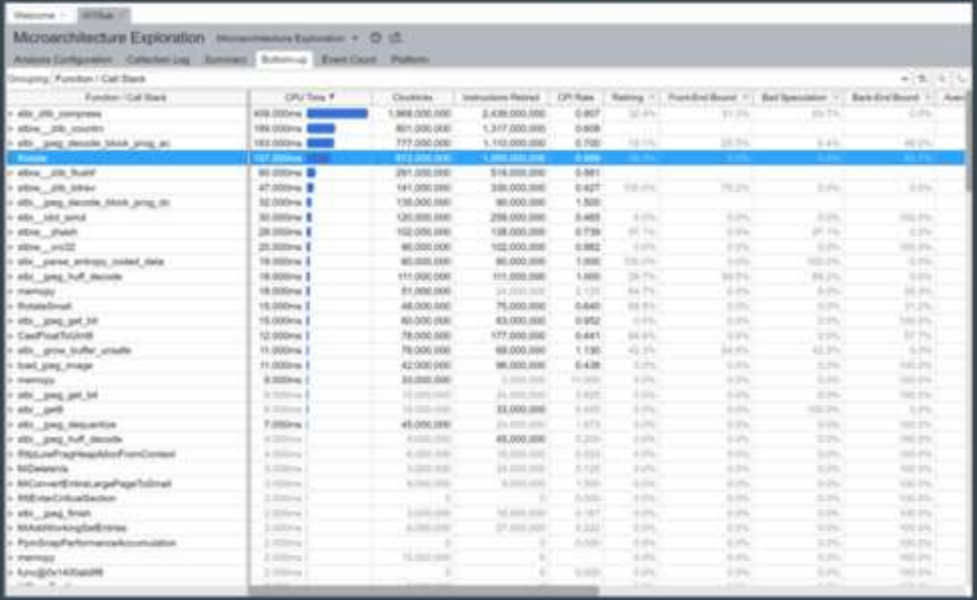
The reason these documentations are so big is of course because the CPU vendor doesn't want to see headlines like this.
And again, you don't need to know the *whole* documentation but you should familiarize yourself with the lingo.

Knowing the hardware will let you make more subtle changes to your code to reach your performance goals without having to pull out the SIMD hammer as your first instinct.

To get more in-depth hardware base performance data you can use one of these vendor specific profilers.

Optimization 1: Better Execution Unit Utilization

Running Intel V-Tune Microarchitecture Exploration to get broad idea of CPU utilization performance metrics of our program.

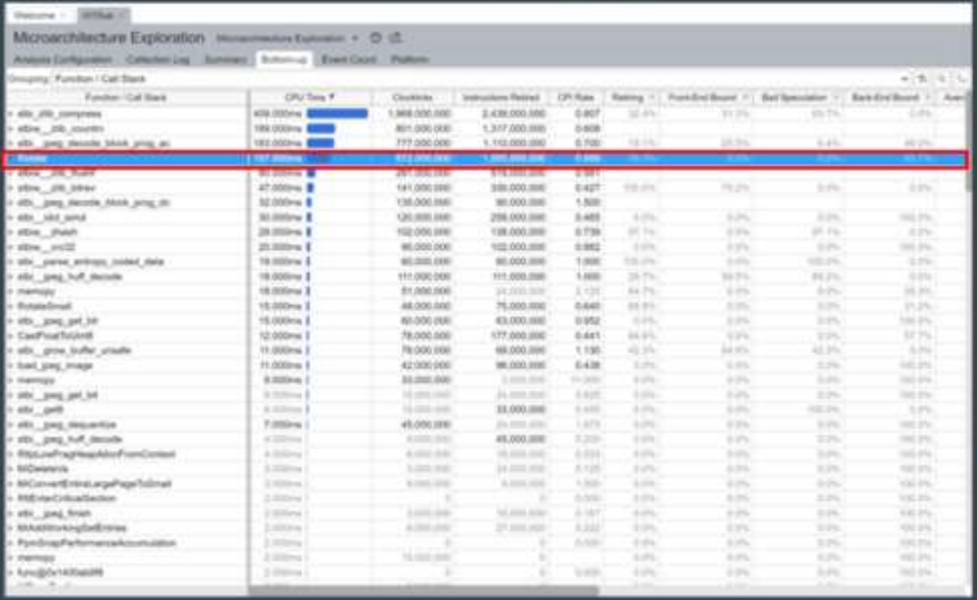


Ok, since we're focusing on intel for this talk, let's pull out V-Tune and run the "Microarchitecture Exploration" analysis to get a broad idea of the CPU utilization performance metrics of our program.

Running this on our executable will present us with this overview. This might look overwhelming at first but we'll break it down here.
First we'll have to find the function that we're interested in, which we can find right here.

Optimization 1: Better Execution Unit Utilization

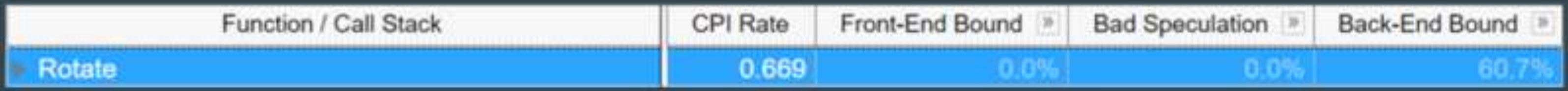
Running Intel V-Tune Microarchitecture Exploration to get broad idea of CPU utilization performance metrics of our program.



Ok, since we're focusing on intel for this talk, let's pull out V-Tune and run the "Microarchitecture Exploration" analysis to get a broad idea of the CPU utilization performance metrics of our program.

Running this on our executable will present us with this overview. This might look overwhelming at first but we'll break it down here.
First we'll have to find the function that we're interested in, which we can find right here.

Optimization 1: Better Execution Unit Utilization



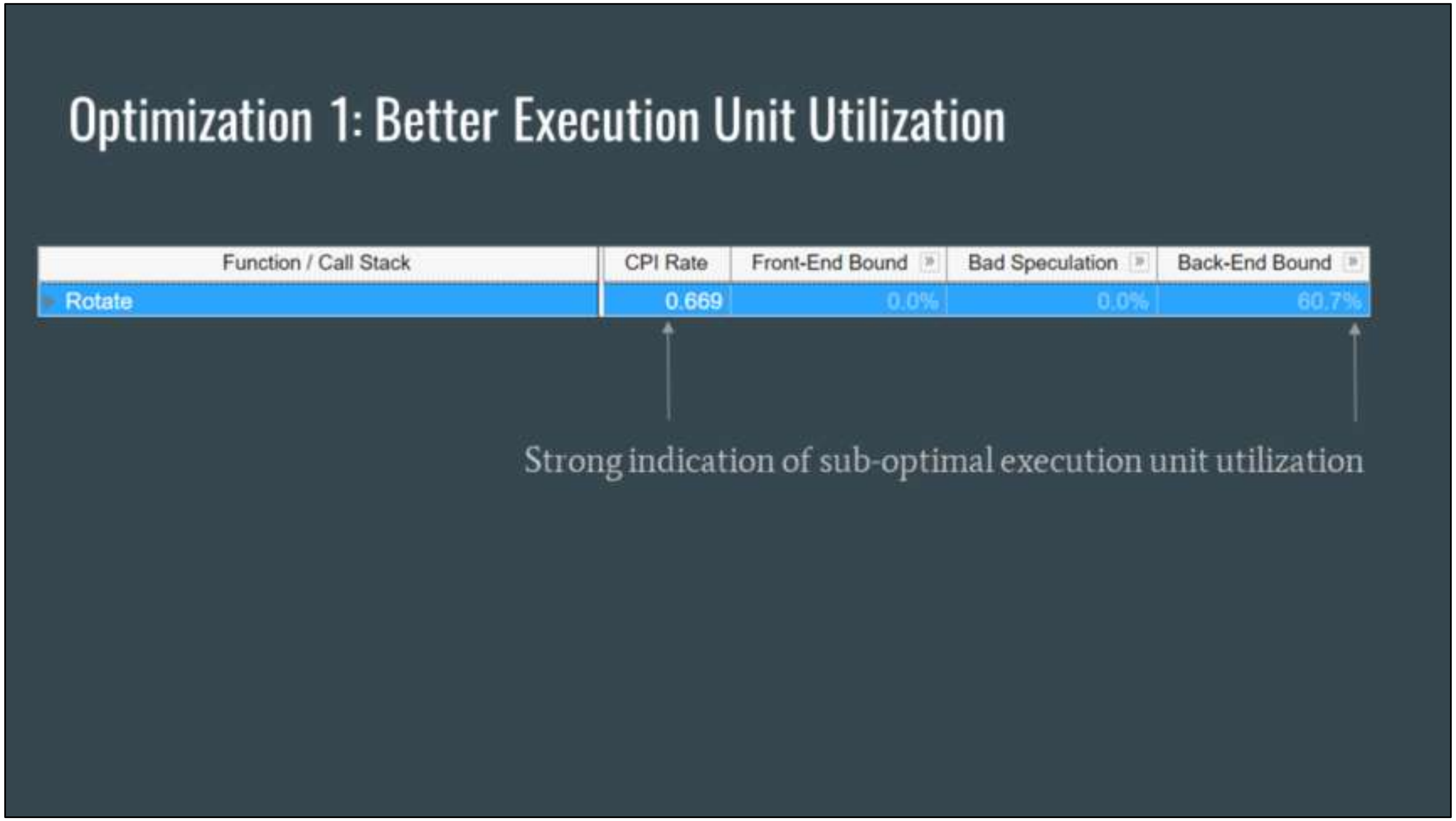
Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Let’s enhance what we see.

Let’s focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there’s some room for improvements when it comes to execution unit utilization.

If you don’t know what an execution unit is, I’ll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it’s one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.



Optimization 1: Better Execution Unit Utilization

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Strong indication of sub-optimal execution unit utilization

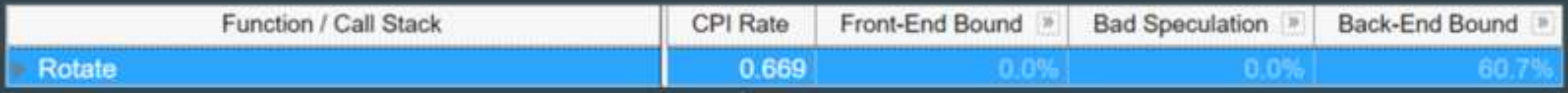
Let's enhance what we see.

Let's focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there's some room for improvements when it comes to execution unit utilization.

If you don't know what an execution unit is, I'll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it's one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.

Optimization 1: Better Execution Unit Utilization



Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Strong indication of sub-optimal execution unit utilization

CPI: Clocks per Instruction, the lower the better.

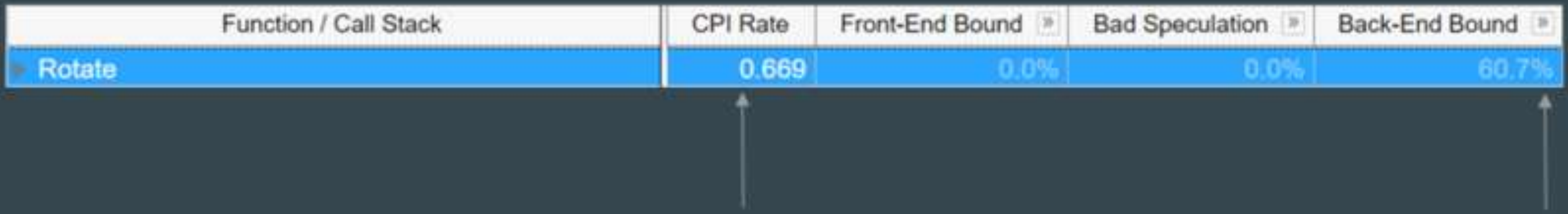
Let's enhance what we see.

Let's focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there's some room for improvements when it comes to execution unit utilization.

If you don't know what an execution unit is, I'll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it's one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.

Optimization 1: Better Execution Unit Utilization



Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Strong indication of sub-optimal execution unit utilization

CPI: Clocks per Instruction, the lower the better.

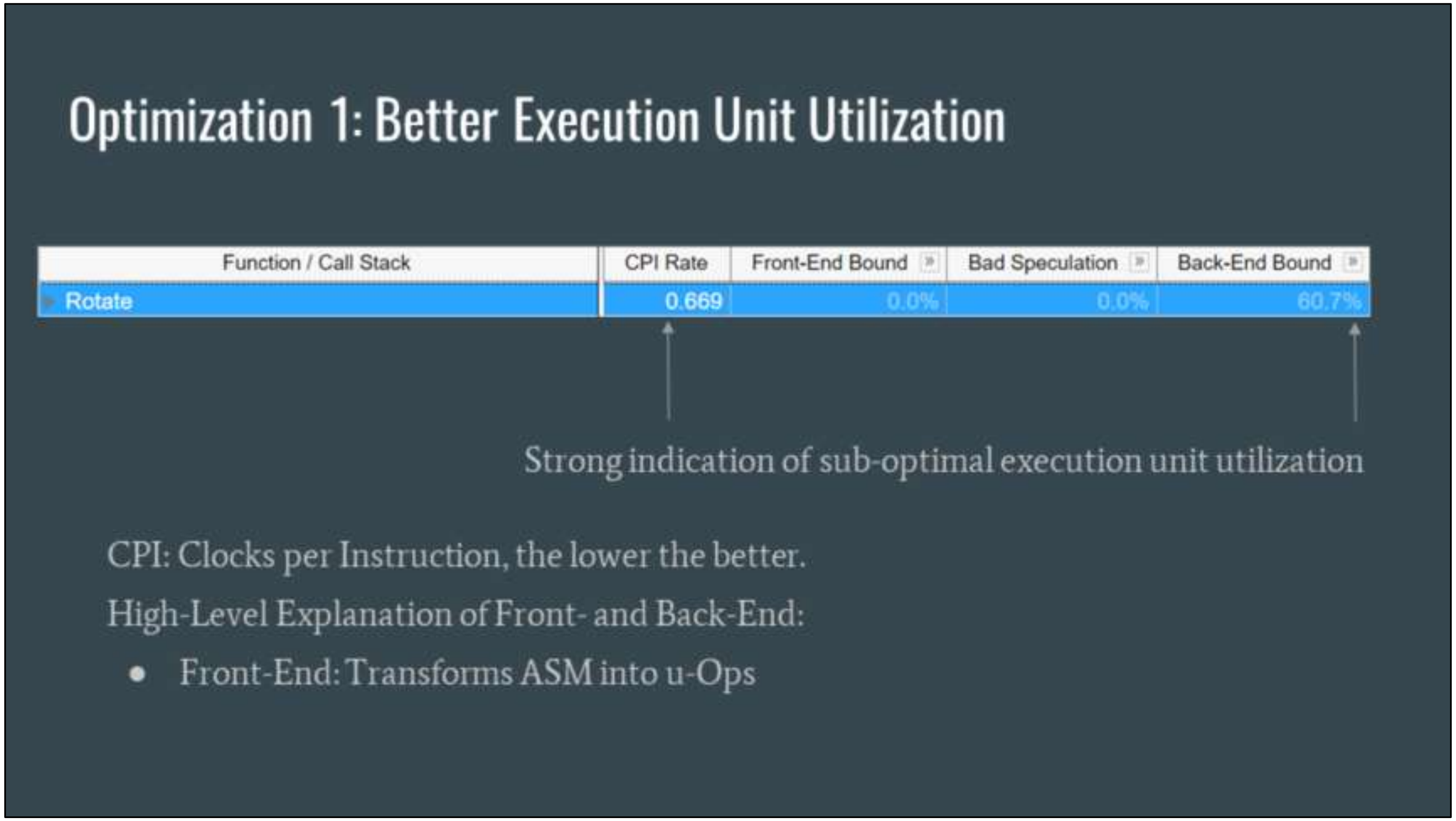
High-Level Explanation of Front- and Back-End:

Let’s enhance what we see.

Let’s focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there’s some room for improvements when it comes to execution unit utilization.

If you don’t know what an execution unit is, I’ll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it’s one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.



Optimization 1: Better Execution Unit Utilization

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Strong indication of sub-optimal execution unit utilization

CPI: Clocks per Instruction, the lower the better.

High-Level Explanation of Front- and Back-End:

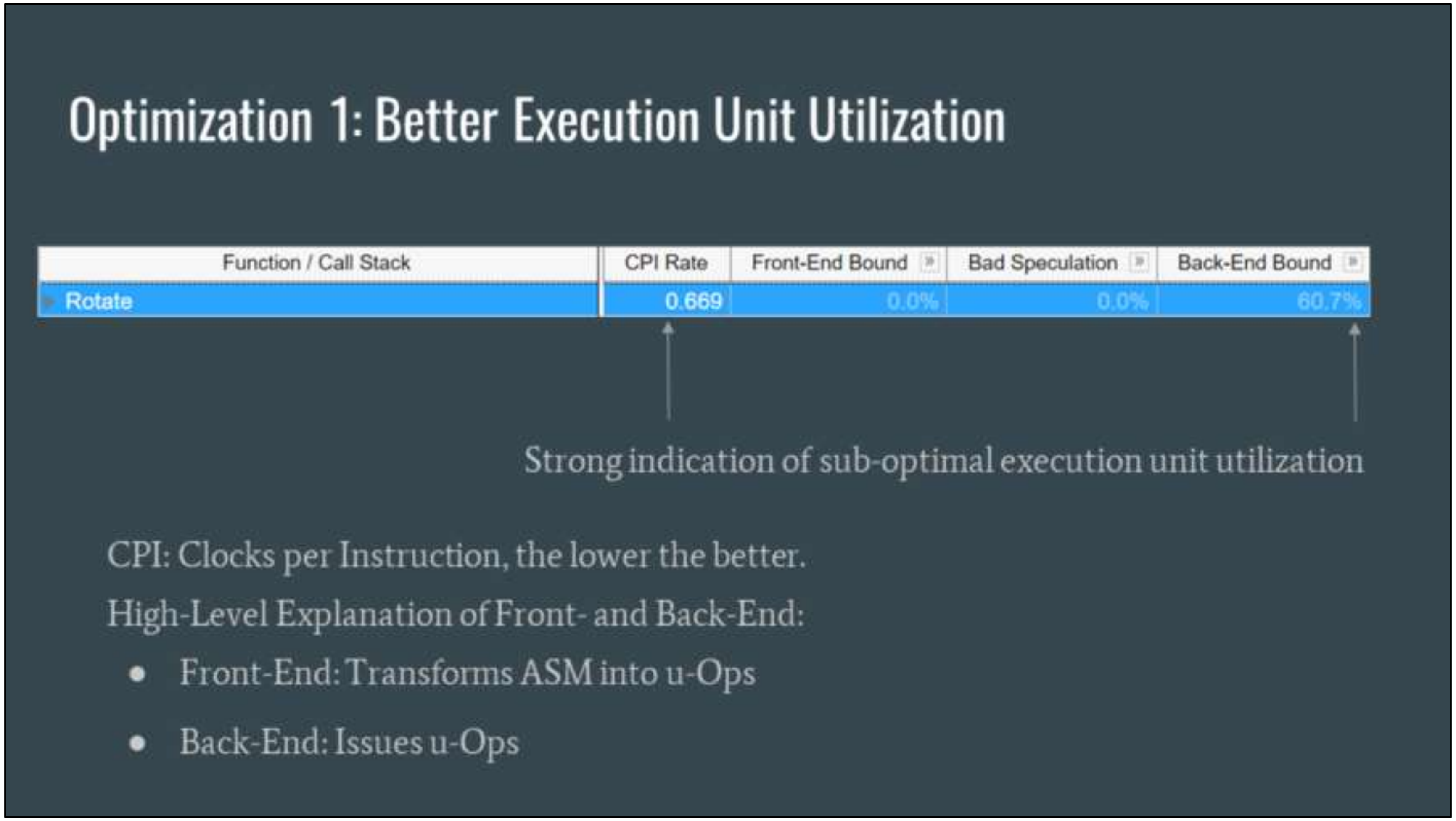
- Front-End: Transforms ASM into u-Ops

Let’s enhance what we see.

Let’s focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there’s some room for improvements when it comes to execution unit utilization.

If you don’t know what an execution unit is, I’ll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it’s one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.



Optimization 1: Better Execution Unit Utilization

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

Strong indication of sub-optimal execution unit utilization

CPI: Clocks per Instruction, the lower the better.

High-Level Explanation of Front- and Back-End:

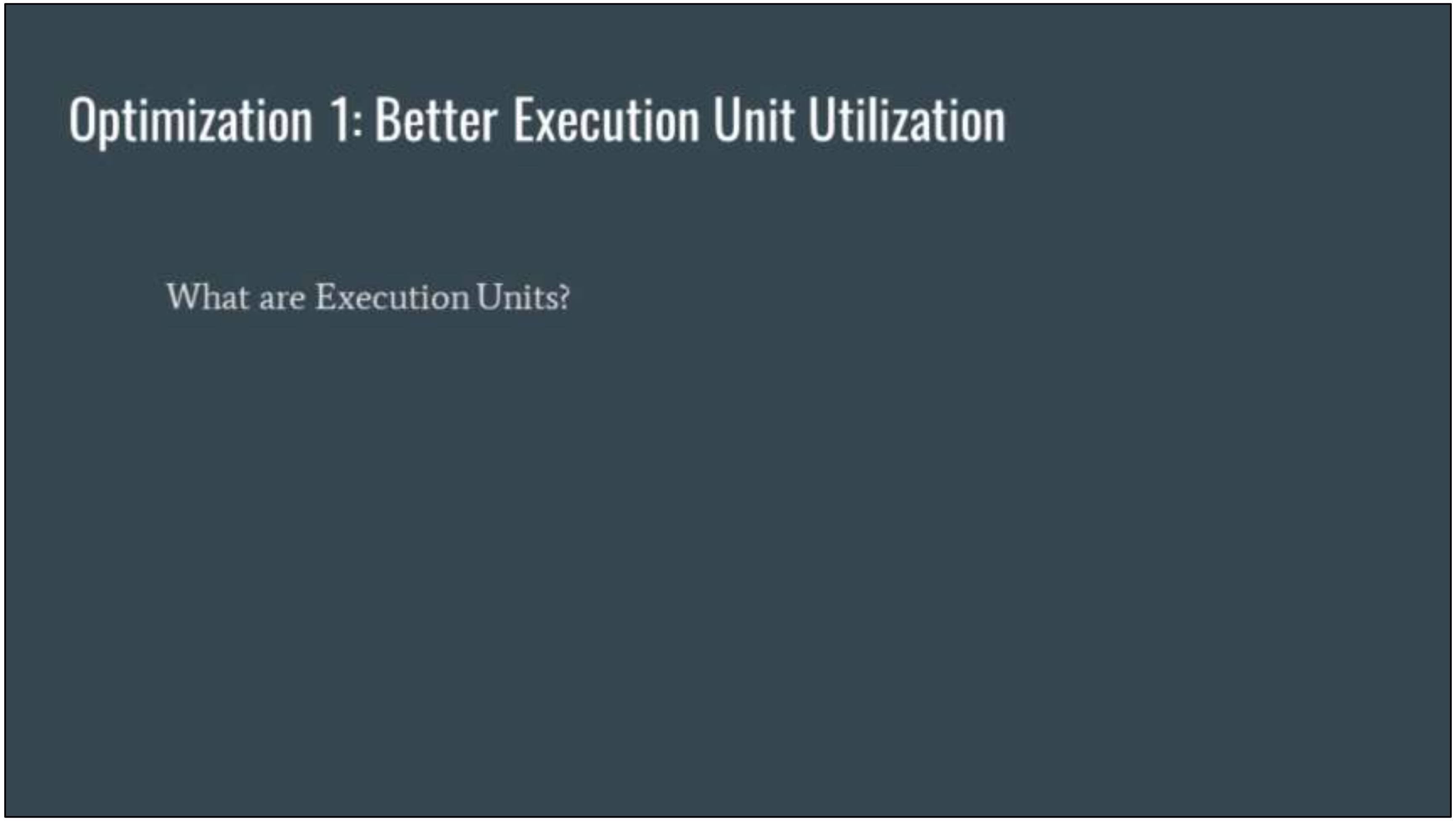
- Front-End: Transforms ASM into u-Ops
- Back-End: Issues u-Ops

Let’s enhance what we see.

Let’s focus on the CPI Rate and Front- & Back-End utlization of our function.
These numbers indicate that there’s some room for improvements when it comes to execution unit utilization.

If you don’t know what an execution unit is, I’ll explain it on the next slide.

But let me quickly explain what CPI, Front- and Back-End is.
CPI is the Clocks per Instruction and in general, the lower this number is, the better
Front-End is what converts the ASM code, generated by the compiler, into micro-operations (u-ops). You can think of u-ops as the native language of your CPU - it’s one layer below ASM.
The Back-End is then what actually executes these u-ops by using the available execution units.



Optimization 1: Better Execution Unit Utilization

What are Execution Units?

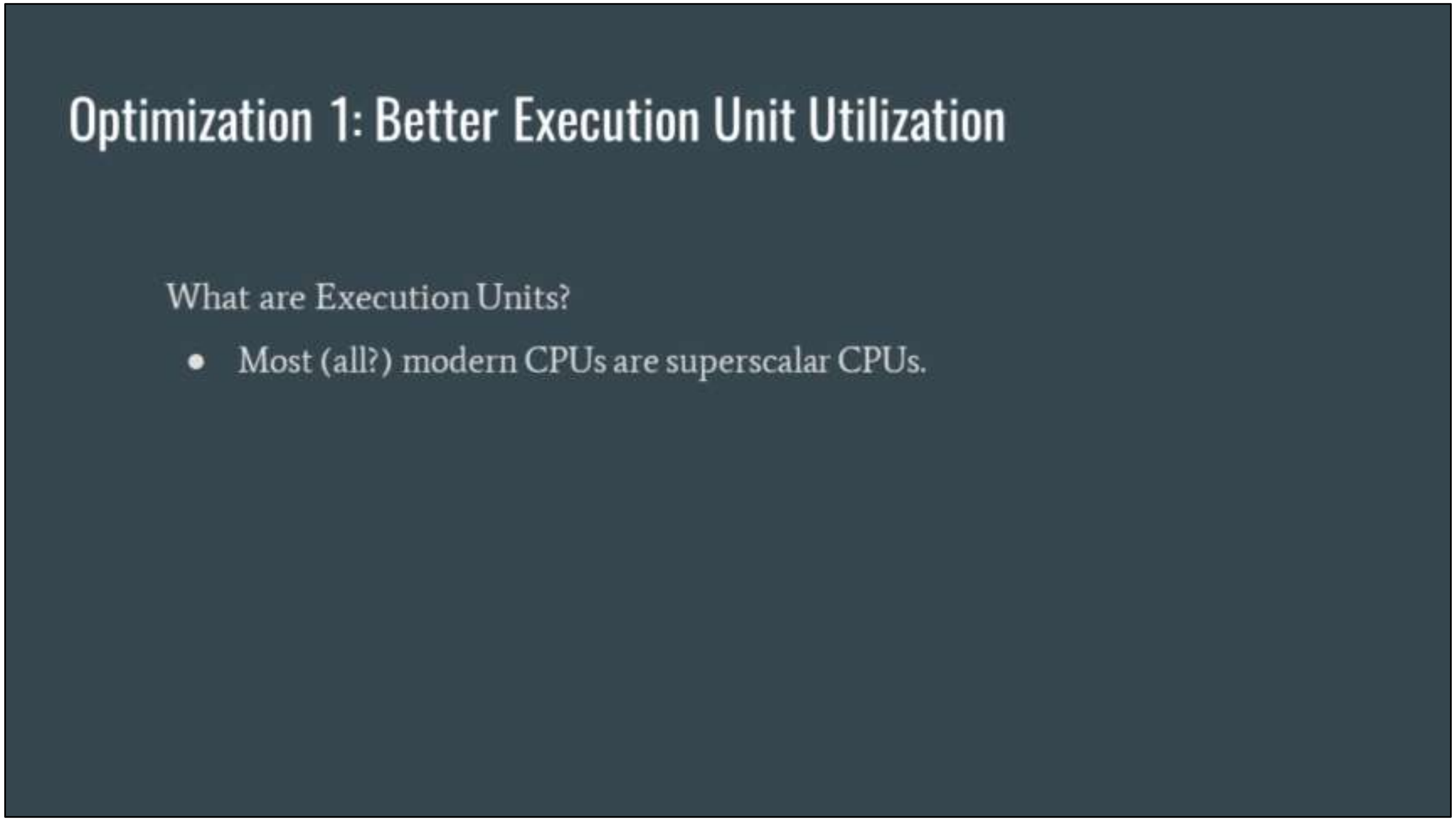
So let me quickly talk about what an execution unit actually is.

All modern CPUs are superscalar CPUs, that means that certain instruction can be run in parallel - this is called “instruction-level-parallelism”.
The available execution units indicate what instructions can be parallelized - we’ll go over what that means on the next slide.

The prerequisite for instruction-level-parallelism is that there are no dependencies between instructions.

That means that something like
A = B+C
D = A+B

Can’t be parallelized because the 2nd expression depends on the outcome of the 1st expressions



Optimization 1: Better Execution Unit Utilization

What are Execution Units?

- Most (all?) modern CPUs are superscalar CPUs.

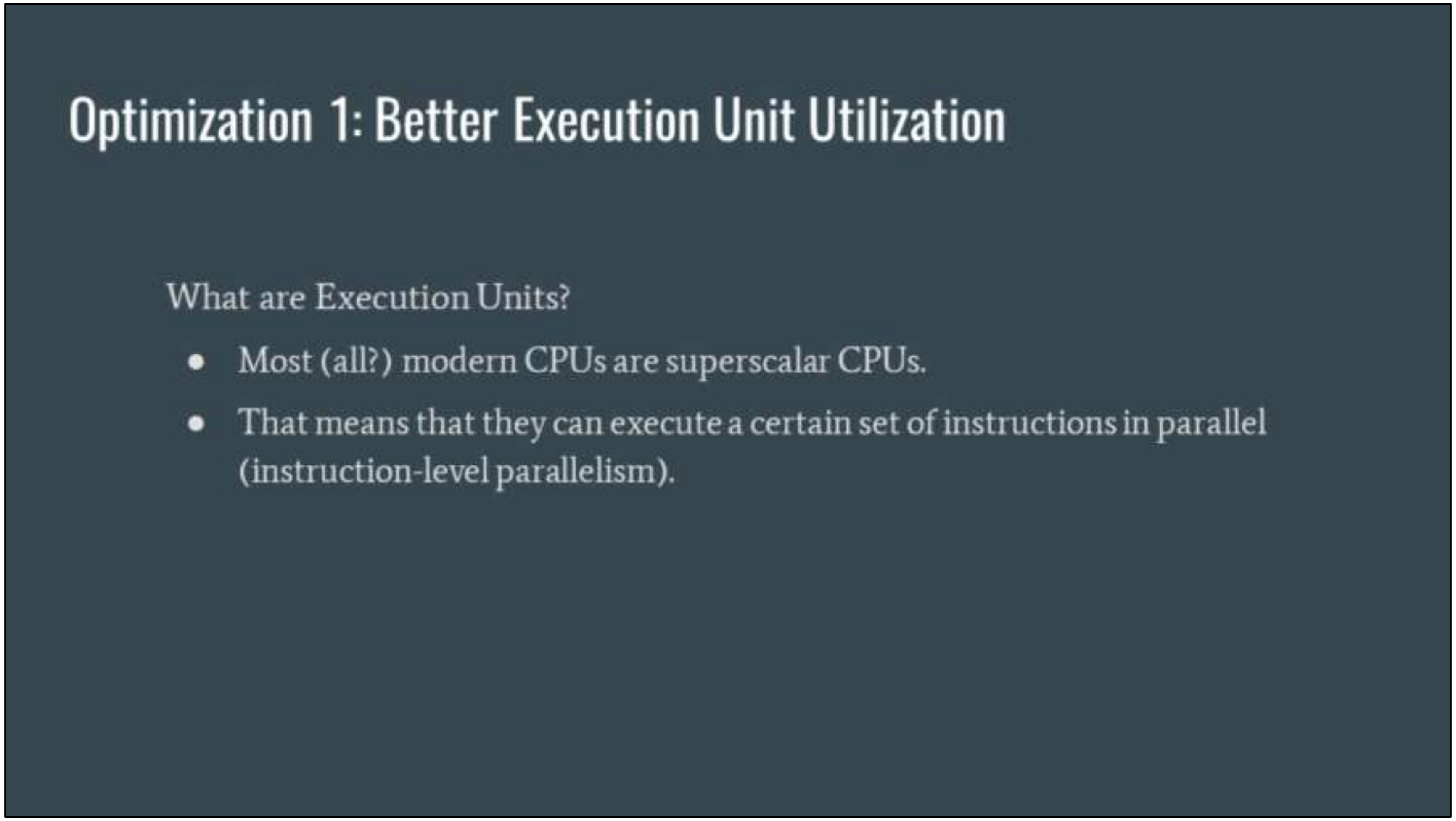
So let me quickly talk about what an execution unit actually is.

All modern CPUs are superscalar CPUs, that means that certain instruction can be run in parallel - this is called “instruction-level-parallelism”.
The available execution units indicate what instructions can be parallelized - we’ll go over what that means on the next slide.

The prerequisite for instruction-level-parallelism is that there are no dependencies between instructions.

That means that something like
A = B+C
D = A+B

Can’t be parallelized because the 2nd expression depends on the outcome of the 1st expressions



Optimization 1: Better Execution Unit Utilization

What are Execution Units?

- Most (all?) modern CPUs are superscalar CPUs.
- That means that they can execute a certain set of instructions in parallel (instruction-level parallelism).

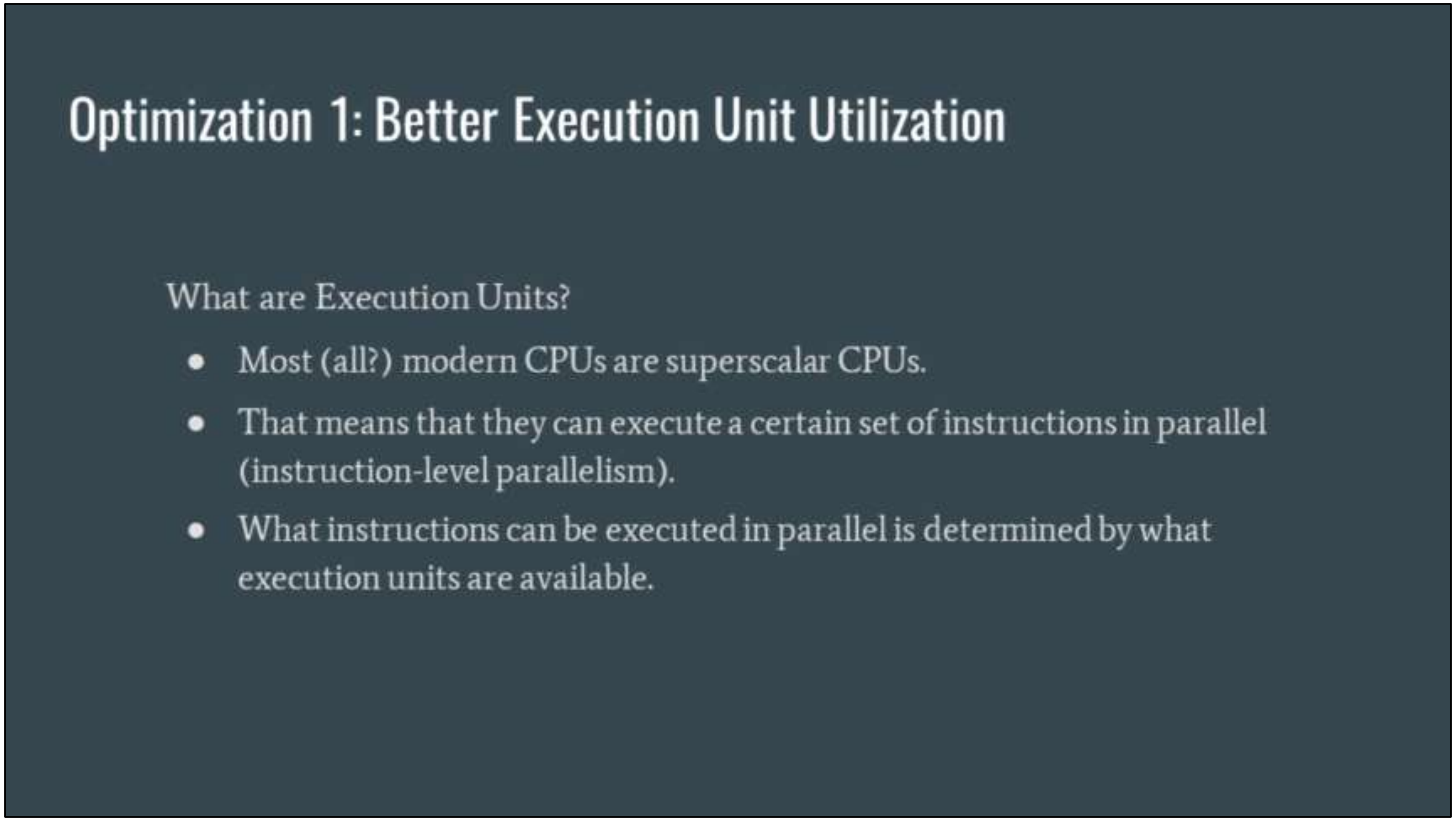
So let me quickly talk about what an execution unit actually is.

All modern CPUs are superscalar CPUs, that means that certain instruction can be run in parallel - this is called “instruction-level-parallelism”.
The available execution units indicate what instructions can be parallelized - we’ll go over what that means on the next slide.

The prerequisite for instruction-level-parallelism is that there are no dependencies between instructions.

That means that something like
A = B+C
D = A+B

Can’t be parallelized because the 2nd expression depends on the outcome of the 1st expressions



Optimization 1: Better Execution Unit Utilization

What are Execution Units?

- Most (all?) modern CPUs are superscalar CPUs.
- That means that they can execute a certain set of instructions in parallel (instruction-level parallelism).
- What instructions can be executed in parallel is determined by what execution units are available.

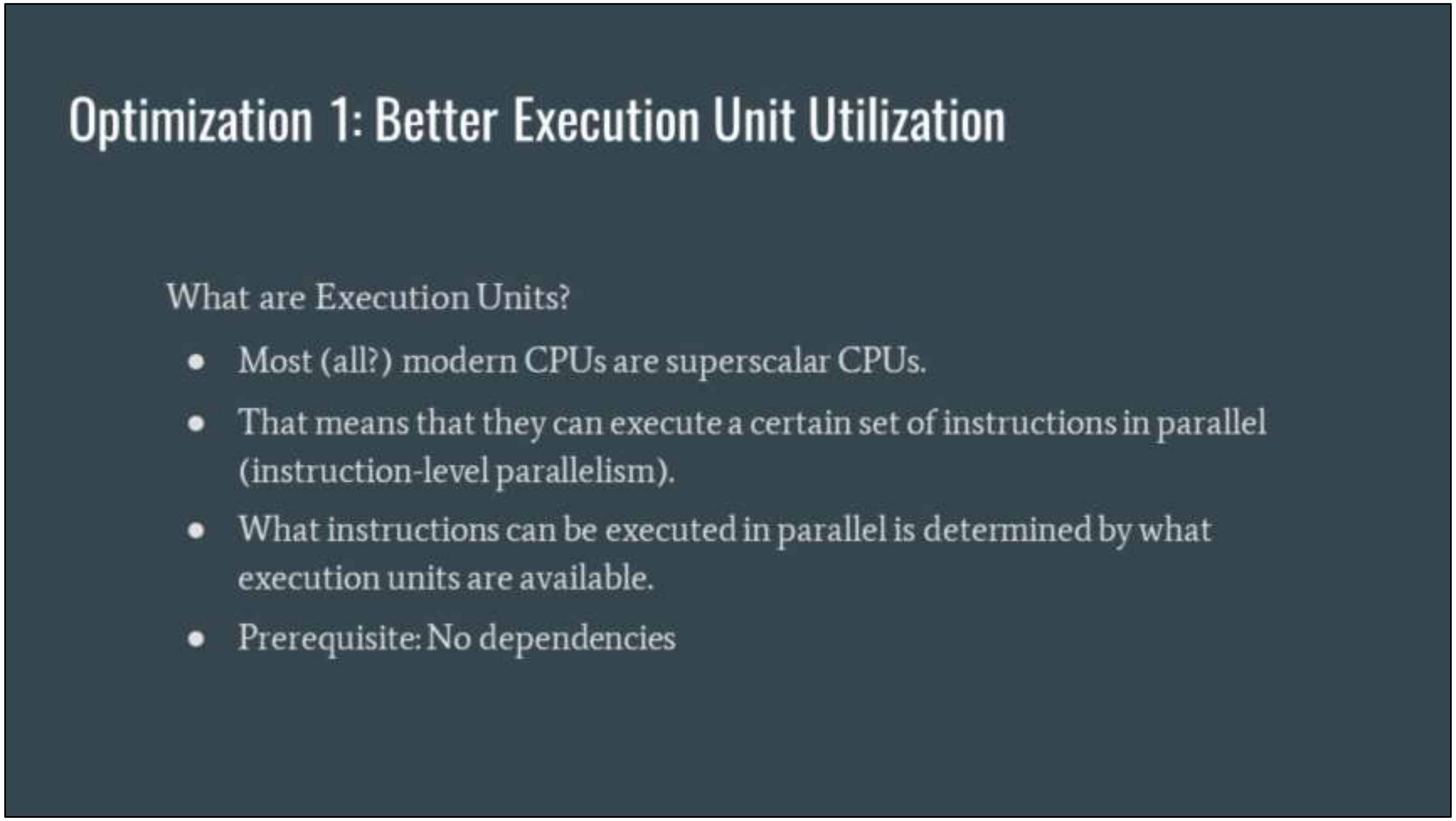
So let me quickly talk about what an execution unit actually is.

All modern CPUs are superscalar CPUs, that means that certain instruction can be run in parallel - this is called “instruction-level-parallelism”.
The available execution units indicate what instructions can be parallelized - we’ll go over what that means on the next slide.

The prerequisite for instruction-level-parallelism is that there are no dependencies between instructions.

That means that something like
A = B+C
D = A+B

Can’t be parallelized because the 2nd expression depends on the outcome of the 1st expressions



Optimization 1: Better Execution Unit Utilization

What are Execution Units?

- Most (all?) modern CPUs are superscalar CPUs.
- That means that they can execute a certain set of instructions in parallel (instruction-level parallelism).
- What instructions can be executed in parallel is determined by what execution units are available.
- Prerequisite: No dependencies

So let me quickly talk about what an execution unit actually is.

All modern CPUs are superscalar CPUs, that means that certain instruction can be run in parallel - this is called “instruction-level-parallelism”.
The available execution units indicate what instructions can be parallelized - we’ll go over what that means on the next slide.

The prerequisite for instruction-level-parallelism is that there are no dependencies between instructions.

That means that something like
A = B+C
D = A+B

Can’t be parallelized because the 2nd expression depends on the outcome of the 1st expressions

Optimization 1: Better Execution Unit Utilization

Intel® 64 and IA-32 Architectures Optimization Reference Manual Chapter 2.3.1.2

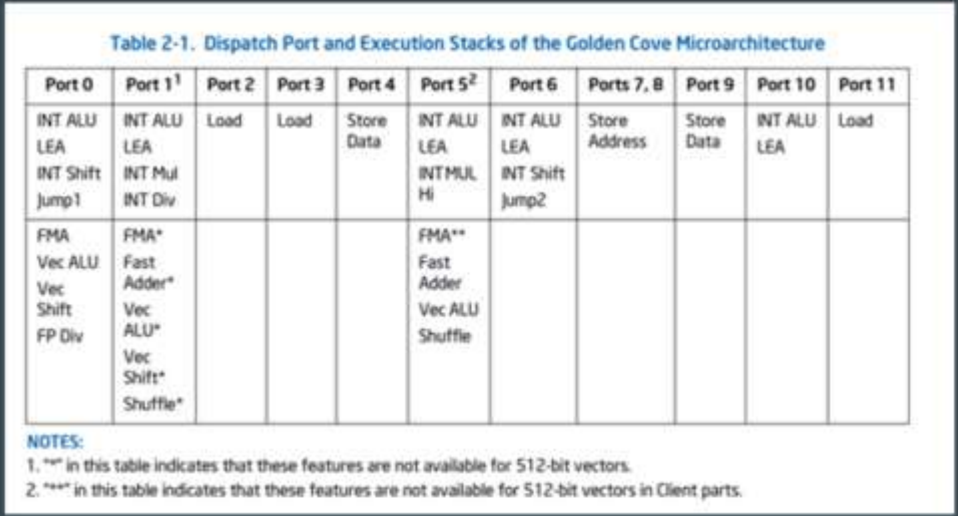


Table 2-1. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

NOTES:
1. "*" in this table indicates that these features are not available for 512-bit vectors.
2. "**" in this table indicates that these features are not available for 512-bit vectors in Client parts.

If we take our handy documentation we can see that there's this list, which gives you an idea of what instructions can be parallelized.

Optimization 1: Better Execution Unit Utilization

Intel® 64 and IA-32 Architectures Optimization Reference Manual Chapter 2.3.1.2

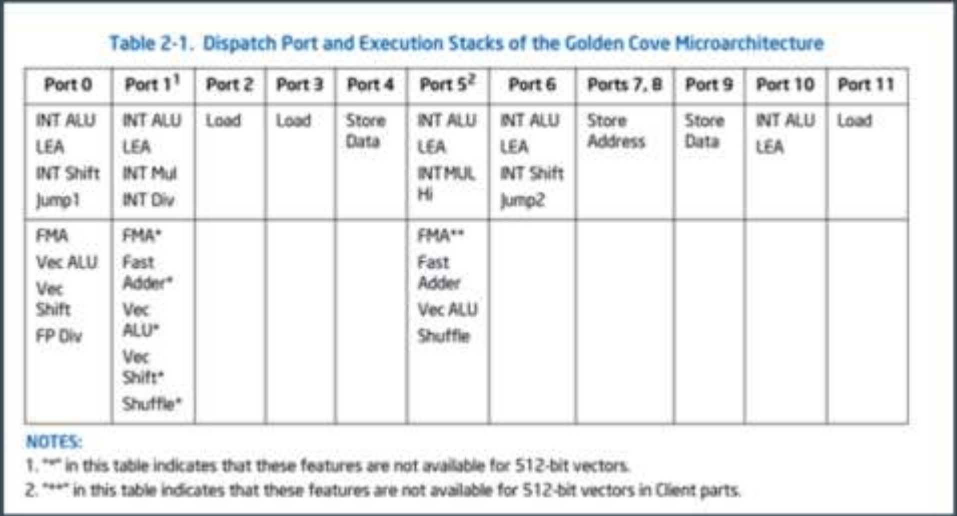


Table 2-1. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

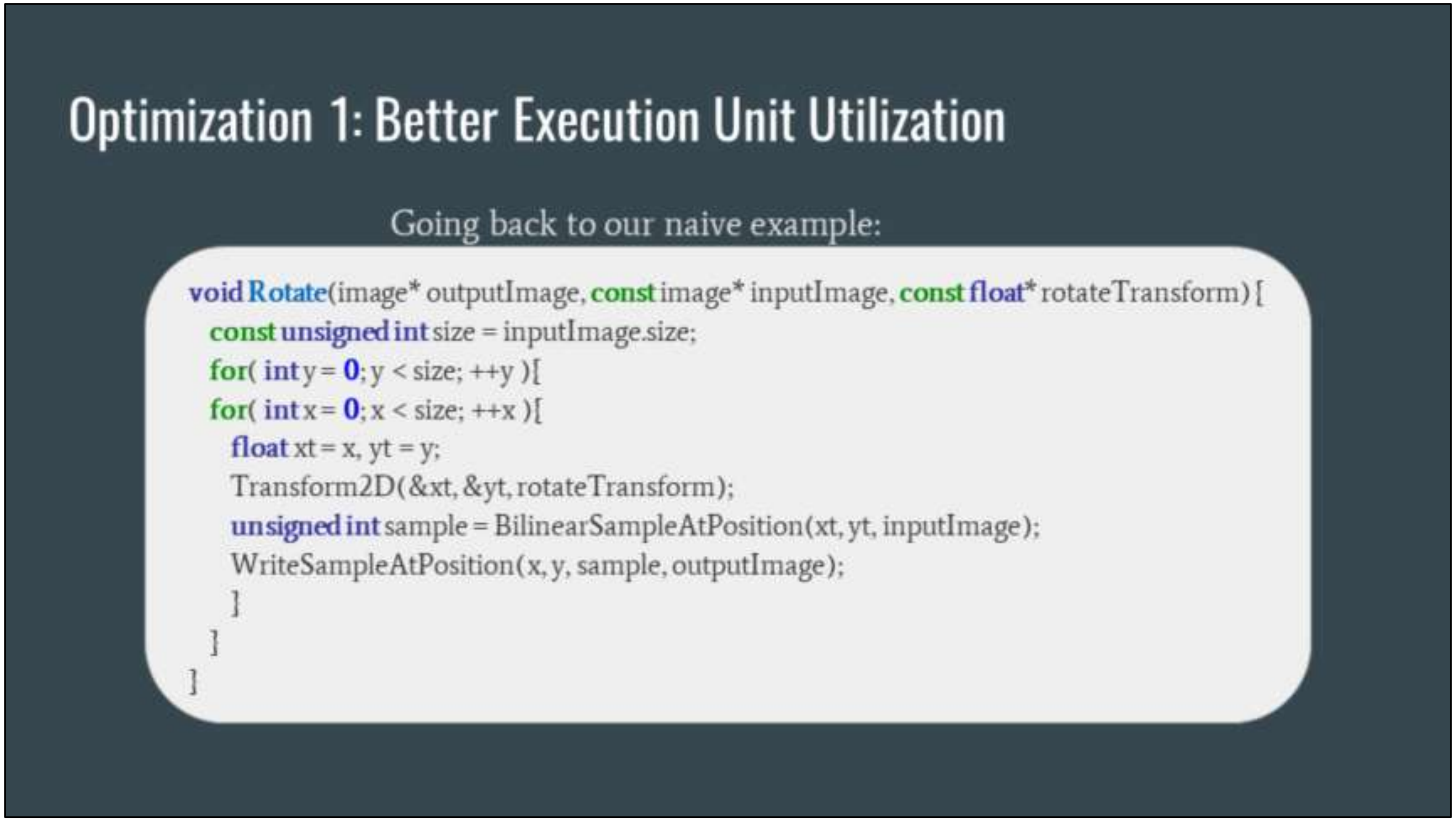
Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INTMUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. "*" in this table indicates that these features are not available for 512-bit vectors.
2. "**" in this table indicates that these features are not available for 512-bit vectors in Client parts.

Gives you an idea of what instructions can be parallelized

If we take our handy documentation we can see that there's this list, which gives you an idea of what instructions can be parallelized.



Optimization 1: Better Execution Unit Utilization

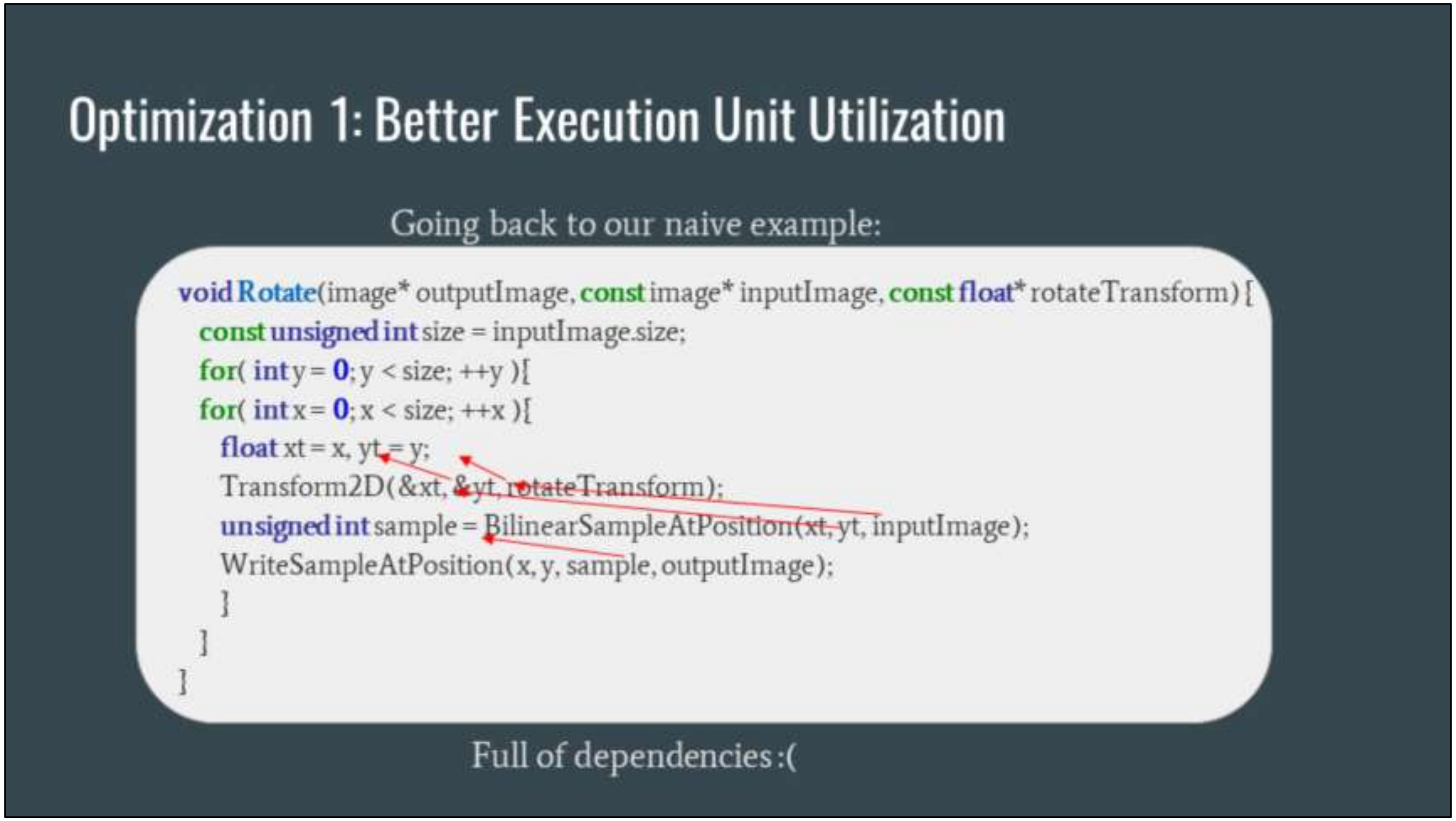
Going back to our naive example:

```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform) {
  const unsigned int size = inputImage.size;
  for( int y = 0; y < size; ++y ){
  for( int x = 0; x < size; ++x ){
    float xt = x, yt = y;
    Transform2D(&xt, &yt, rotateTransform);
    unsigned int sample = BilinearSampleAtPosition(xt, yt, inputImage);
    WriteSampleAtPosition(x, y, sample, outputImage);
    }
  }
}
```

So let’s go back to our naive example.

Unfortunately we have several dependencies here.
BilinearSampleAtPosition depends on the output of Transform2D and WriteSampleAtPosition depends on the output of BilinearSampleAtPosition.

As seen before, this will result in sub-optimal instruction-level-parallelism.



Optimization 1: Better Execution Unit Utilization

Going back to our naive example:

```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform) {
  const unsigned int size = inputImage.size;
  for( int y = 0; y < size; ++y ){
  for( int x = 0; x < size; ++x ){
    float xt = x, yt = y;
    Transform2D(&xt, &yt, rotateTransform);
    unsigned int sample = BilinearSampleAtPosition(xt, yt, inputImage);
    WriteSampleAtPosition(x, y, sample, outputImage);
    }
  }
}
```

Full of dependencies :(

So let's go back to our naive example.

Unfortunately we have several dependencies here.
BilinearSampleAtPosition depends on the output of Transform2D and WriteSampleAtPosition depends on the output of BilinearSampleAtPosition.

As seen before, this will result in sub-optimal instruction-level-parallelism.

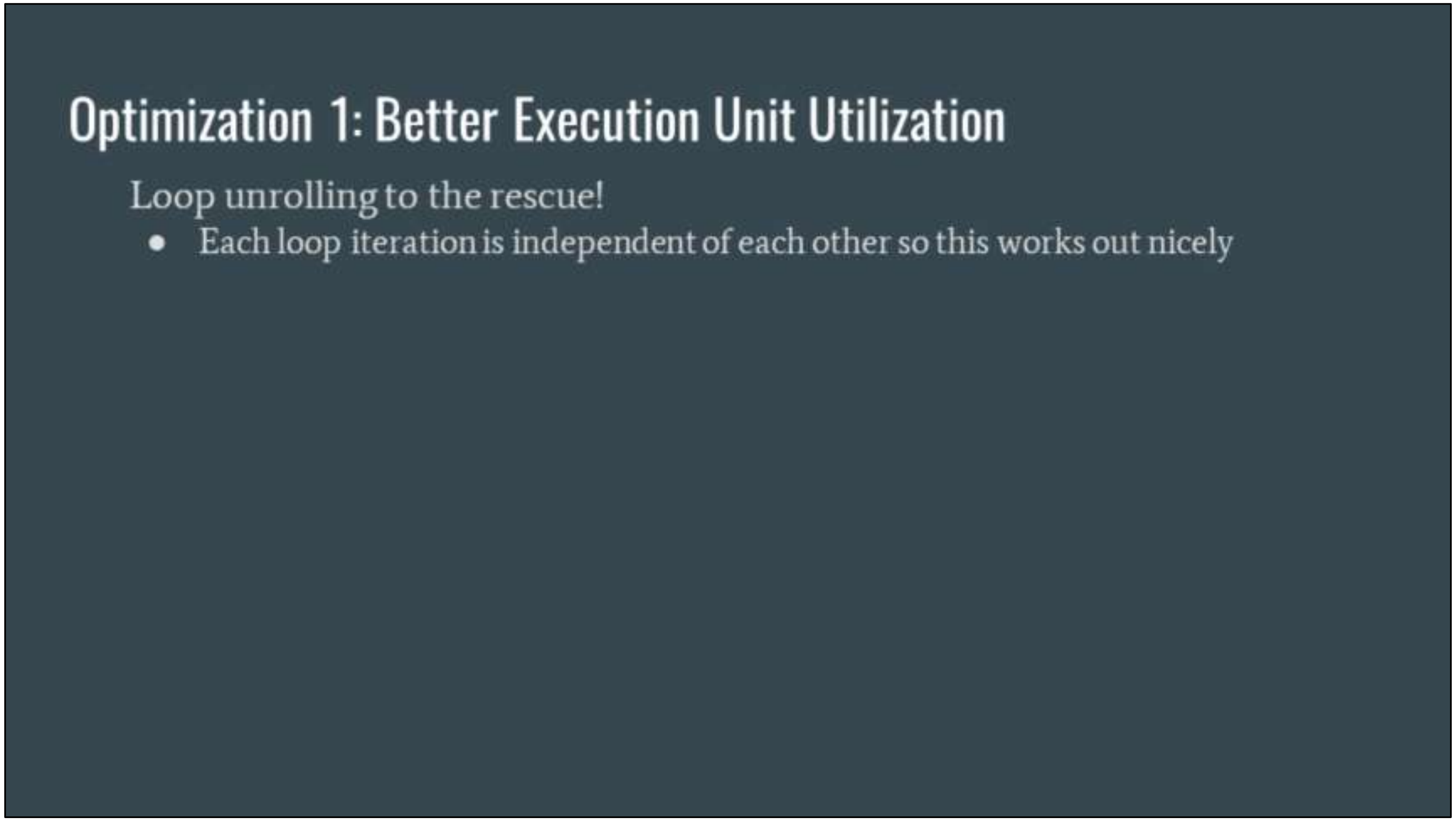
Optimization 1: Better Execution Unit Utilization

Loop unrolling to the rescue!

What we could do however is execute multiple elements per loop iteration since each loop iteration is independent.

This is called loop unrolling.
This is what the naive code would look like with 4x loop unrolling applied.

Note that we also have to change the Transform2D, BilinearSampleAtPosition & WriteSampleAtPosition functions to work on 4 elements.



Optimization 1: Better Execution Unit Utilization

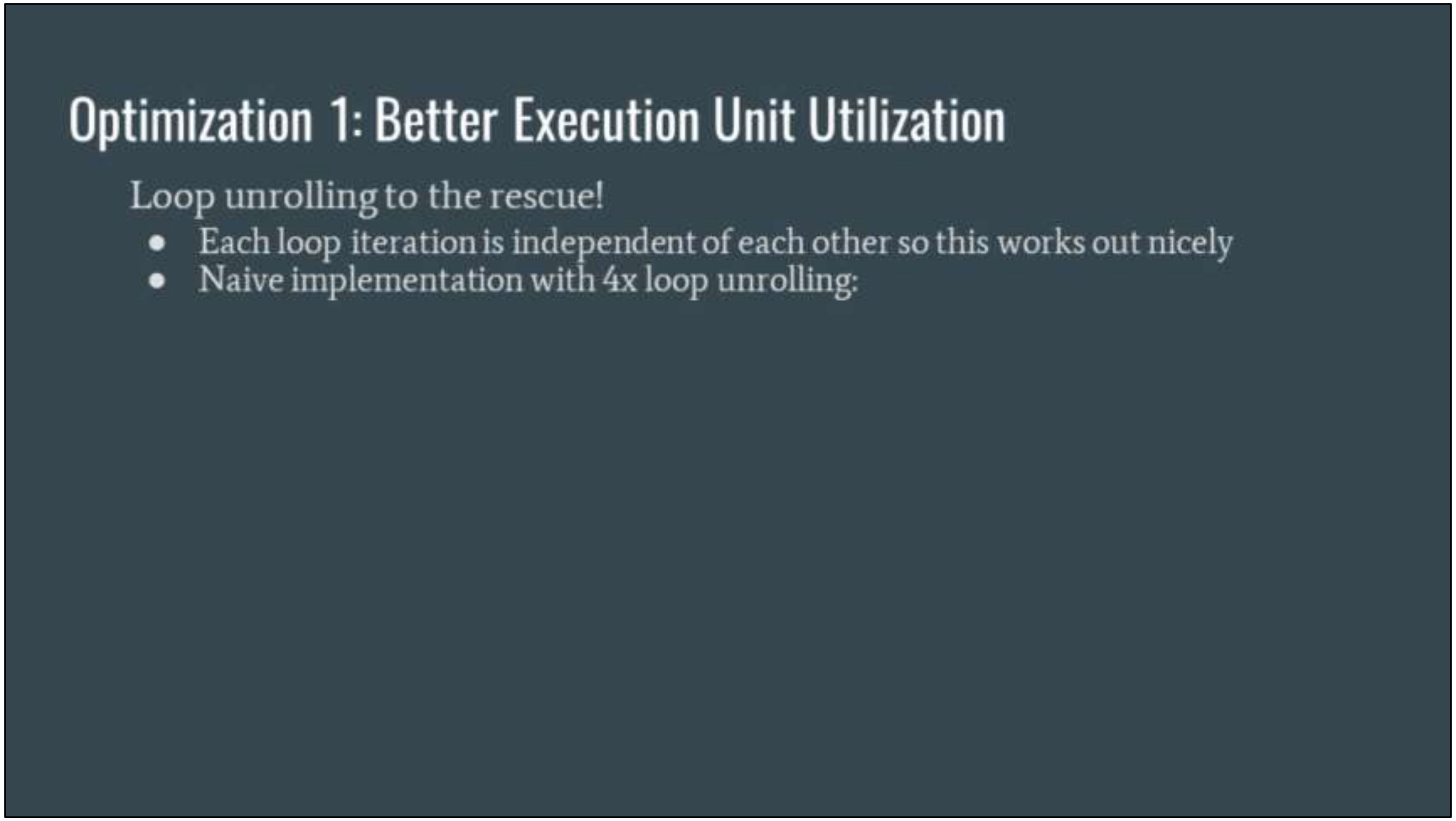
Loop unrolling to the rescue!

- Each loop iteration is independent of each other so this works out nicely

What we could do however is execute multiple elements per loop iteration since each loop iteration is independent.

This is called loop unrolling.
This is what the naive code would look like with 4x loop unrolling applied.

Note that we also have to change the Transform2D, BilinearSampleAtPosition & WriteSampleAtPosition functions to work on 4 elements.



Optimization 1: Better Execution Unit Utilization

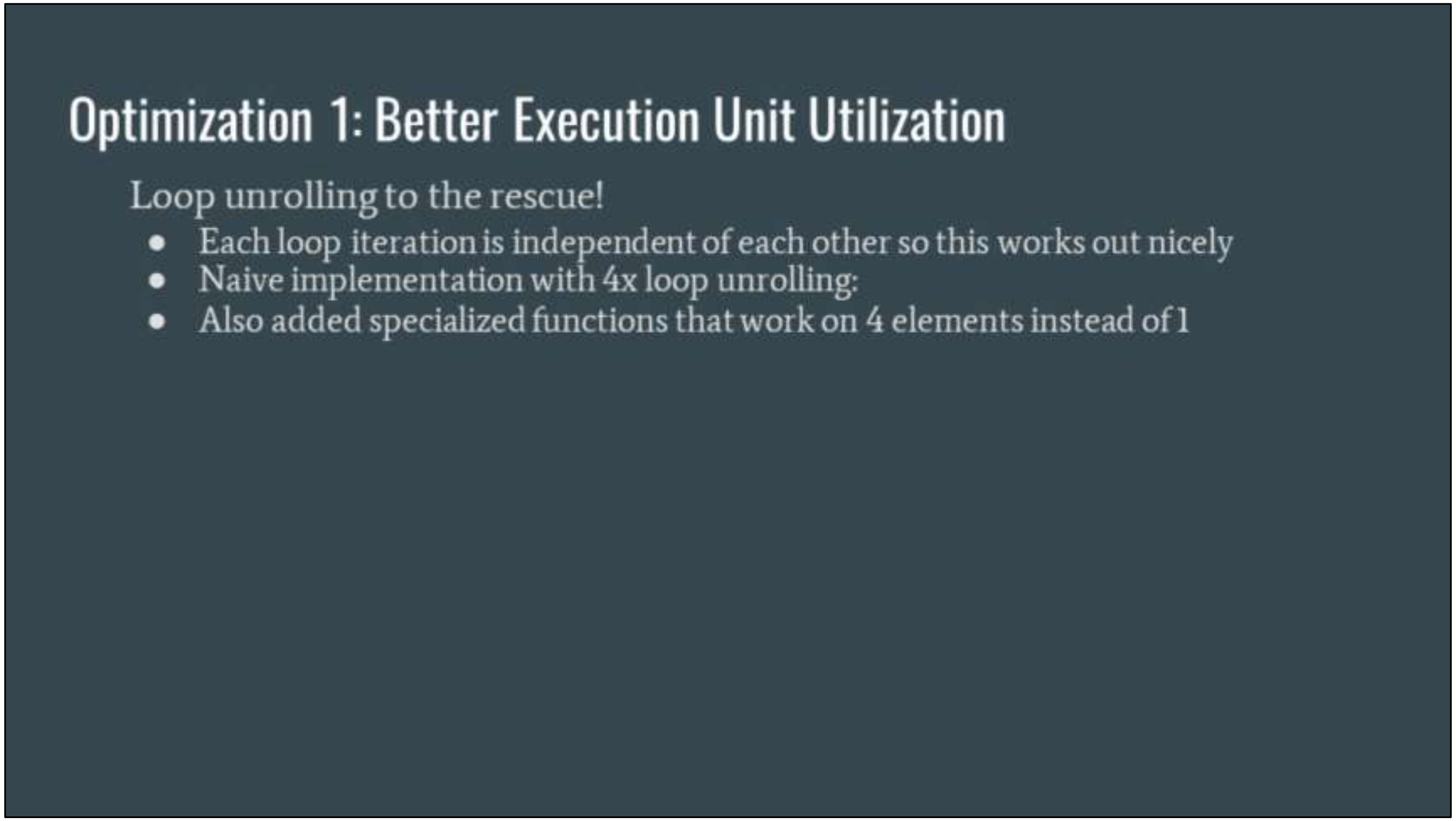
Loop unrolling to the rescue!

- Each loop iteration is independent of each other so this works out nicely
- Naive implementation with 4x loop unrolling:

What we could do however is execute multiple elements per loop iteration since each loop iteration is independent.

This is called loop unrolling.
This is what the naive code would look like with 4x loop unrolling applied.

Note that we also have to change the Transform2D, BilinearSampleAtPosition & WriteSampleAtPosition functions to work on 4 elements.


Optimization 1: Better Execution Unit Utilization

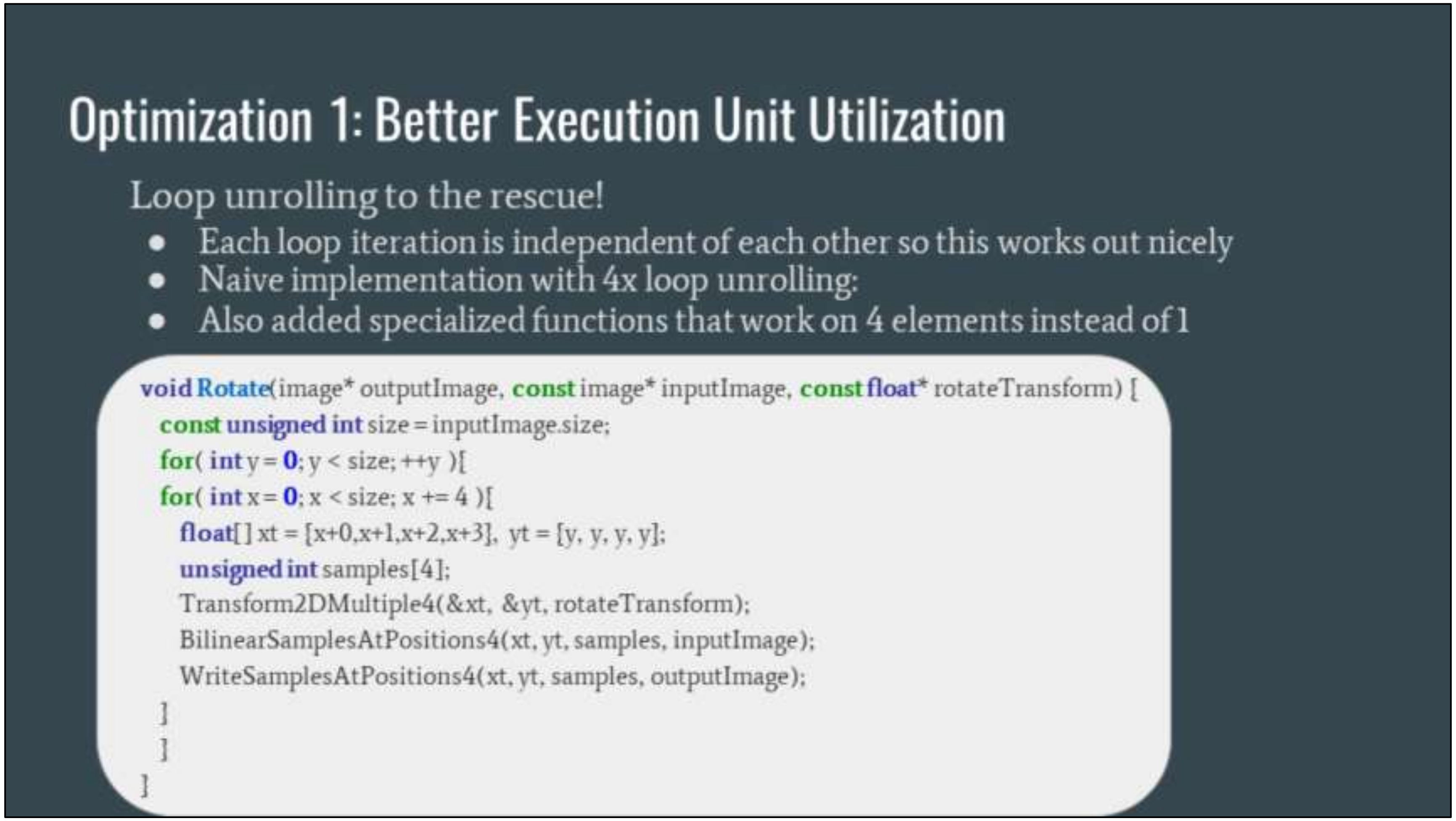
Loop unrolling to the rescue!

- Each loop iteration is independent of each other so this works out nicely
- Naive implementation with 4x loop unrolling:
- Also added specialized functions that work on 4 elements instead of 1

What we could do however is execute multiple elements per loop iteration since each loop iteration is independent.

This is called loop unrolling.
This is what the naive code would look like with 4x loop unrolling applied.

Note that we also have to change the Transform2D, BilinearSampleAtPosition & WriteSampleAtPosition functions to work on 4 elements.



Optimization 1: Better Execution Unit Utilization

Loop unrolling to the rescue!

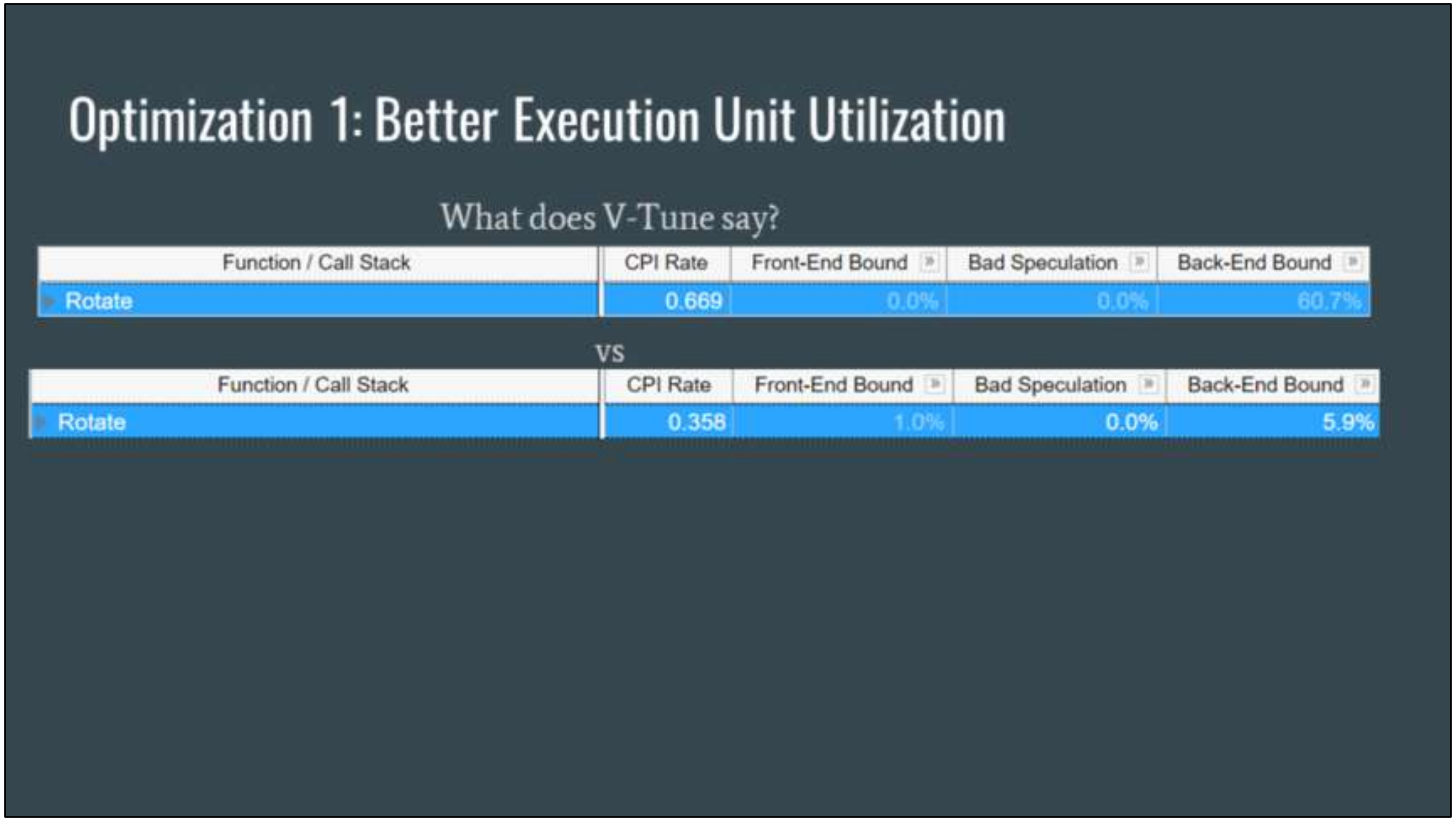
- Each loop iteration is independent of each other so this works out nicely
- Naive implementation with 4x loop unrolling:
- Also added specialized functions that work on 4 elements instead of 1

```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform) [
  const unsigned int size = inputImage.size;
  for( int y = 0; y < size; ++y )[
  for( int x = 0; x < size; x += 4 )[
    float[] xt = [x+0,x+1,x+2,x+3], yt = [y, y, y, y];
    unsigned int samples[4];
    Transform2DMultiple4(&xt, &yt, rotateTransform);
    BilinearSamplesAtPositions4(xt, yt, samples, inputImage);
    WriteSamplesAtPositions4(xt, yt, samples, outputImage);
  ]
  ]
]
```

What we could do however is execute multiple elements per loop iteration since each loop iteration is independent.

This is called loop unrolling.
This is what the naive code would look like with 4x loop unrolling applied.

Note that we also have to change the Transform2D, BilinearSampleAtPosition & WriteSampleAtPosition functions to work on 4 elements.



Optimization 1: Better Execution Unit Utilization

What does V-Tune say?

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

vs

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.358	1.0%	0.0%	5.9%

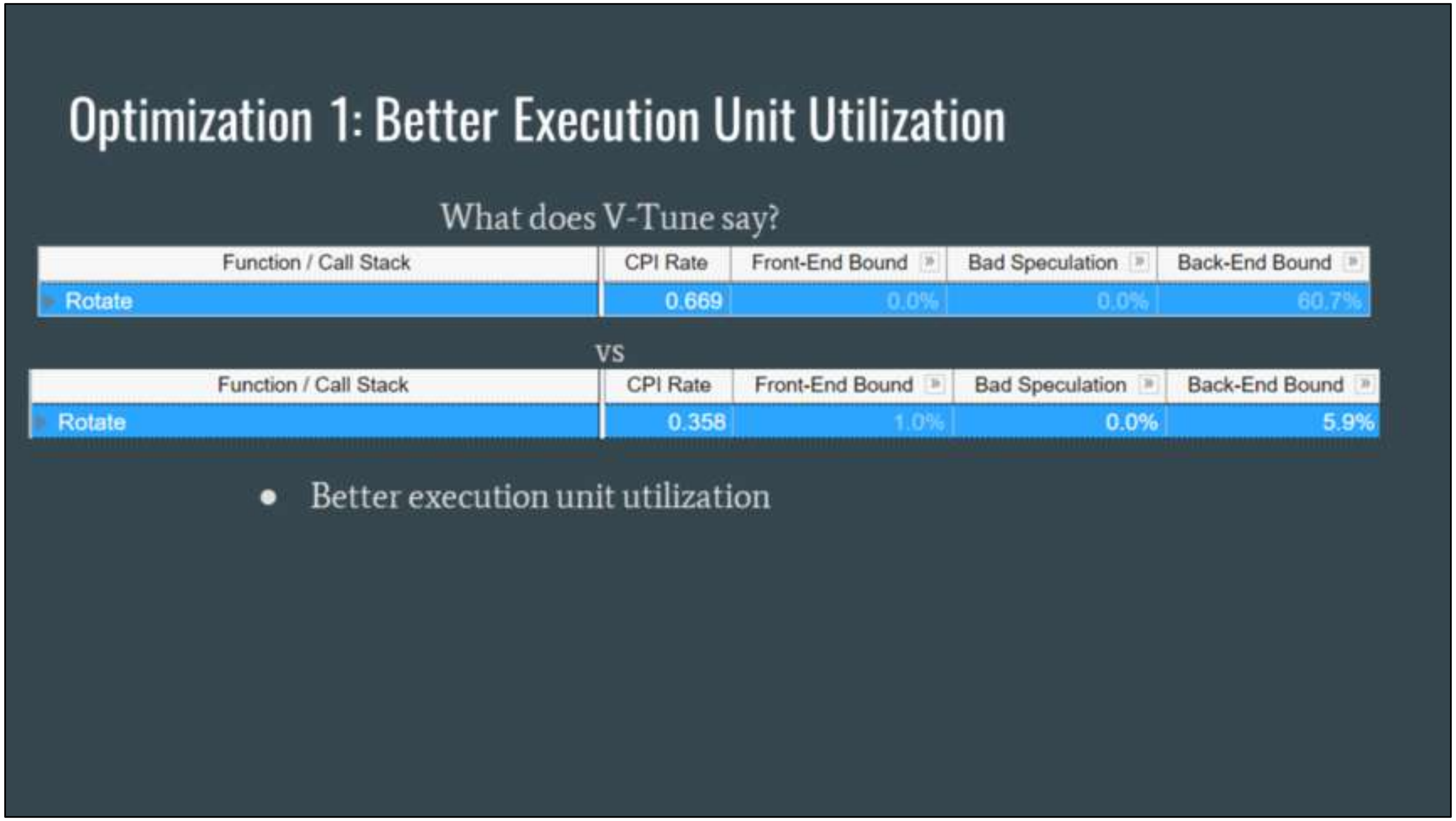
So after this has been added, let’s check the Microarchitecture Exploration analysis again in V-Tune.

This is what we had before and
This is what we have now with the loop unrolling inplace.

As you can see the CPI Rate went down (this is good) and we’re also less Back-End Bound than before.
This is an indicator that this code now is better parallelizable on an instruction level.

The icing on the cake is that this also leaves more room for the compiler to apply optimizations like auto-vectorization, but we’ll go into more detail about that later.
Overall the code is still readable and the changes that we’ve had to do are only minimal.

The reward for our work is about a 50ms improvement over the naive version.



Optimization 1: Better Execution Unit Utilization

What does V-Tune say?

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

vs

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.358	1.0%	0.0%	5.9%

- Better execution unit utilization

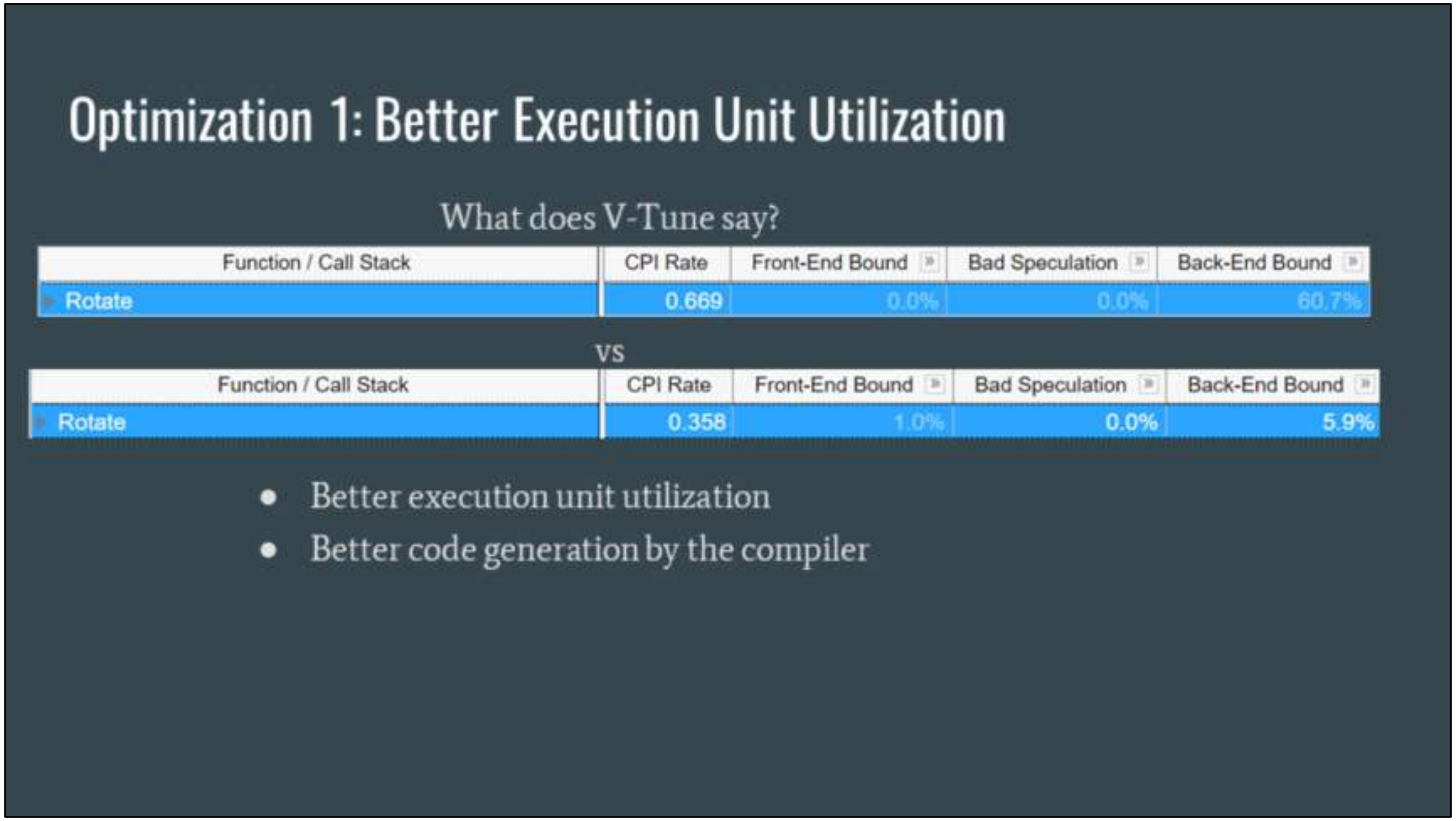
So after this has been added, let's check the Microarchitecture Exploration analysis again in V-Tune.

This is what we had before and
This is what we have now with the loop unrolling inplace.

As you can see the CPI Rate went down (this is good) and we're also less Back-End Bound than before.
This is an indicator that this code now is better parallelizable on an instruction level.

The icing on the cake is that this also leaves more room for the compiler to apply optimizations like auto-vectorization, but we'll go into more detail about that later.
Overall the code is still readable and the changes that we've had to do are only minimal.

The reward for our work is about a 50ms improvement over the naive version.



Optimization 1: Better Execution Unit Utilization

What does V-Tune say?

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

vs

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.358	1.0%	0.0%	5.9%

- Better execution unit utilization
- Better code generation by the compiler

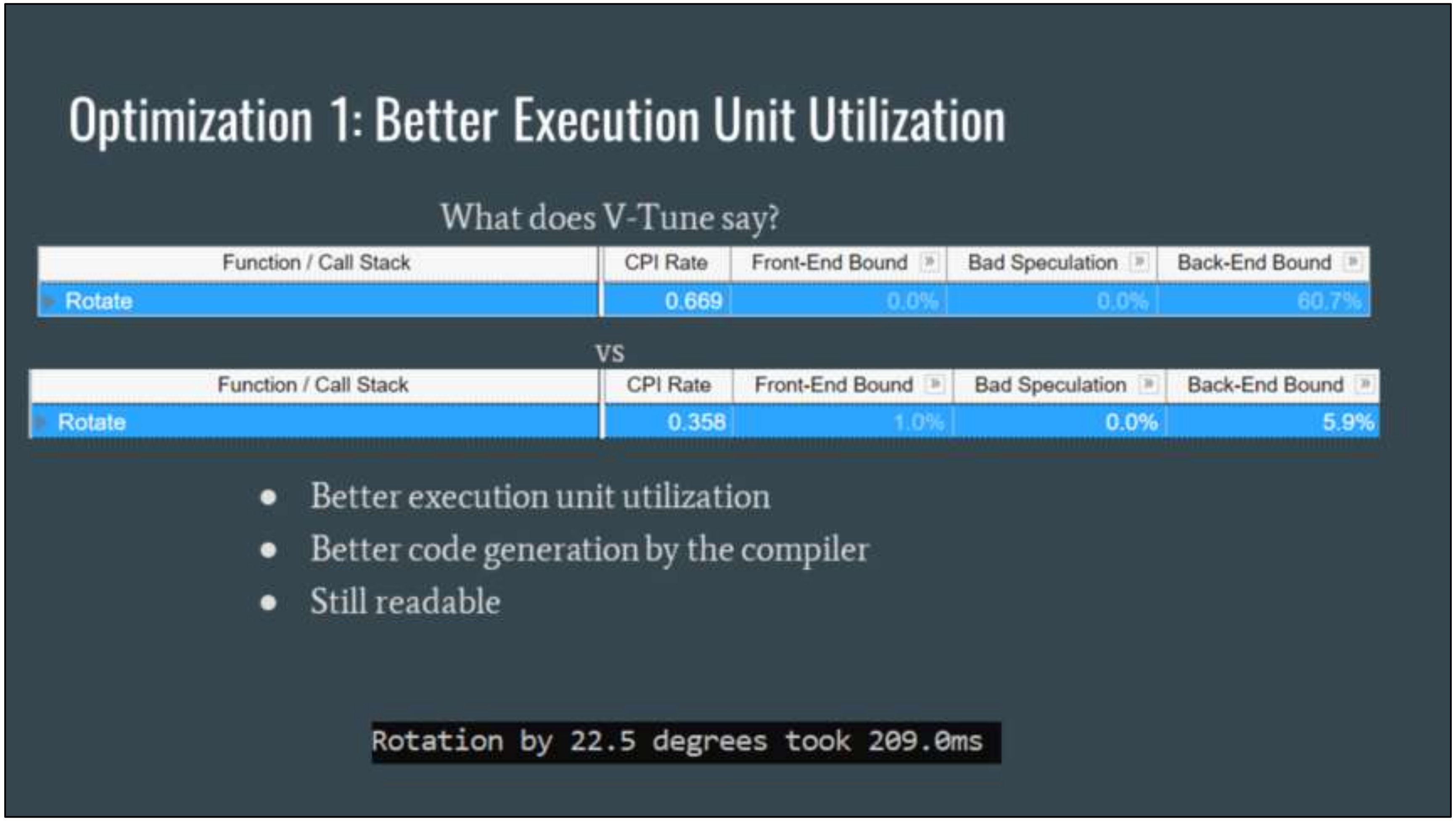
So after this has been added, let's check the Microarchitecture Exploration analysis again in V-Tune.

This is what we had before and
This is what we have now with the loop unrolling inplace.

As you can see the CPI Rate went down (this is good) and we're also less Back-End Bound than before.
This is an indicator that this code now is better parallelizable on an instruction level.

The icing on the cake is that this also leaves more room for the compiler to apply optimizations like auto-vectorization, but we'll go into more detail about that later.
Overall the code is still readable and the changes that we've had to do are only minimal.

The reward for our work is about a 50ms improvement over the naive version.



Optimization 1: Better Execution Unit Utilization

What does V-Tune say?

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

vs

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.358	1.0%	0.0%	5.9%

- Better execution unit utilization
- Better code generation by the compiler
- Still readable

```
Rotation by 22.5 degrees took 209.0ms
```

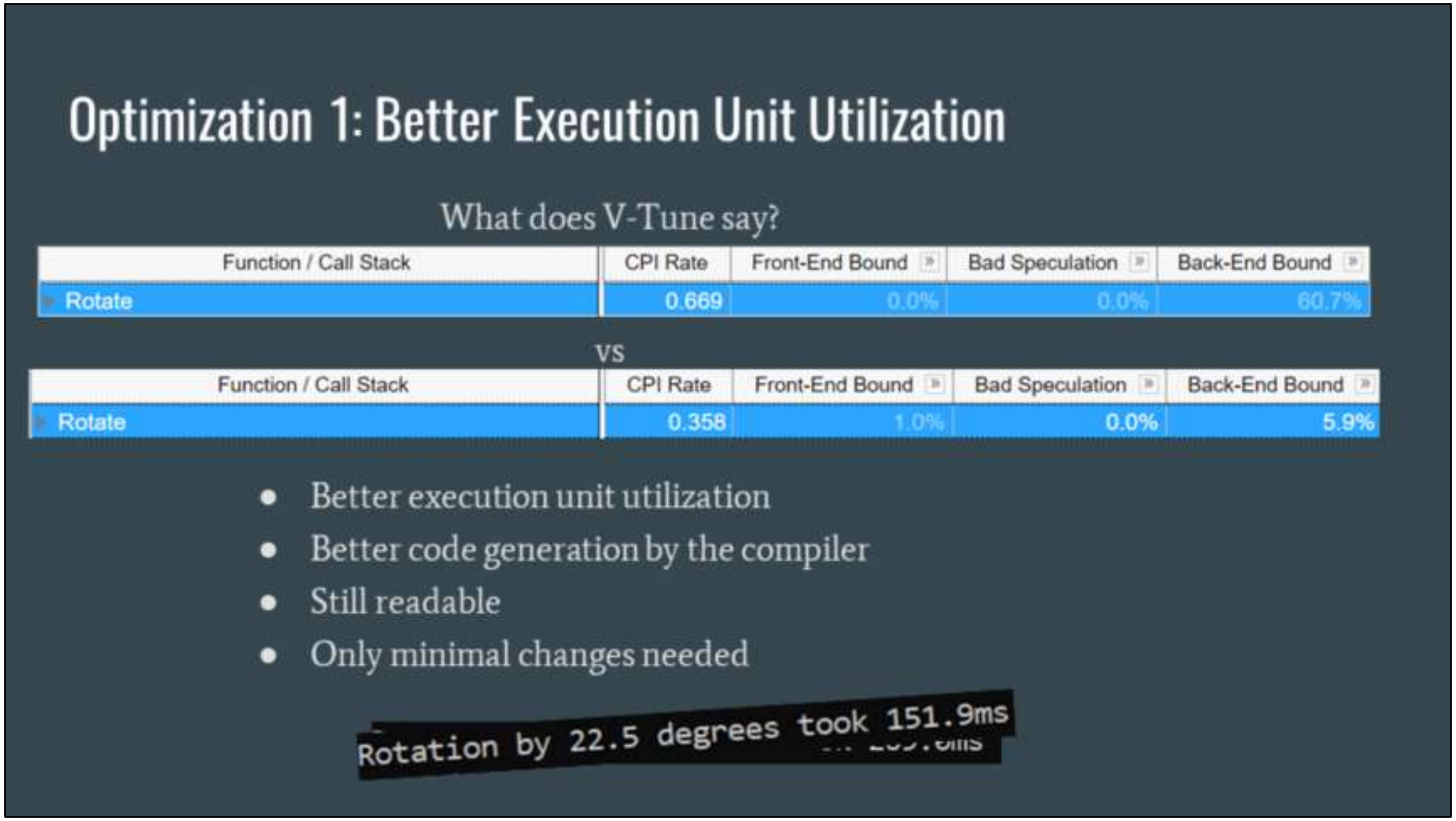
So after this has been added, let’s check the Microarchitecture Exploration analysis again in V-Tune.

This is what we had before and
This is what we have now with the loop unrolling inplace.

As you can see the CPI Rate went down (this is good) and we’re also less Back-End Bound than before.
This is an indicator that this code now is better parallelizable on an instruction level.

The icing on the cake is that this also leaves more room for the compiler to apply optimizations like auto-vectorization, but we’ll go into more detail about that later.
Overall the code is still readable and the changes that we’ve had to do are only minimal.

The reward for our work is about a 50ms improvement over the naive version.



Optimization 1: Better Execution Unit Utilization

What does V-Tune say?

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.669	0.0%	0.0%	60.7%

vs

Function / Call Stack	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound
Rotate	0.358	1.0%	0.0%	5.9%

- Better execution unit utilization
- Better code generation by the compiler
- Still readable
- Only minimal changes needed

```
Rotation by 22.5 degrees took 151.9ms
```

So after this has been added, let's check the Microarchitecture Exploration analysis again in V-Tune.

This is what we had before and
This is what we have now with the loop unrolling inplace.

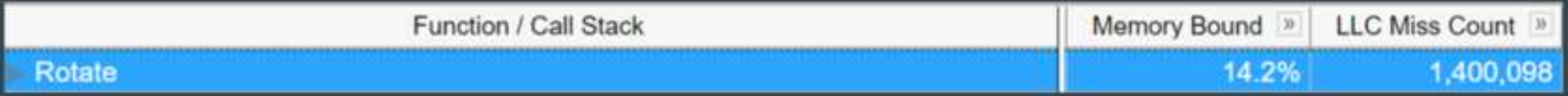
As you can see the CPI Rate went down (this is good) and we're also less Back-End Bound than before.
This is an indicator that this code now is better parallelizable on an instruction level.

The icing on the cake is that this also leaves more room for the compiler to apply optimizations like auto-vectorization, but we'll go into more detail about that later.
Overall the code is still readable and the changes that we've had to do are only minimal.

The reward for our work is about a 50ms improvement over the naive version.

Optimization 2: Loop Blocking

Lets look at memory accesses and cache utilization using V-Tune Memory Access Analysis



Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

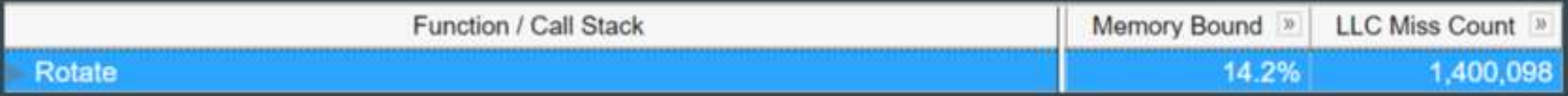
Ouch

Let’s check the memory access performance analysis of our program.
For that there’s the “Memory Access” Analysis in V-Tune.

Running this for our function shows us that we’re somewhat memory bound and have lots of cache misses.
Question to audience: “What could be the reason?”

Optimization 2: Loop Blocking

Lets look at memory accesses and cache utilization using V-Tune Memory Access Analysis



Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

Ouch

What could be the reason?

Let’s check the memory access performance analysis of our program.
For that there’s the “Memory Access” Analysis in V-Tune.

Running this for our function shows us that we’re somewhat memory bound and have lots of cache misses.
Question to audience: “What could be the reason?”

Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):

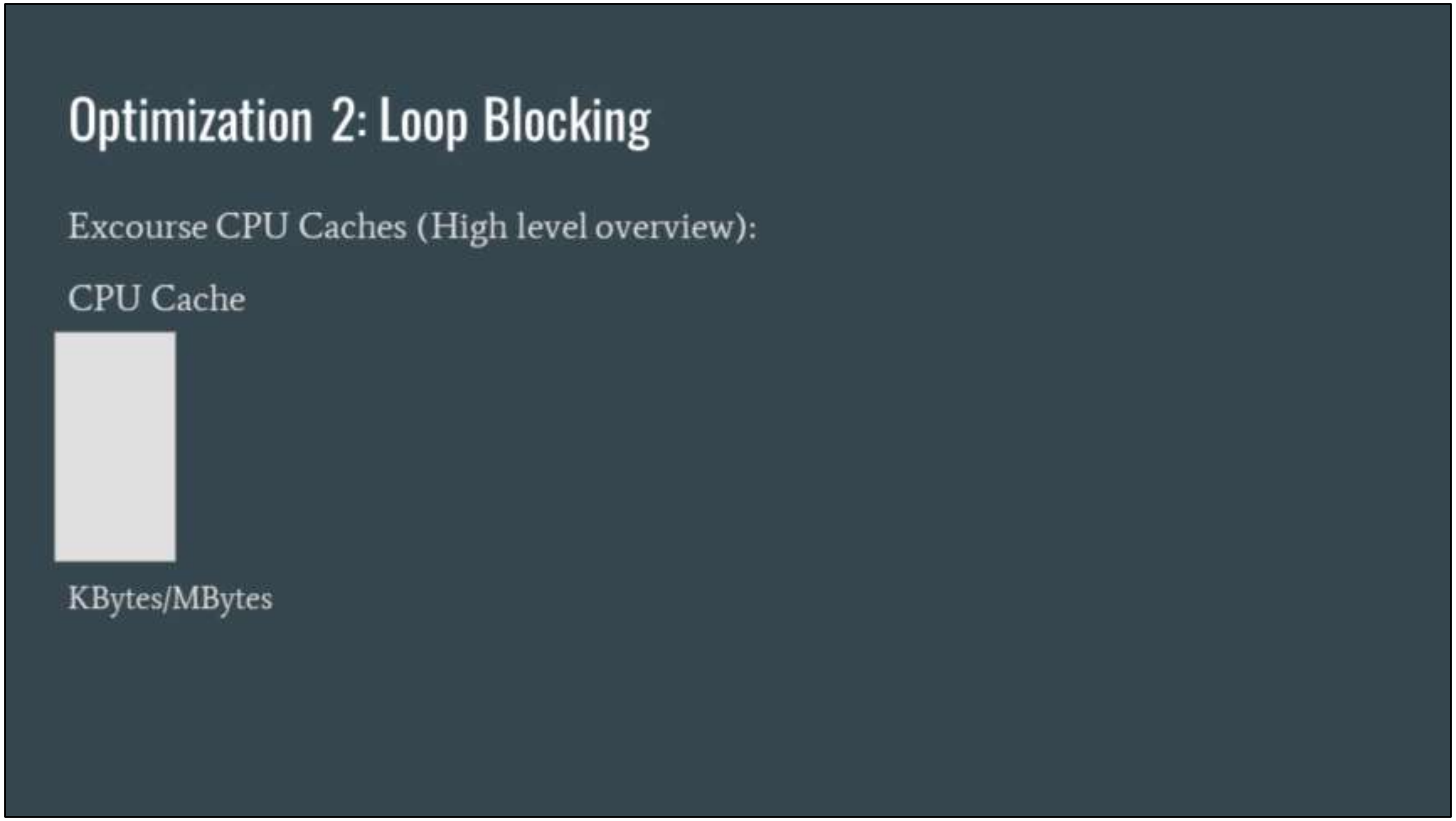
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):

CPU Cache

KBytes/MBytes

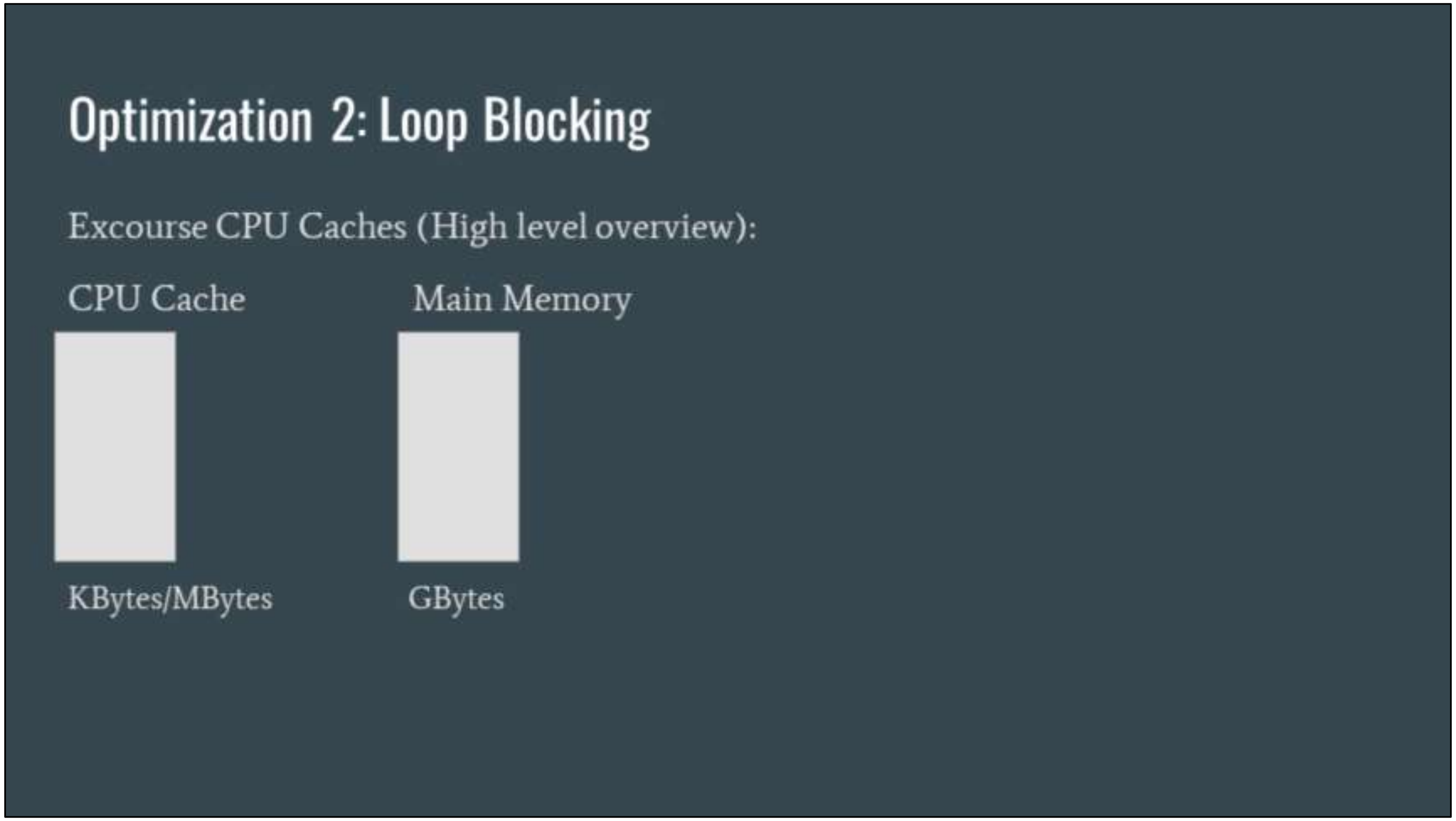
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

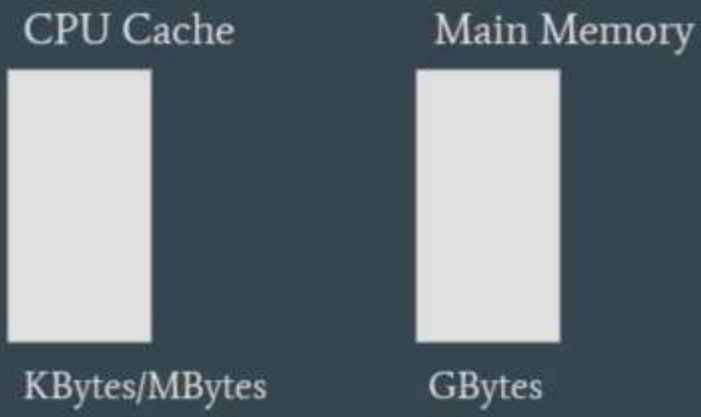
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



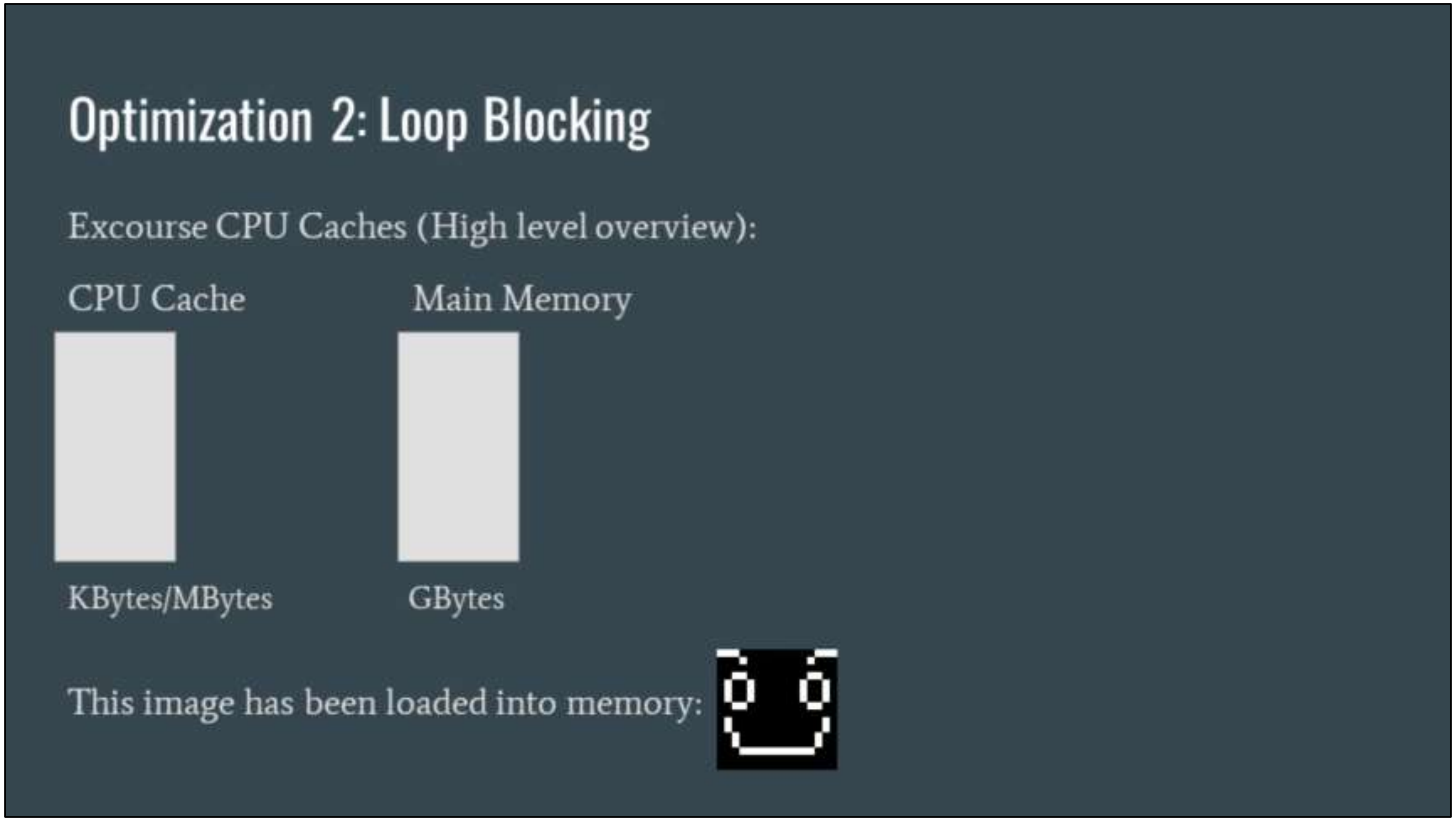
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

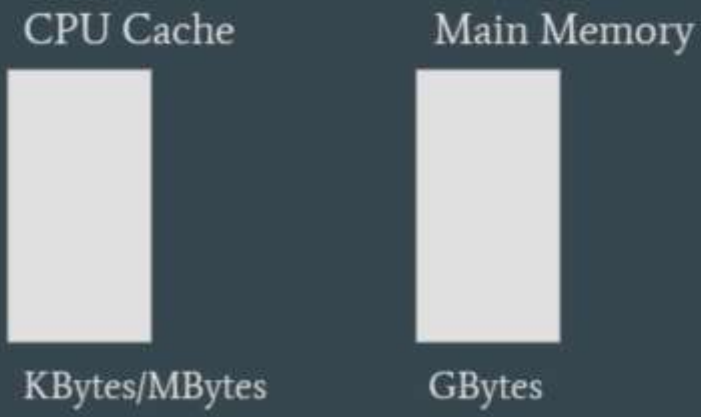
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



This image has been loaded into memory:

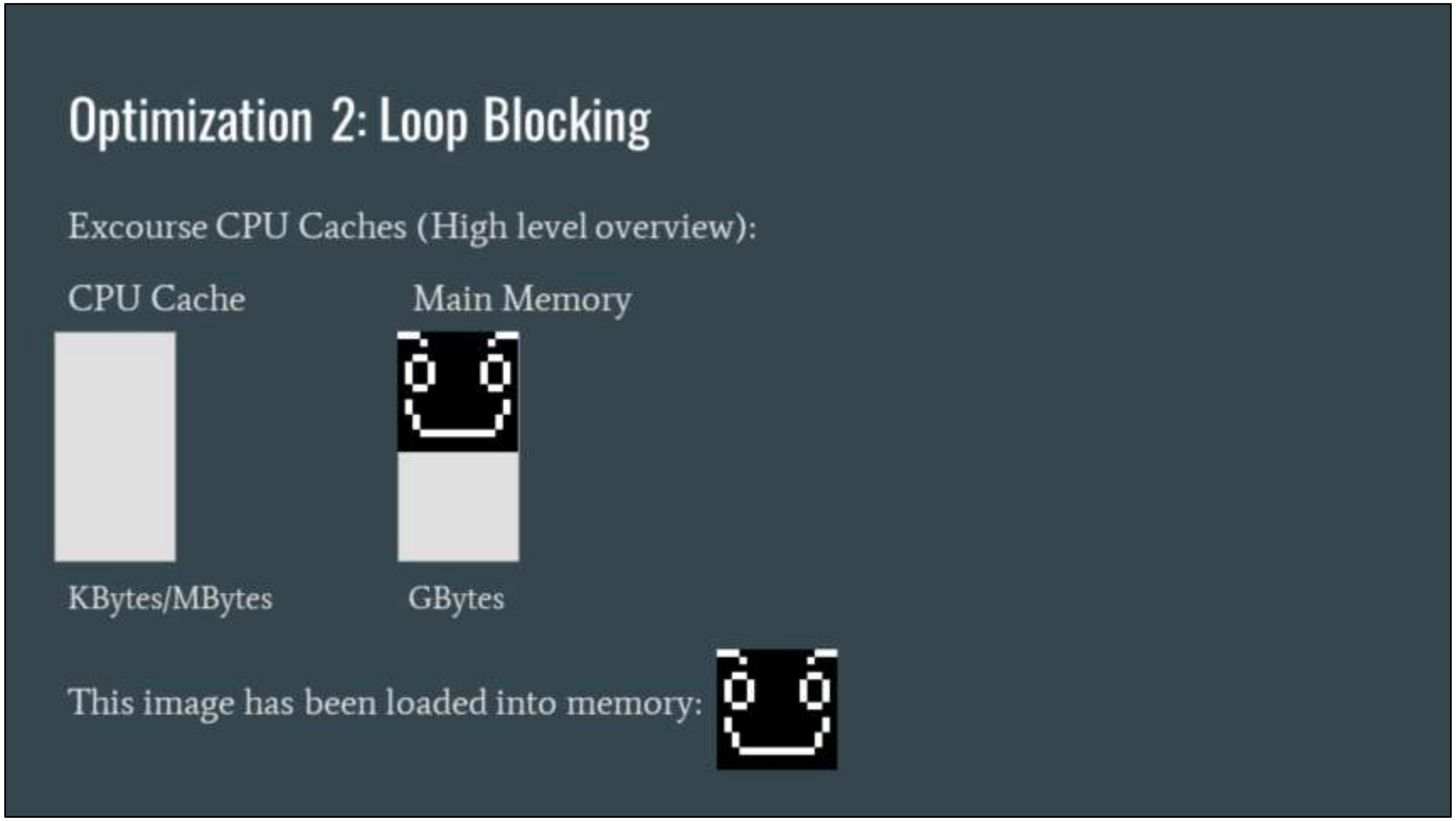
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

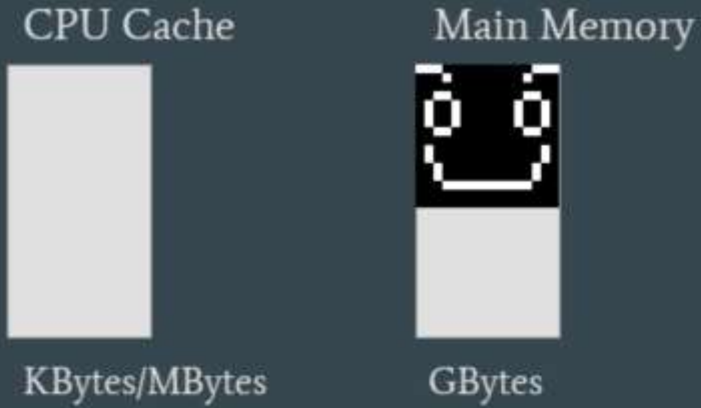
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



This image has been loaded into memory:

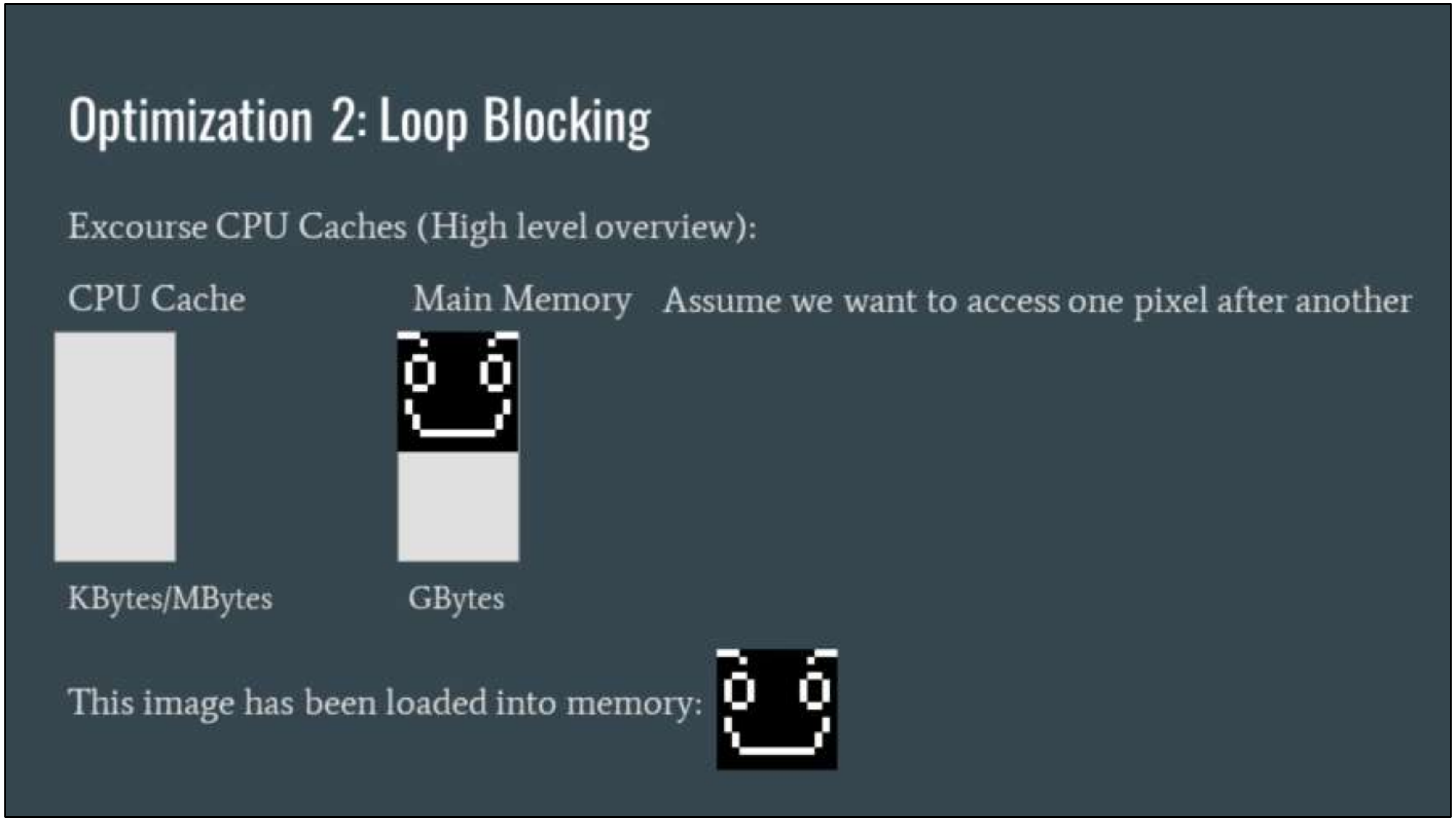
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):

CPU Cache

Main Memory

Assume we want to access one pixel after another

KBytes/MBytes

GBytes

This image has been loaded into memory:

To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

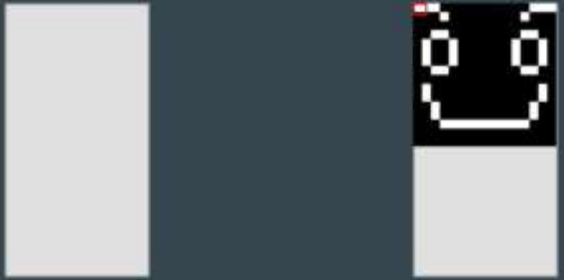
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.

Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):

CPU Cache   Main Memory   Assume we want to access one pixel after another



This image has been loaded into memory:

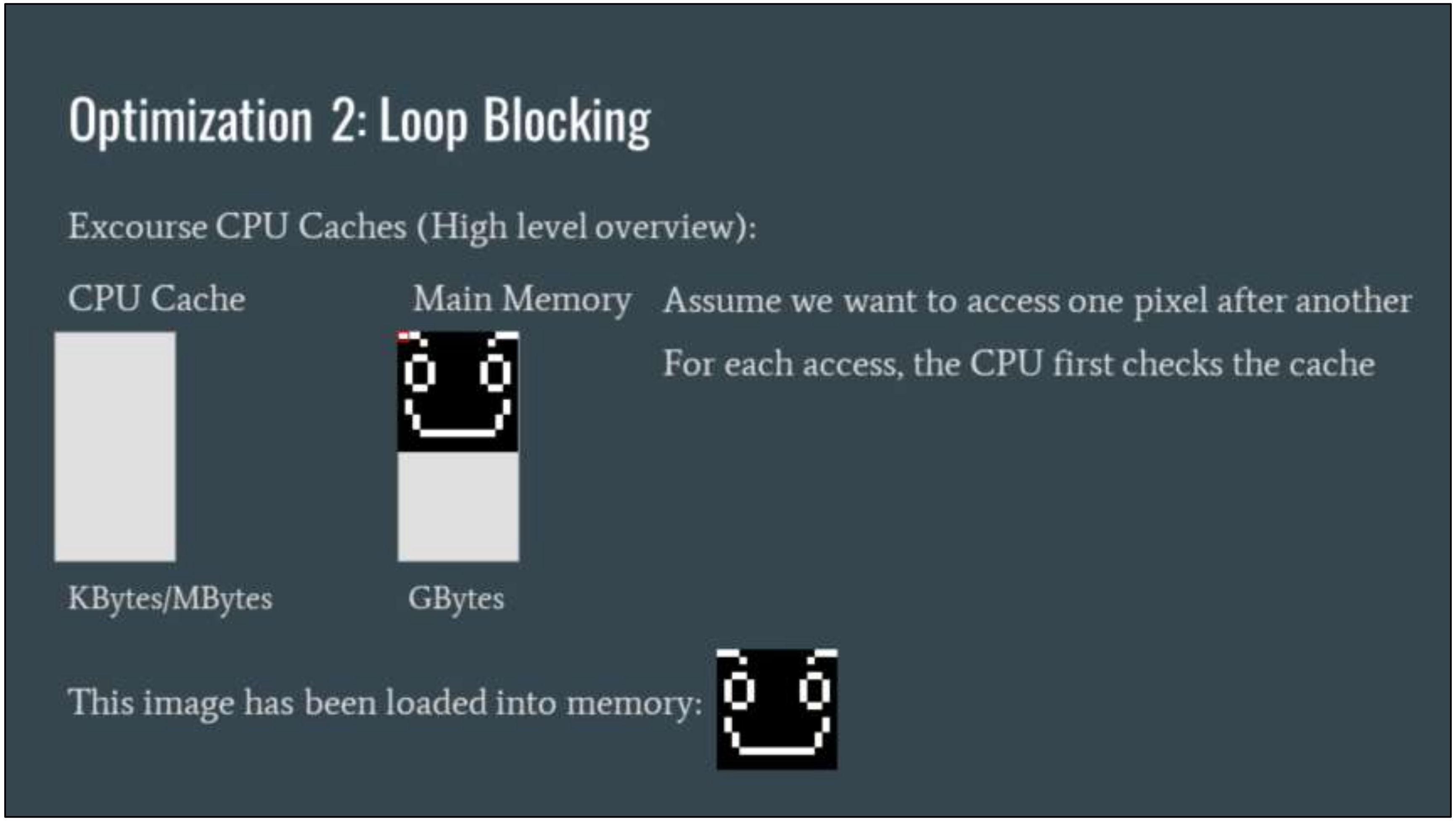
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

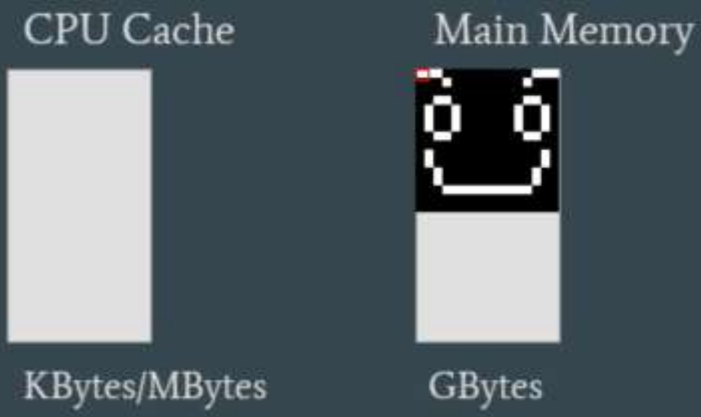
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



Assume we want to access one pixel after another

For each access, the CPU first checks the cache

This image has been loaded into memory:

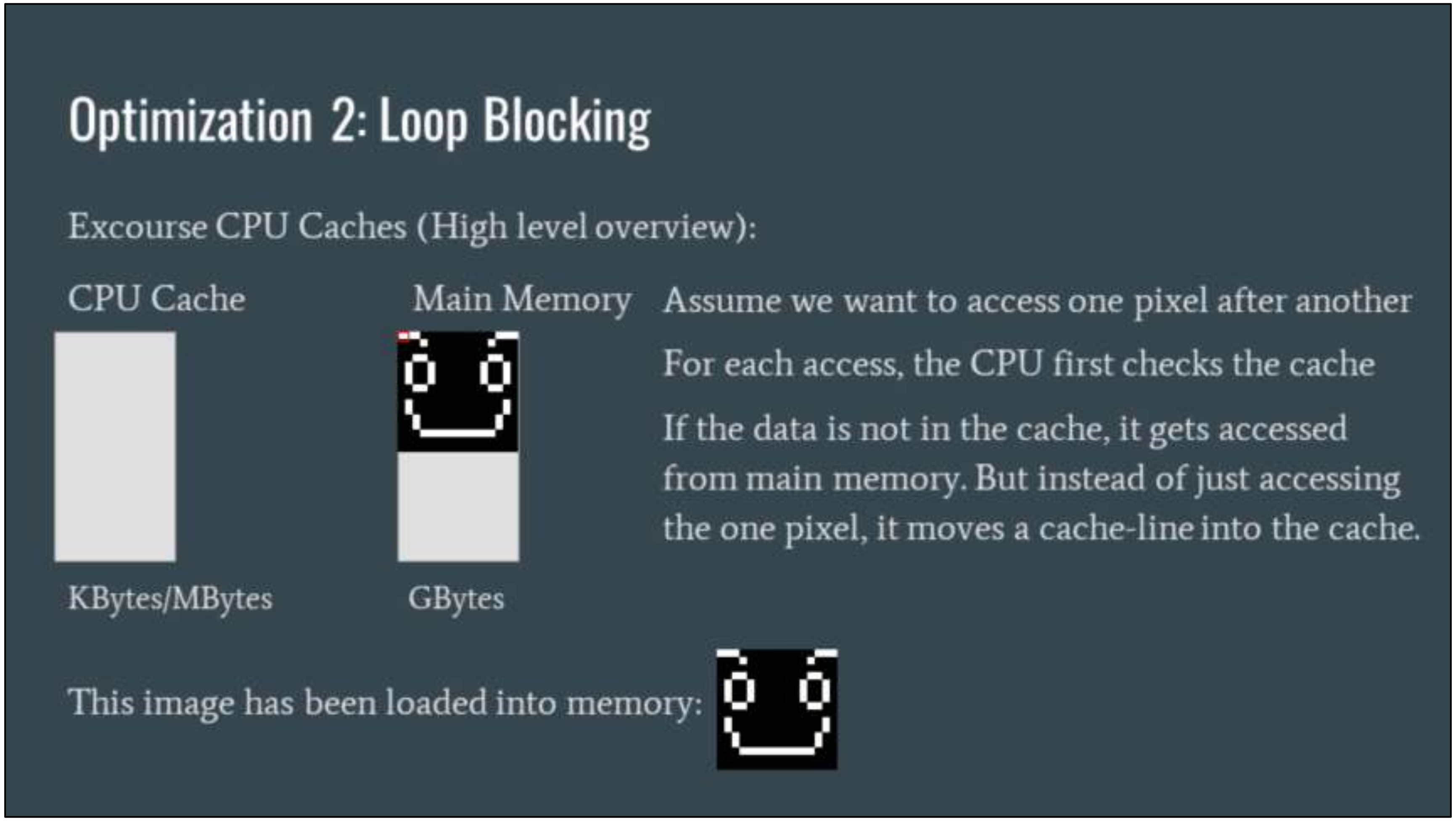
To get into what the reason for these performance problems are, we’ll have to do a quick excourse so that we’re all onboard regarding cpu caches.
I’ll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let’s assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

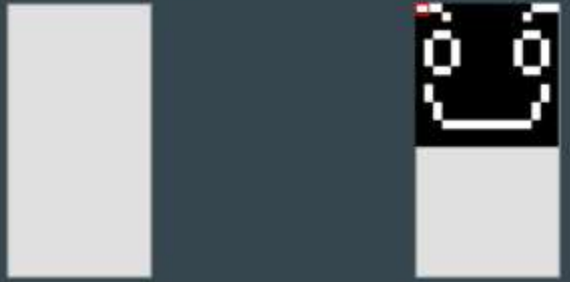
In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won’t only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



Assume we want to access one pixel after another

For each access, the CPU first checks the cache

If the data is not in the cache, it gets accessed from main memory. But instead of just accessing the one pixel, it moves a cache-line into the cache.

This image has been loaded into memory:

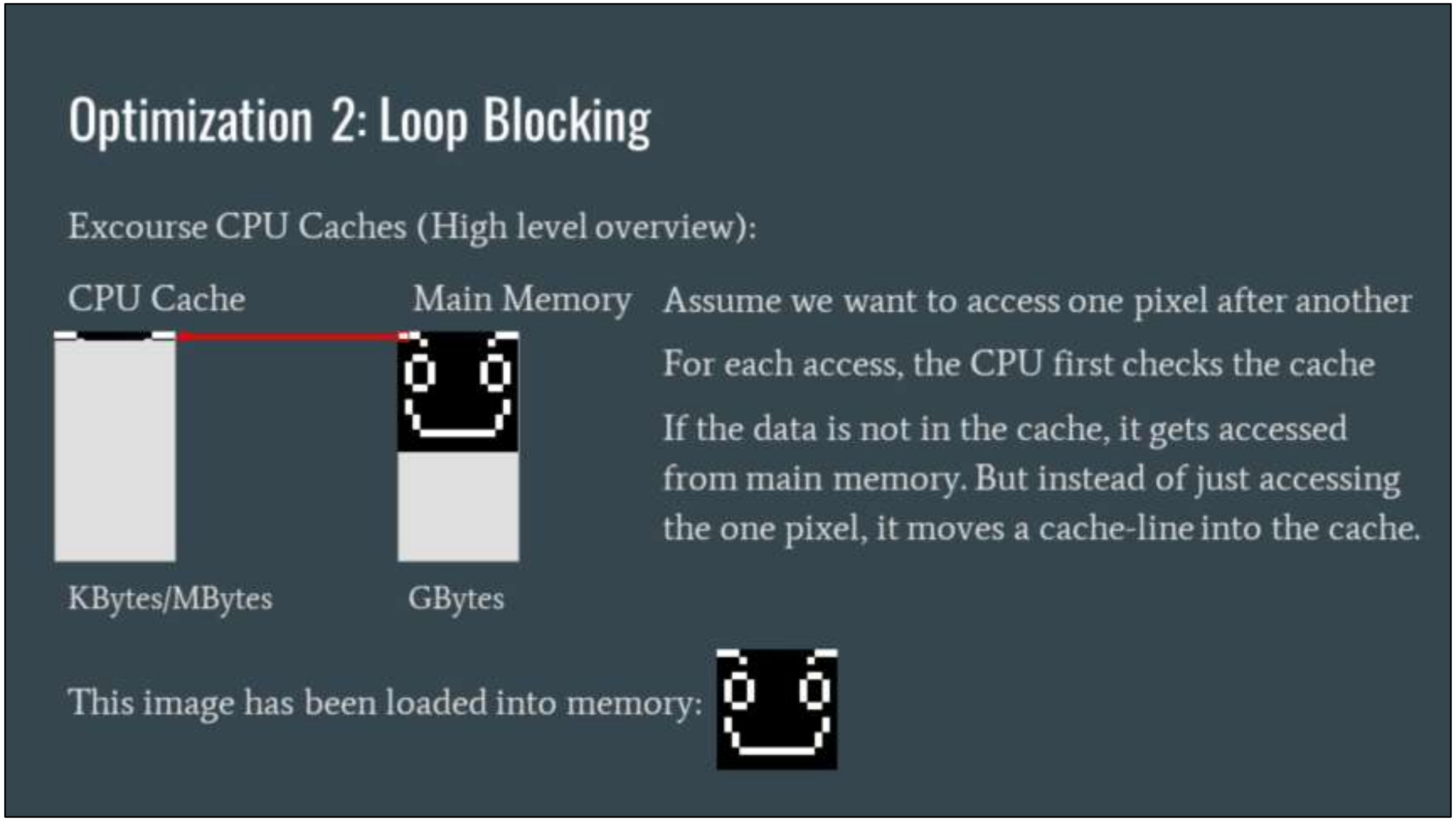
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):

Assume we want to access one pixel after another

For each access, the CPU first checks the cache

If the data is not in the cache, it gets accessed from main memory. But instead of just accessing the one pixel, it moves a cache-line into the cache.

This image has been loaded into memory:

To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

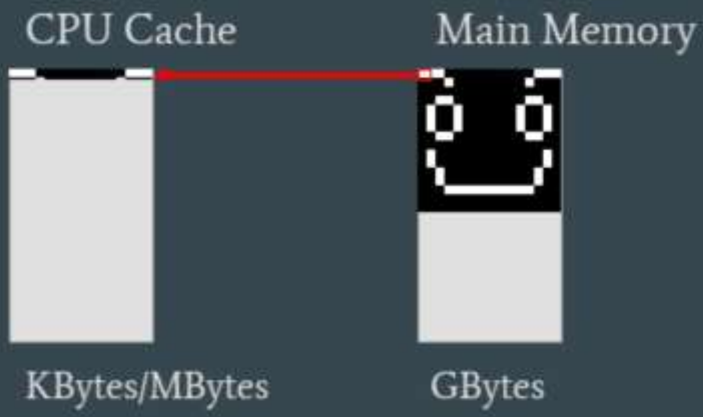
If the data is in one of the caches - great.

In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.

Optimization 2: Loop Blocking

Excourse CPU Caches (High level overview):



Assume we want to access one pixel after another

For each access, the CPU first checks the cache

If the data is not in the cache, it gets accessed from main memory. But instead of just accessing the one pixel, it moves a cache-line into the cache.

This is known as a cache miss

This image has been loaded into memory:

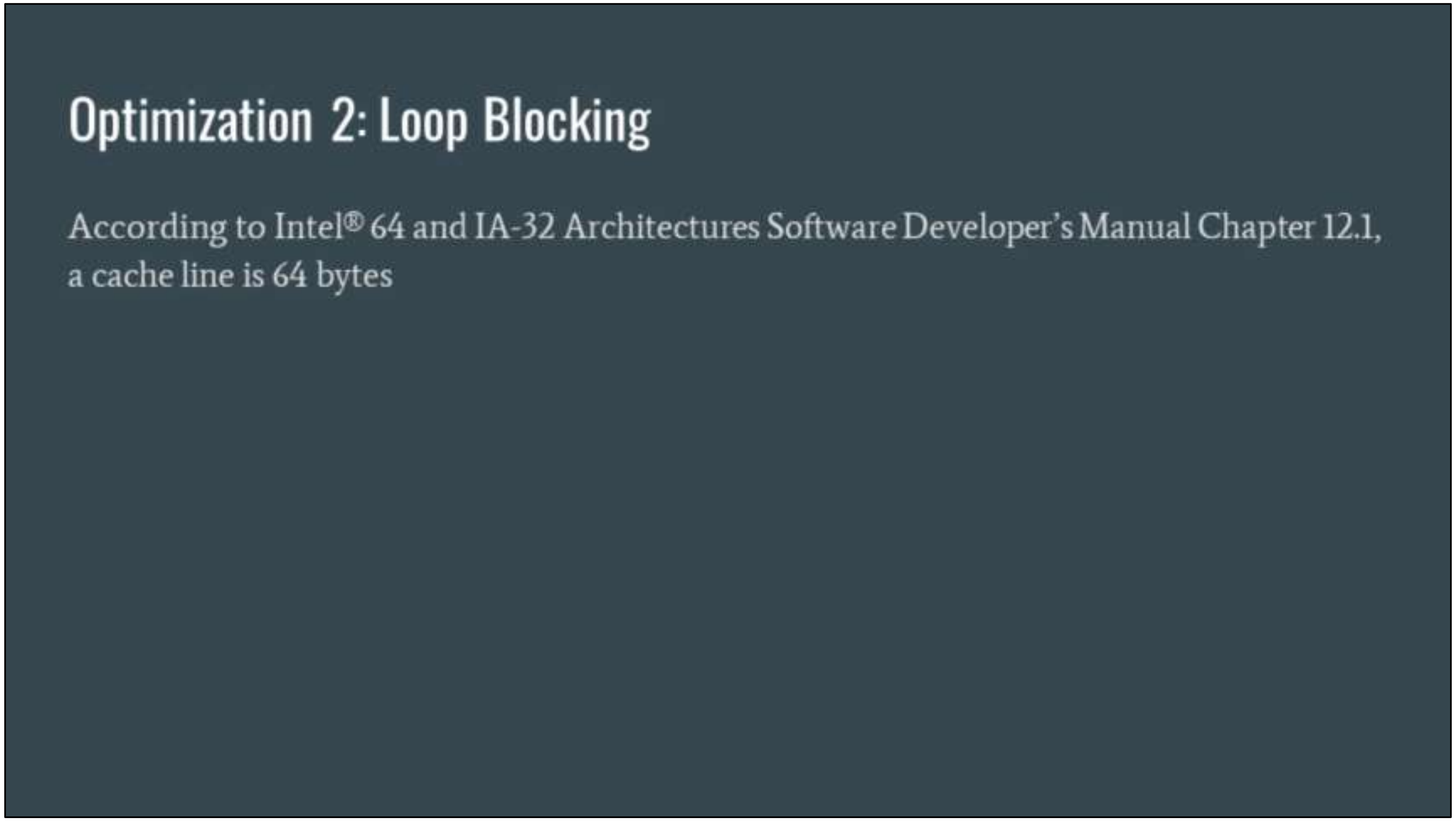
To get into what the reason for these performance problems are, we'll have to do a quick excourse so that we're all onboard regarding cpu caches.
I'll give you quick, high level overview about cpu caches - there is more to this than what is on this slides but the informations on these slides
Should be enough so that the upcoming optimizations make sense.

So, let's assume we load this image into memory (eg: using fread()).
If we now where to read the first pixel of this image (eg: via a pointer) the CPU will first check if the requested data is in one of its caches - more on that later.

If the data is in one of the caches - great.

In our case, where we want to access the first pixel of a newly loaded image however, the data will most likely not be in the CPU cache.
So what happens is that the data is getting read from main memory - this is orders of magnitude slower than accessing data from the CPU cache.

It won't only read one pixel worth of data though (assuming that one pixel is 32bit) - it will read a whole cache line worth of data and put that line into the CPU cache.



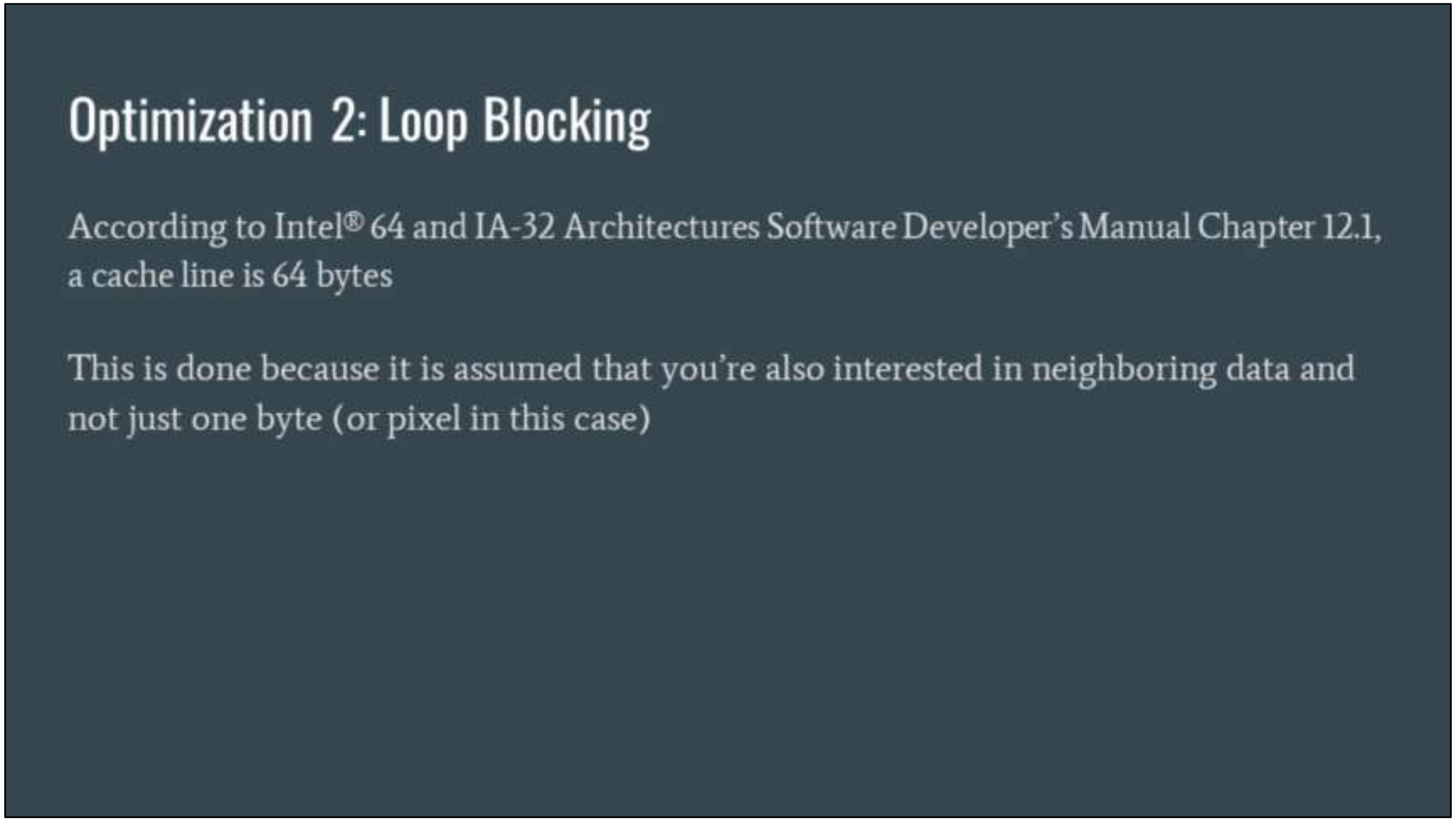
Optimization 2: Loop Blocking

According to Intel® 64 and IA-32 Architectures Software Developer's Manual Chapter 12.1, a cache line is 64 bytes

According to the documentation, a cache line is 64 byte - this could vary between different CPUs from different vendors, though.
CPUID can also be used to query the cache line size.

The CPU always reads a whole cache line when accessing even only 1 byte - this is done because it is assumed that you're also interested in neighboring data (eg. if you loop over an array and work on every item).

Additionally, there's a piece of hardware in the CPU called a prefetcher, this will detect sequential memory access and prefetch data from main memory into the CPU cache before it is actually accessed by your program.
By the time you actually do access this data, it'll already be in the CPU cache and you'll get a cache hit.



Optimization 2: Loop Blocking

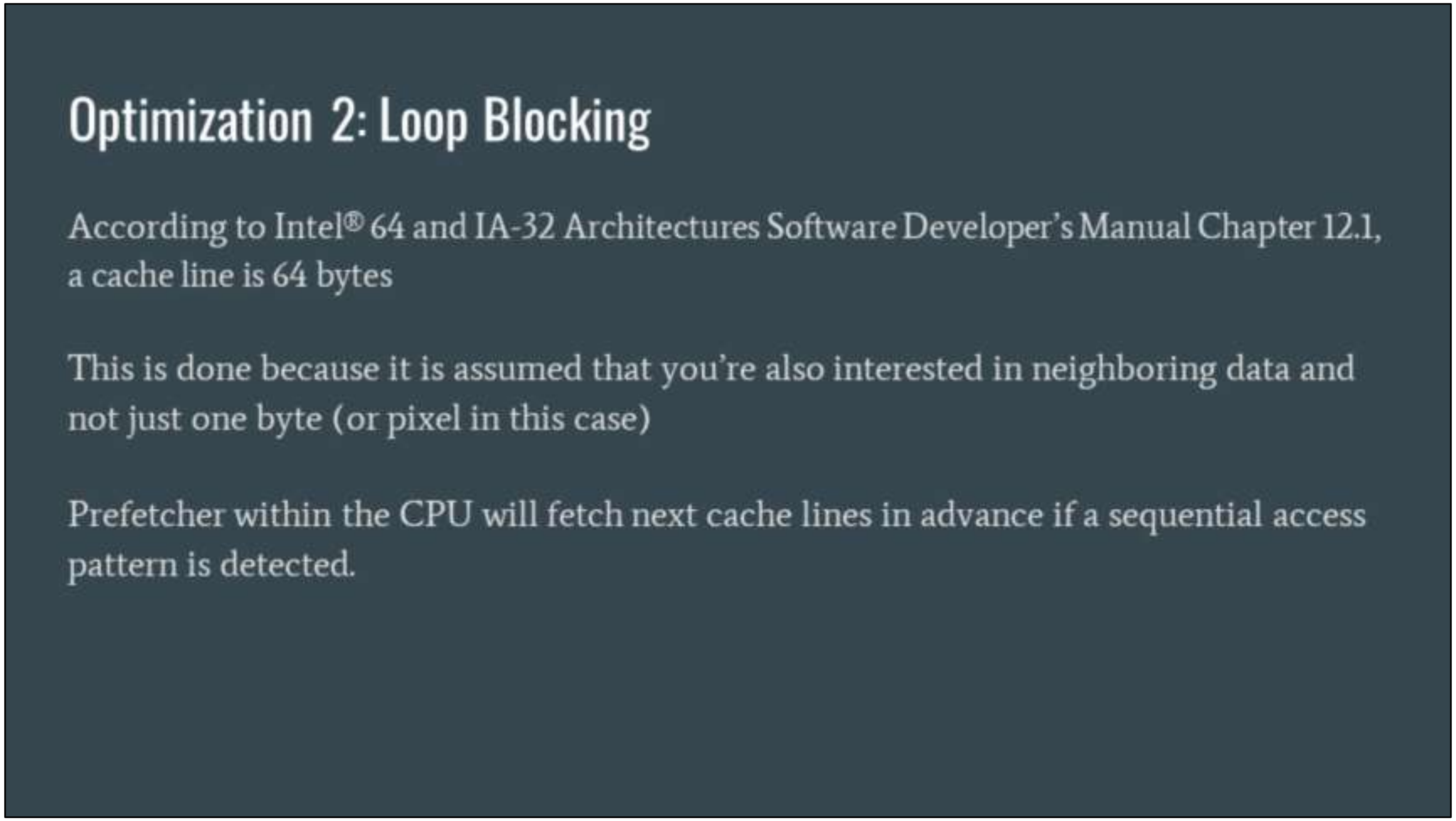
According to Intel® 64 and IA-32 Architectures Software Developer's Manual Chapter 12.1, a cache line is 64 bytes

This is done because it is assumed that you're also interested in neighboring data and not just one byte (or pixel in this case)

According to the documentation, a cache line is 64 byte - this could vary between different CPUs from different vendors, though.
CPUID can also be used to query the cache line size.

The CPU always reads a whole cache line when accessing even only 1 byte - this is done because it is assumed that you're also interested in neighboring data (eg. if you loop over an array and work on every item).

Additionally, there's a piece of hardware in the CPU called a prefetcher, this will detect sequential memory access and prefetch data from main memory into the CPU cache before it is actually accessed by your program.
By the time you actually do access this data, it'll already be in the CPU cache and you'll get a cache hit.



Optimization 2: Loop Blocking

According to Intel® 64 and IA-32 Architectures Software Developer's Manual Chapter 12.1, a cache line is 64 bytes

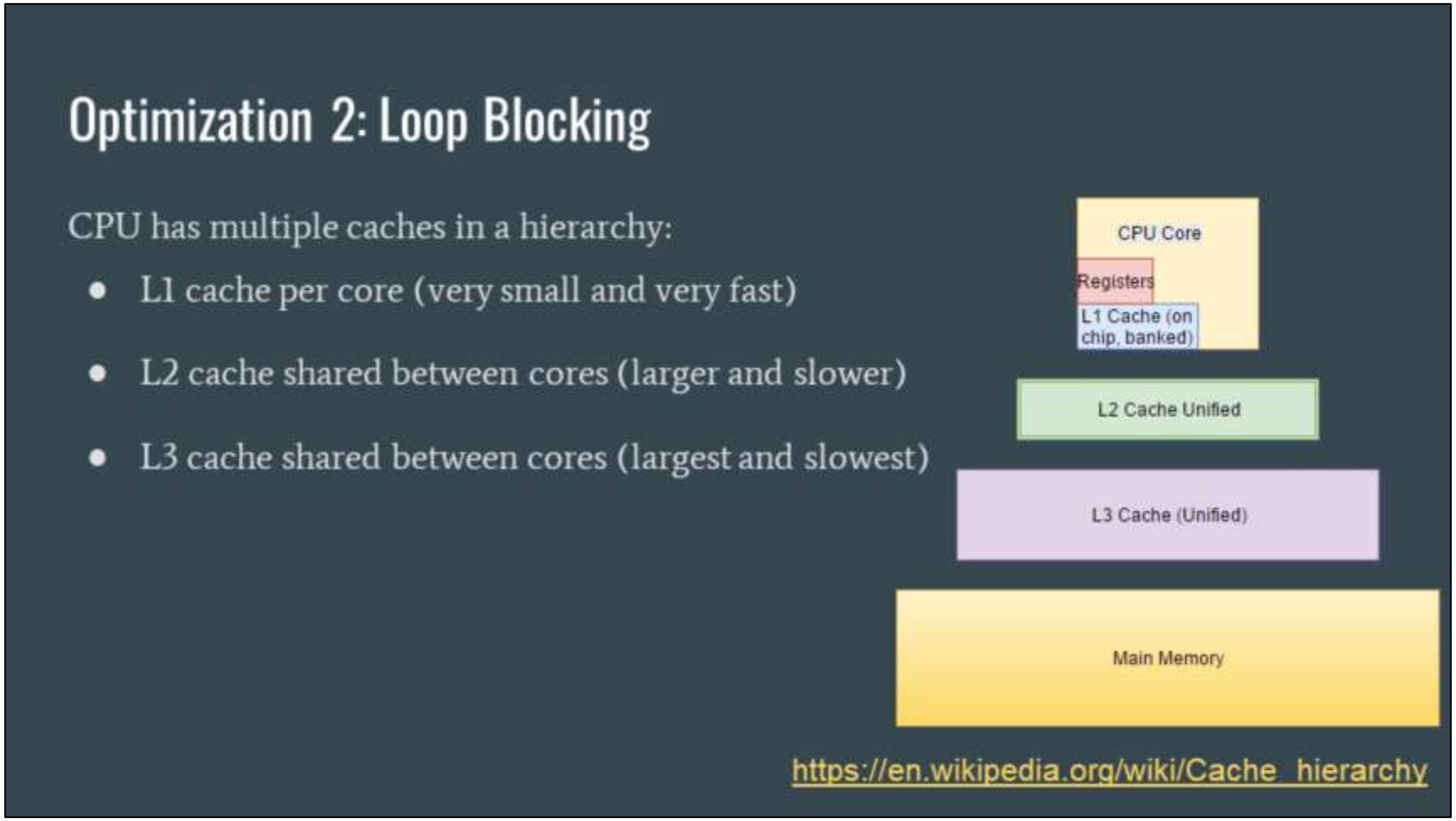
This is done because it is assumed that you're also interested in neighboring data and not just one byte (or pixel in this case)

Prefetcher within the CPU will fetch next cache lines in advance if a sequential access pattern is detected.

According to the documentation, a cache line is 64 byte - this could vary between different CPUs from different vendors, though.
CPUID can also be used to query the cache line size.

The CPU always reads a whole cache line when accessing even only 1 byte - this is done because it is assumed that you're also interested in neighboring data (eg. if you loop over an array and work on every item).

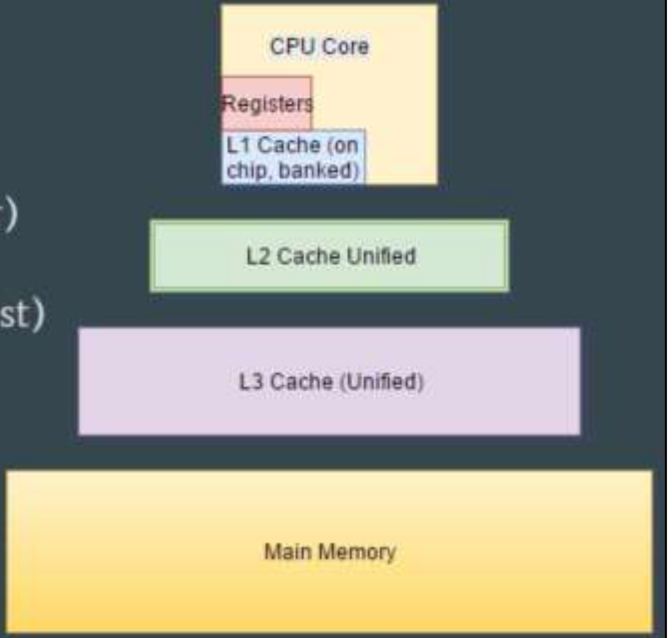
Additionally, there's a piece of hardware in the CPU called a prefetcher, this will detect sequential memory access and prefetch data from main memory into the CPU cache before it is actually accessed by your program.
By the time you actually do access this data, it'll already be in the CPU cache and you'll get a cache hit.



Optimization 2: Loop Blocking

CPU has multiple caches in a hierarchy:

- L1 cache per core (very small and very fast)
- L2 cache shared between cores (larger and slower)
- L3 cache shared between cores (largest and slowest)

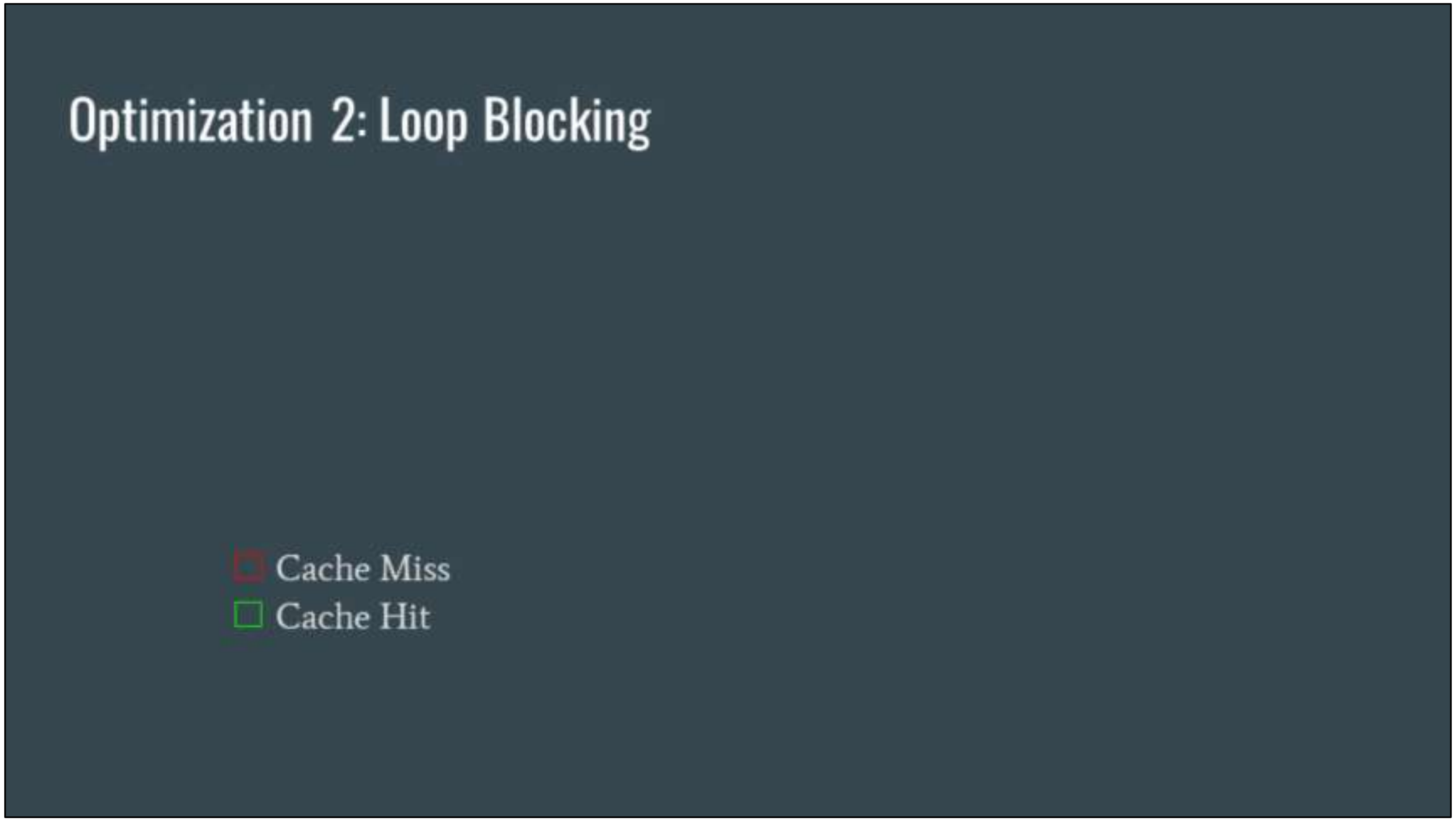


https://en.wikipedia.org/wiki/Cache_hierarchy

CPUs mostly have multiple caches with different characteristics.

You have your L1 cache. Every core has it’s own private L1 cache which is mostly broken down into a L1 Data cache and L1 Instruction cache.

Then you have a somewhat larger L2 cache shared between cores and an even bigger but slower L3 cache which is also shared between cores.



Optimization 2: Loop Blocking

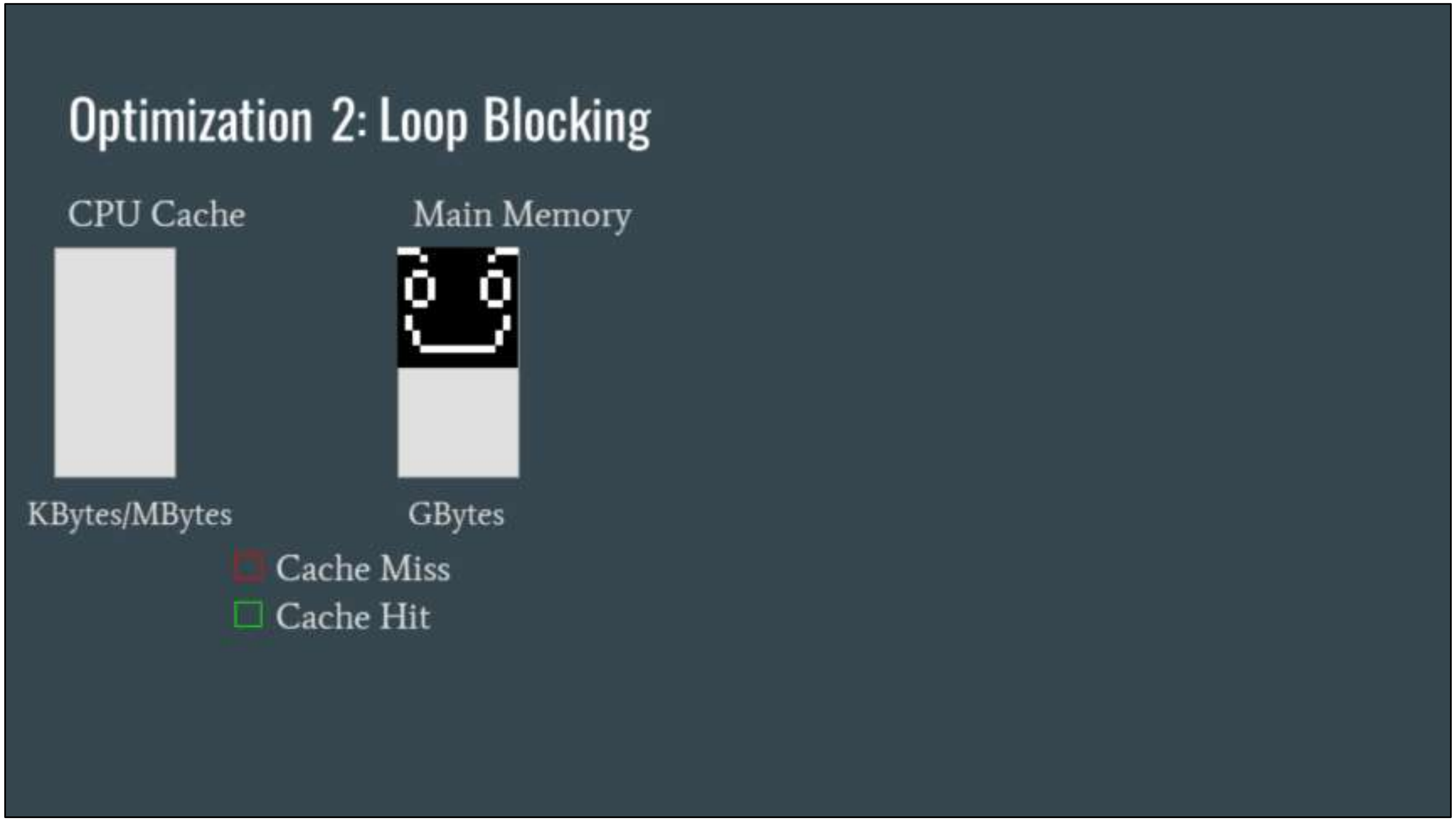
🟥 Cache Miss
🟩 Cache Hit

So let’s look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



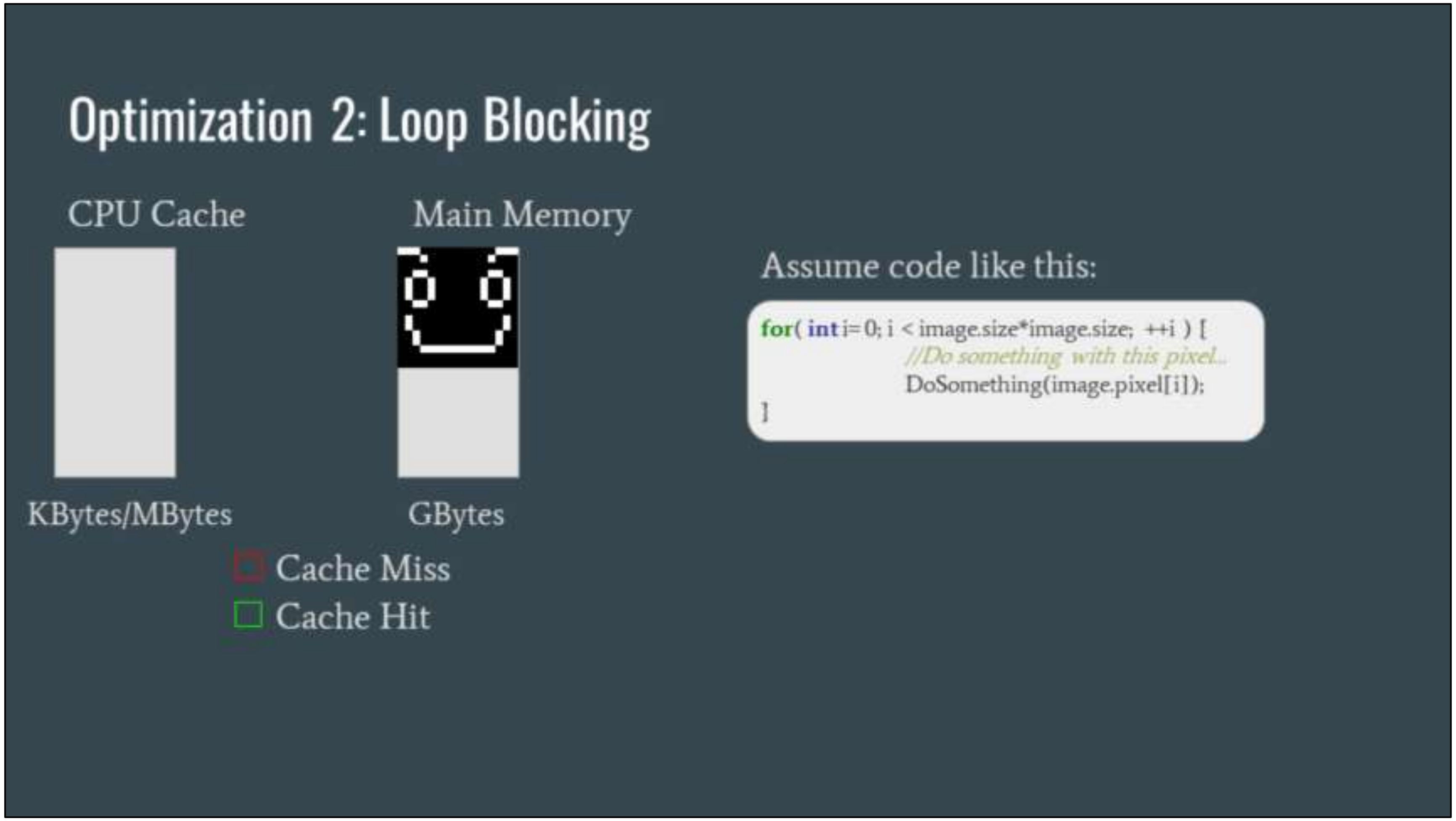
Optimization 2: Loop Blocking

So let's look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we'll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we'll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking

Assume code like this:

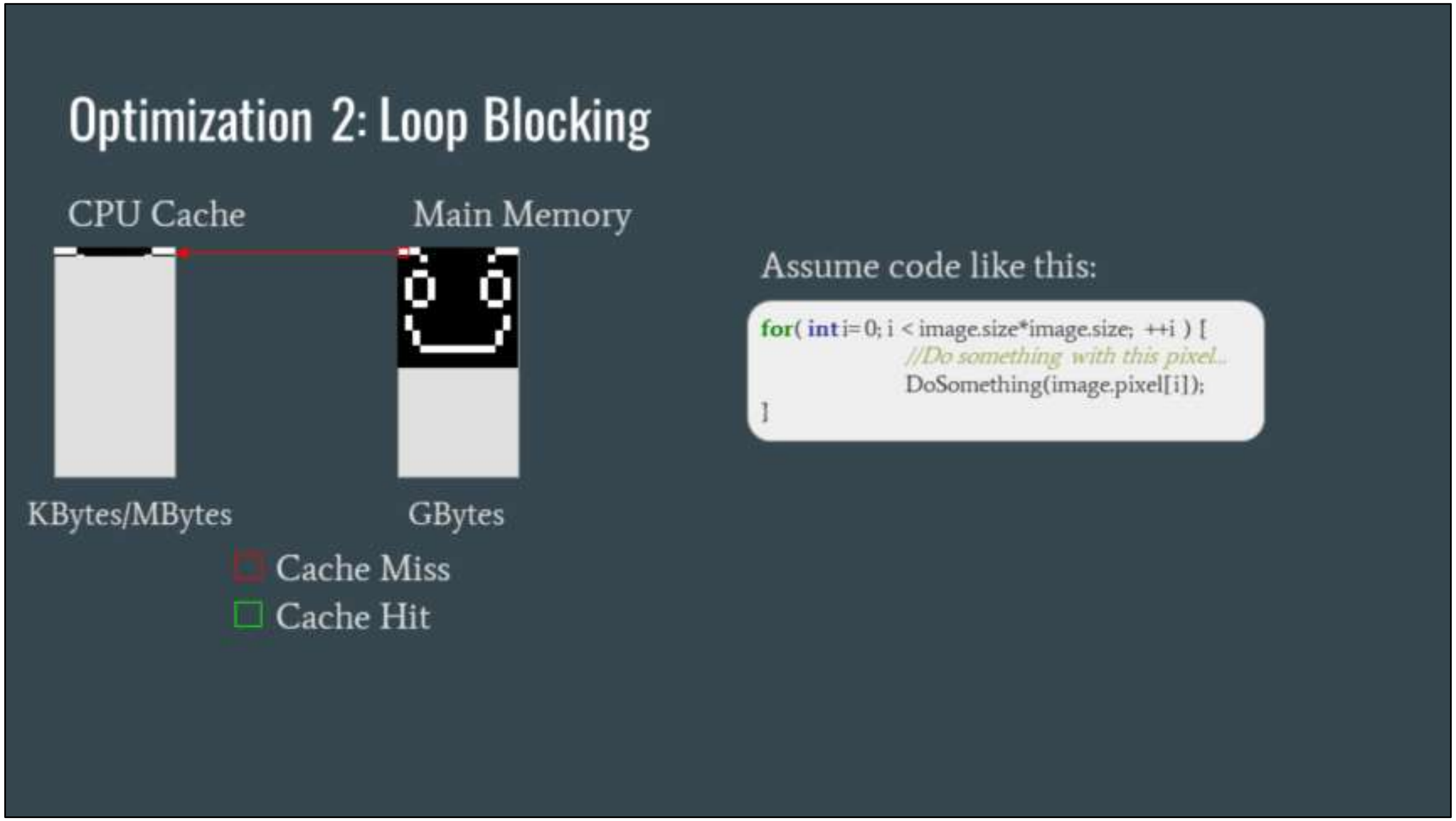
```
for( int i=0; i < image.size*image.size;  ++i ) {
            //Do something with this pixel...
            DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

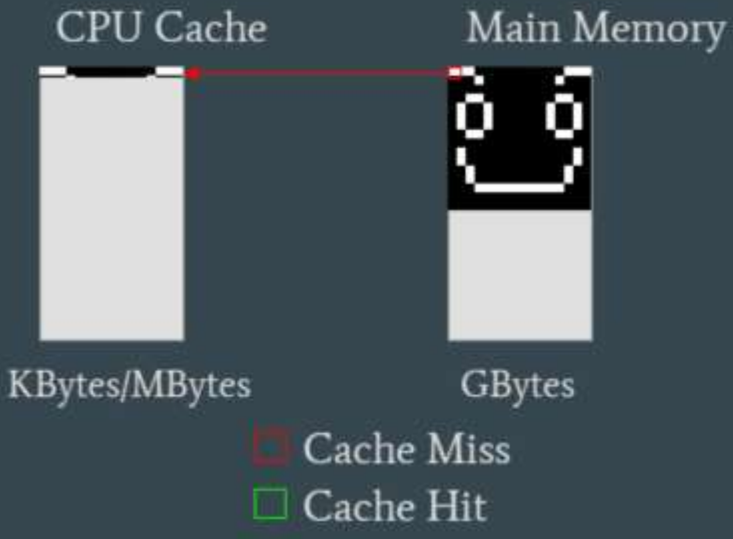
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

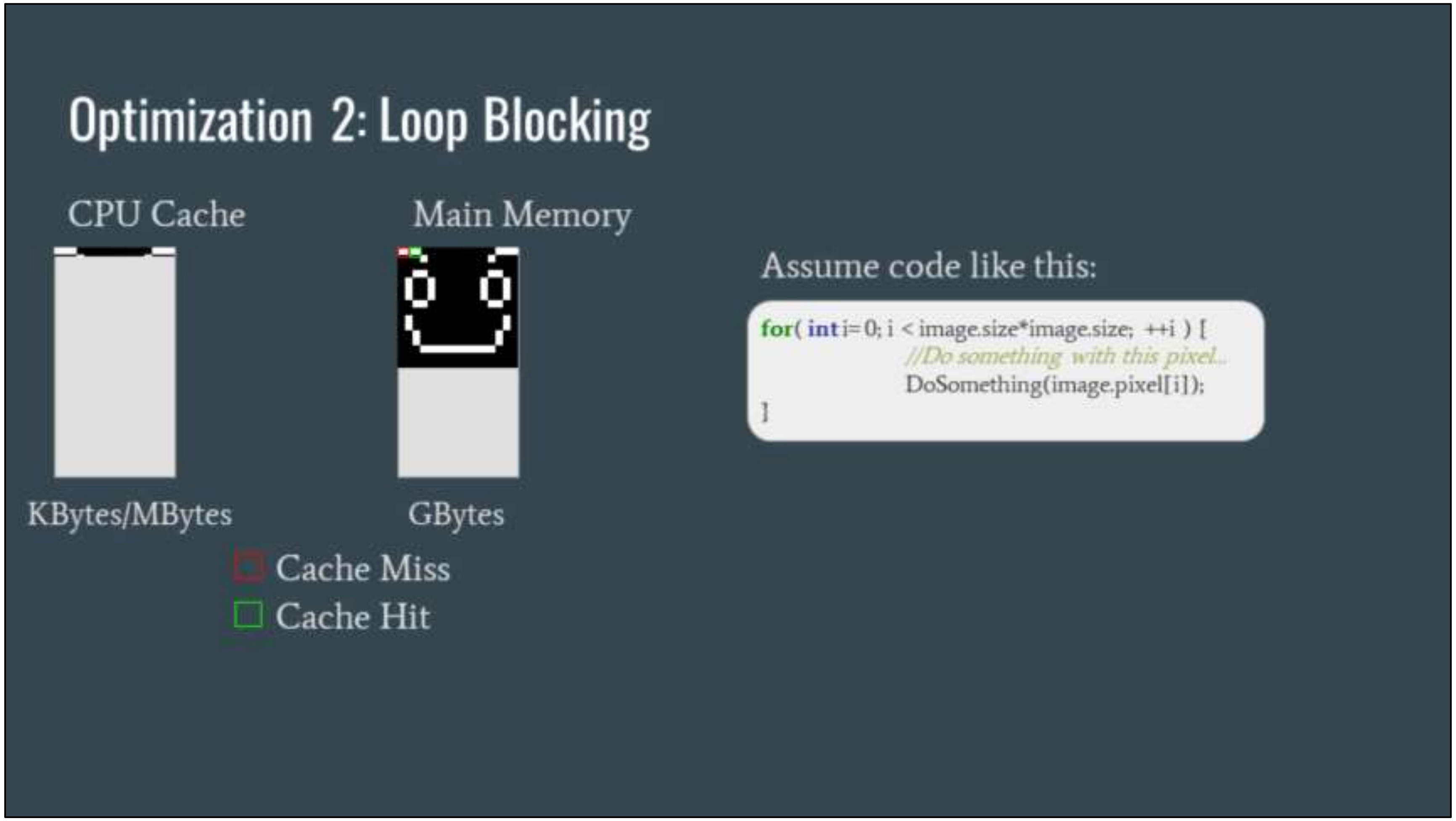
```
for( int i=0; i < image.size*image.size; ++i ) {
        //Do something with this pixel...
        DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

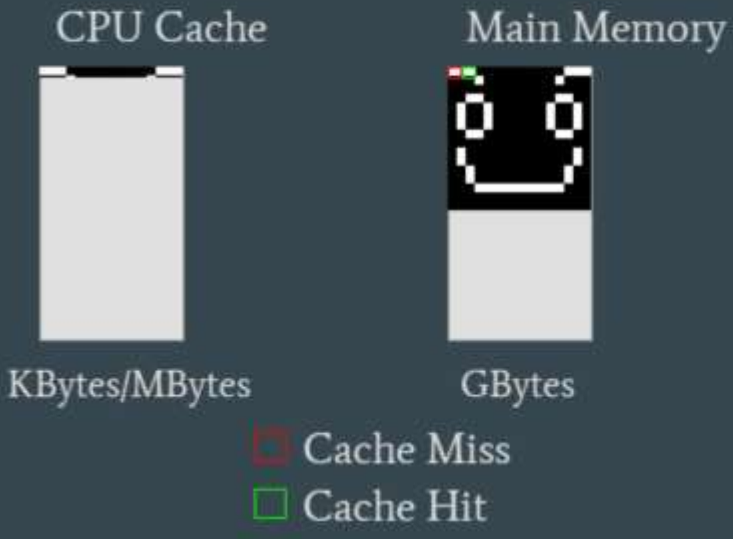
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

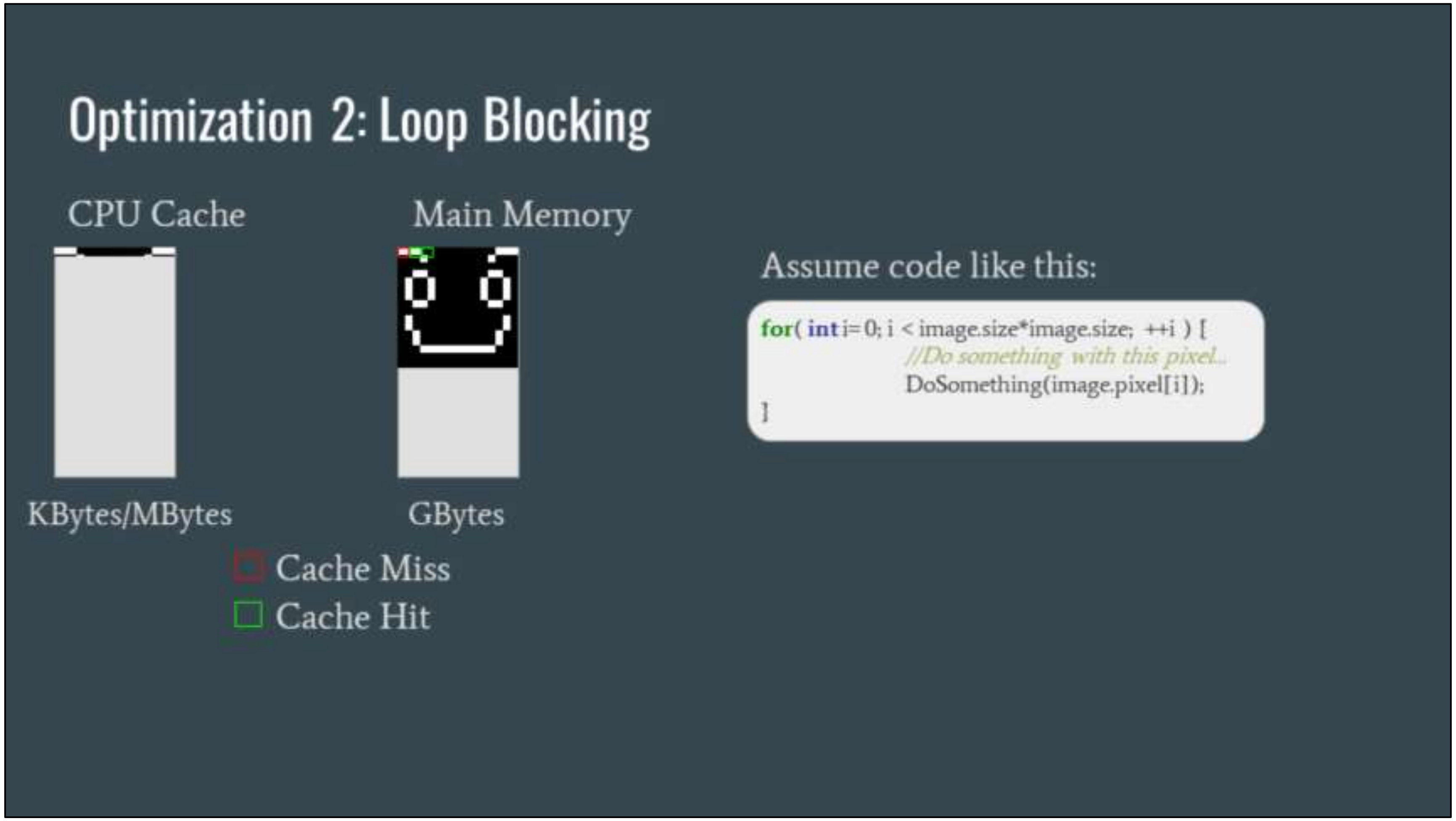
```
for( int i=0; i < image.size*image.size; ++i ) {
                //Do something with this pixel...
                DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

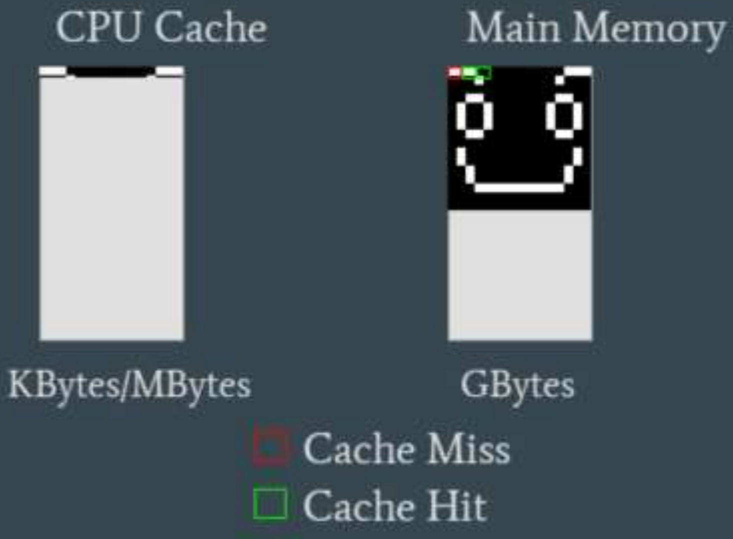
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

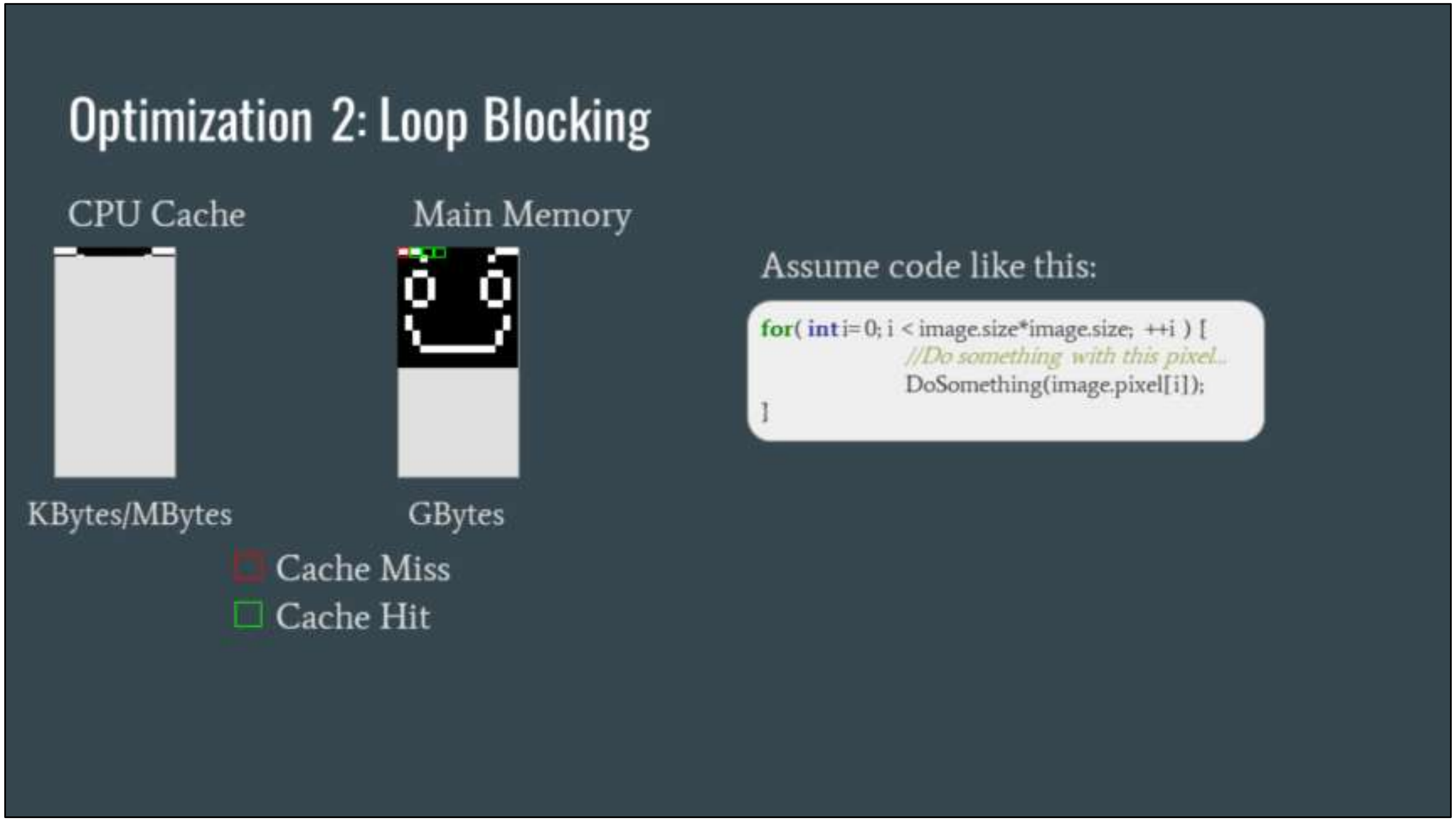
```
for( int i=0; i < image.size*image.size;  ++i ) {
            //Do something with this pixel...
            DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking

Assume code like this:

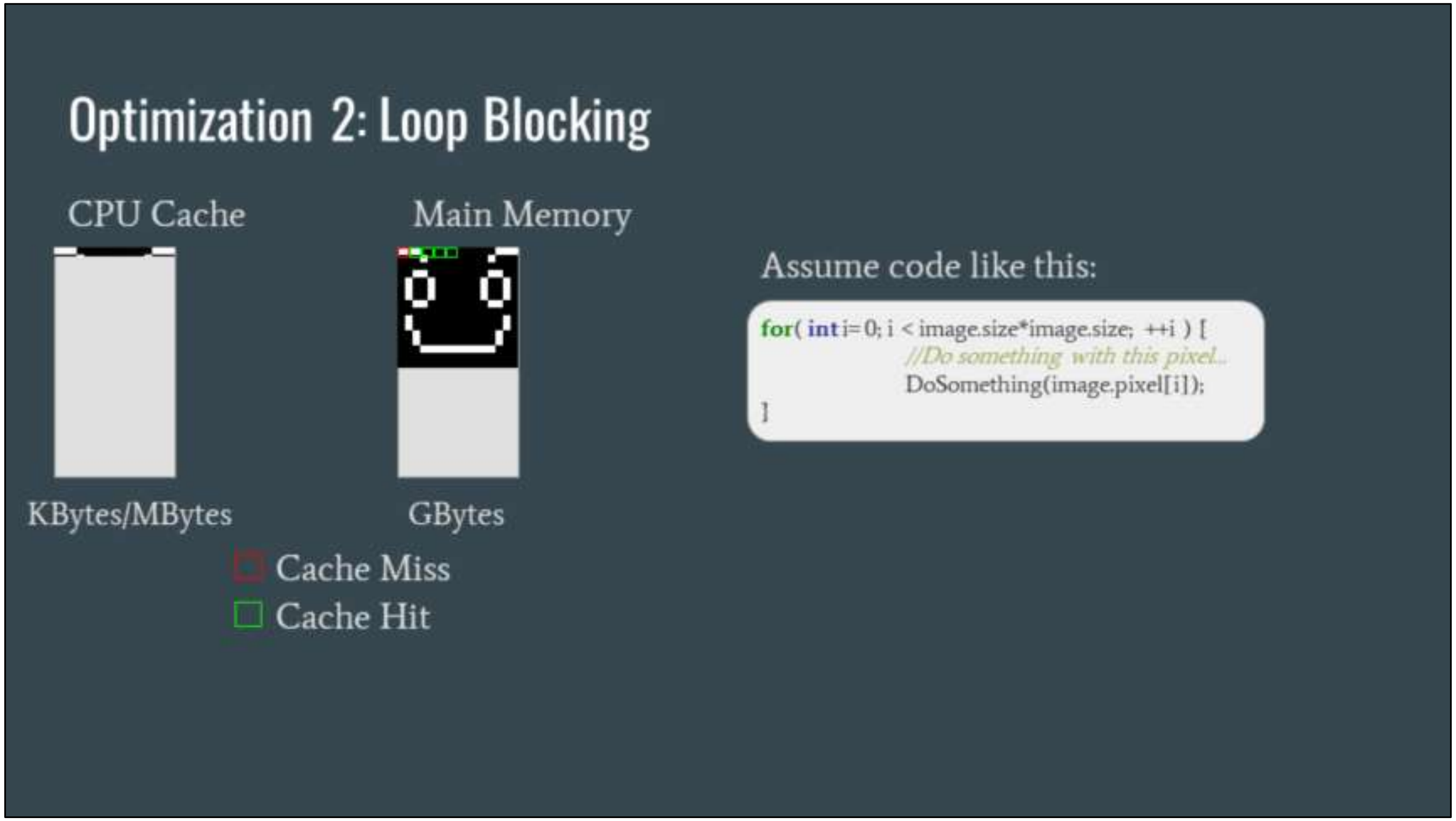
```
for( int i=0; i < image.size*image.size;  ++i ) {
                    //Do something with this pixel...
                    DoSomething(image.pixel[i]);
}
```

So let's look at this example again to understand how the prefetcher works.

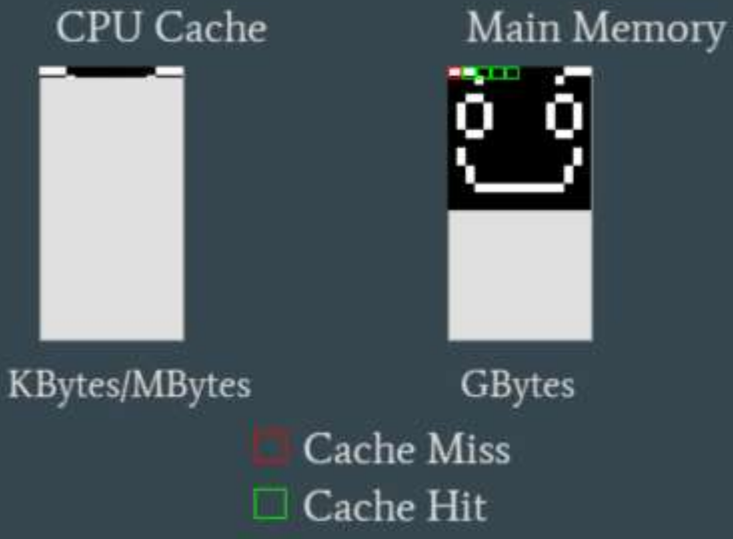
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we'll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we'll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

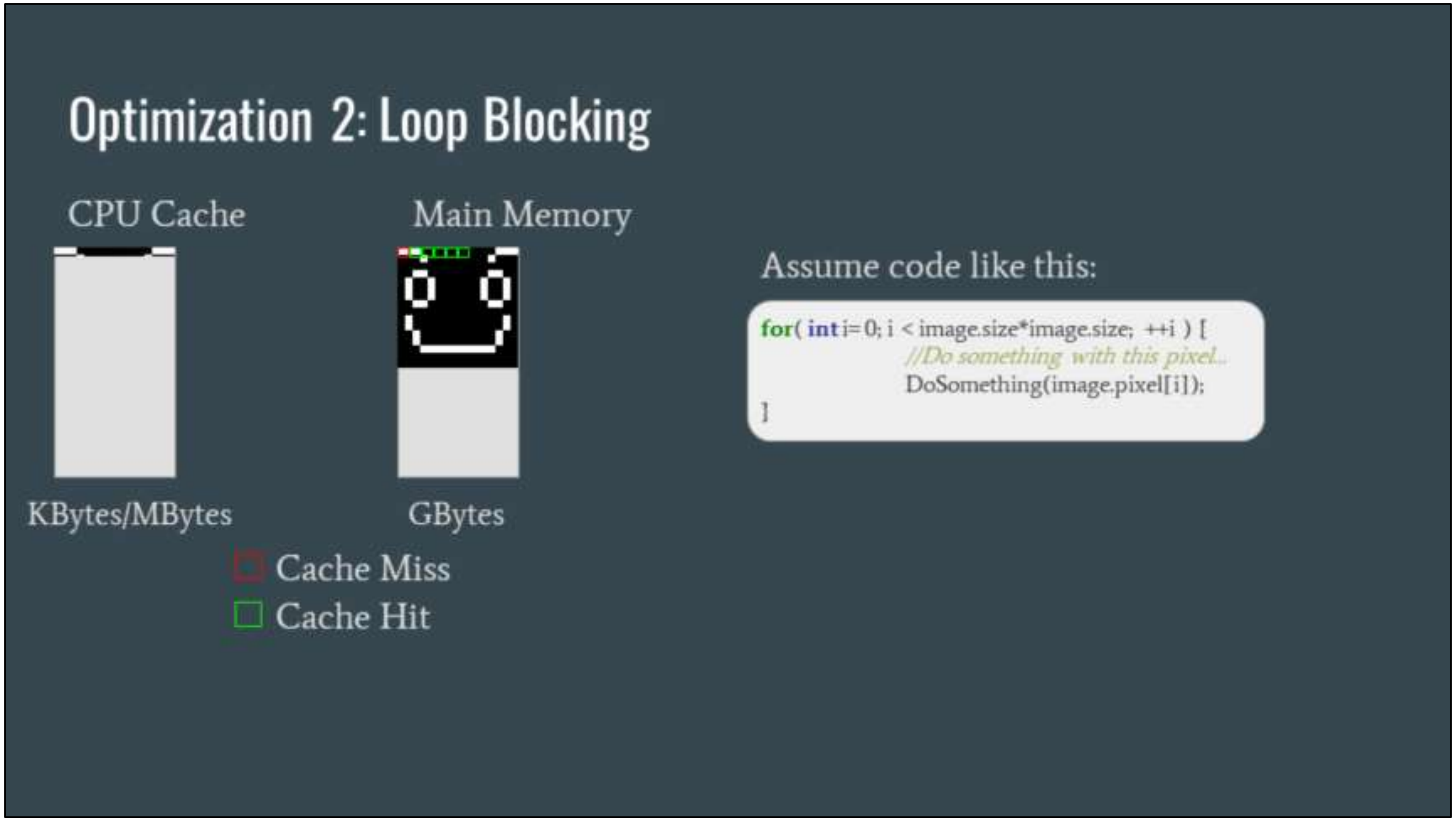
```
for( int i=0; i < image.size*image.size;  ++i ) {
            //Do something with this pixel...
            DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

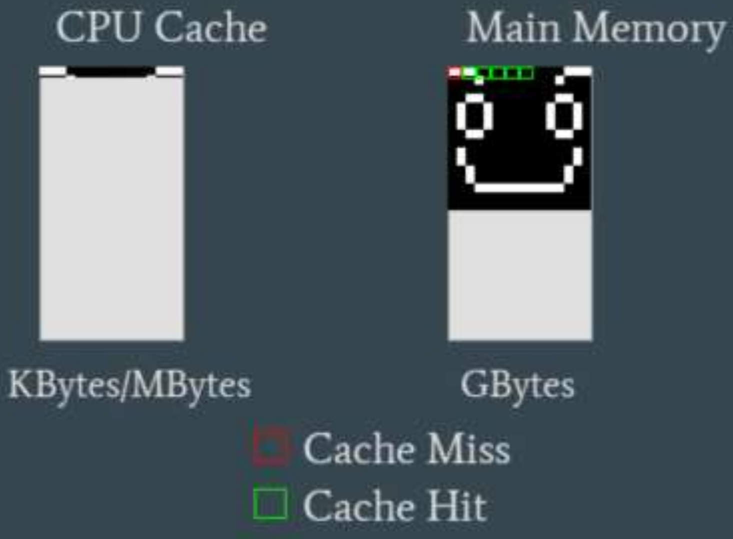
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

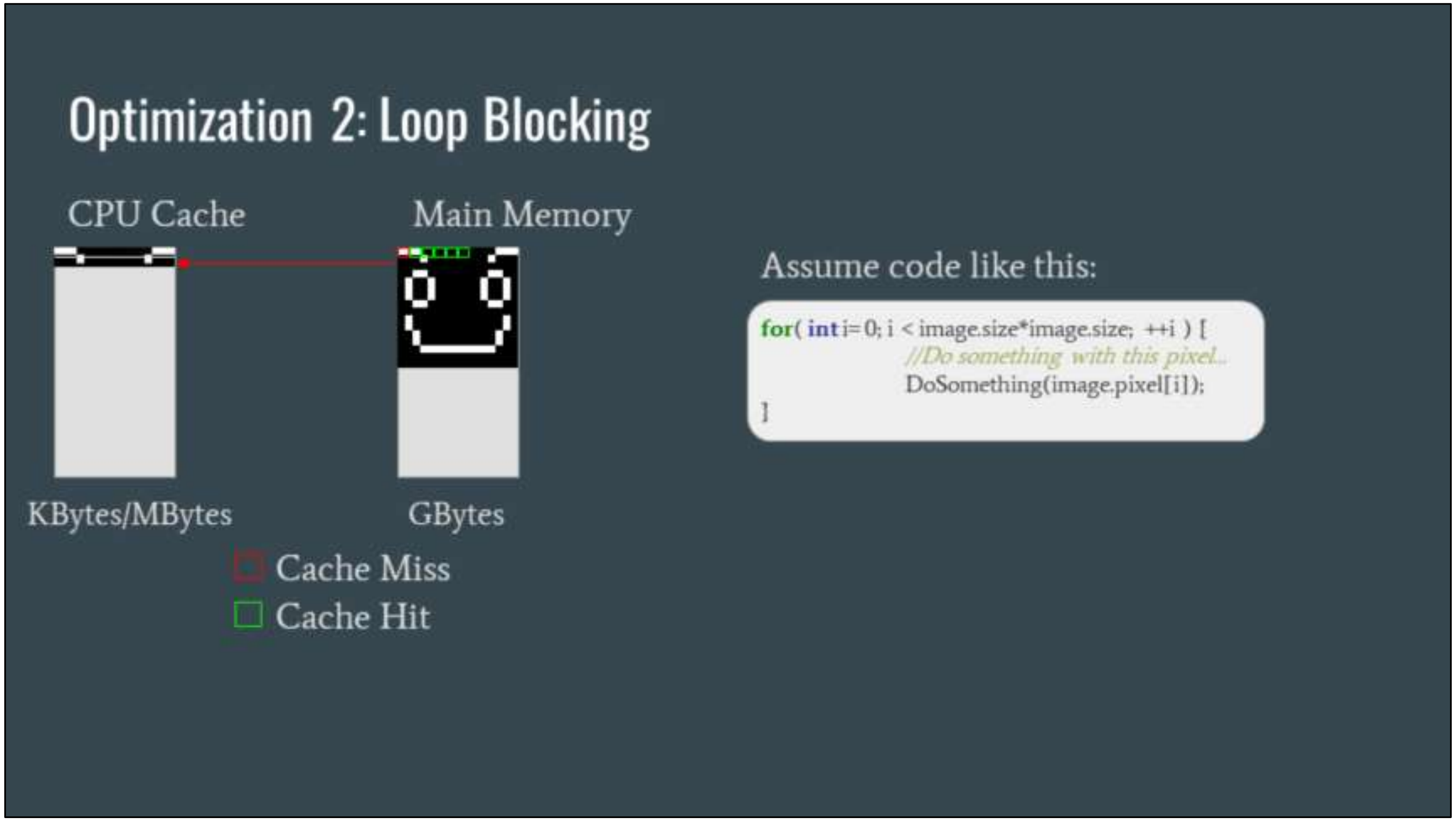
```
for( int i=0; i < image.size*image.size; ++i ) {
        //Do something with this pixel...
        DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

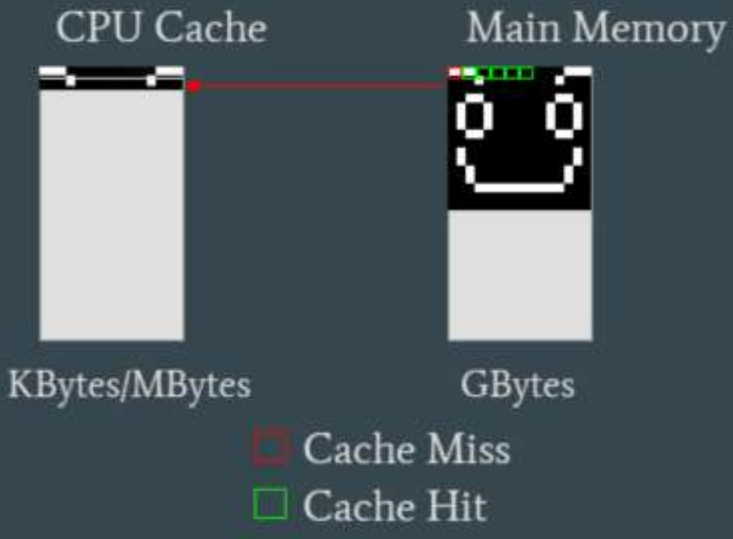
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

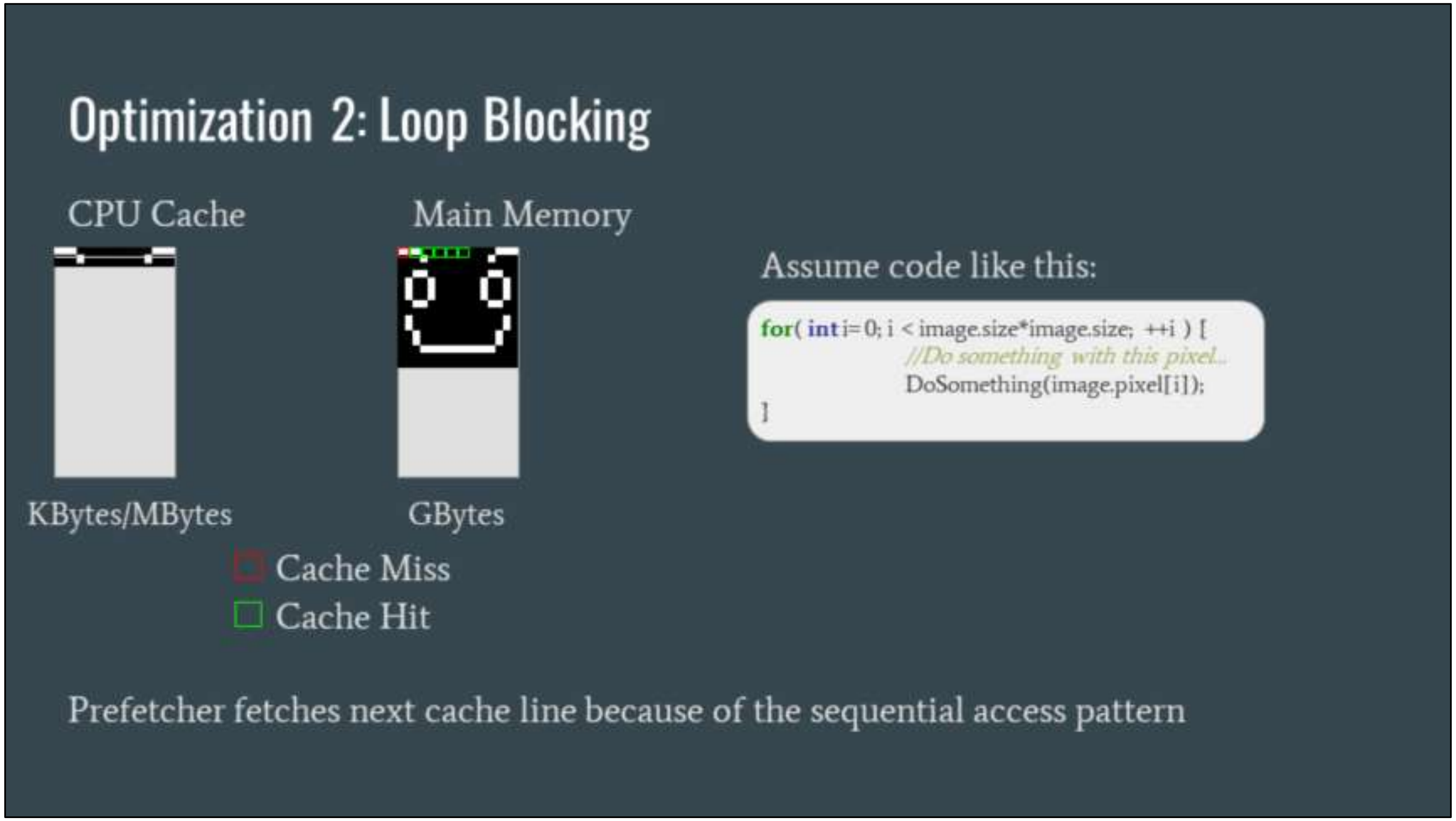
```
for( int i=0; i < image.size*image.size; ++i ) {
        //Do something with this pixel...
        DoSomething(image.pixel[i]);
}
```

So let’s look at this example again to understand how the prefetcher works.

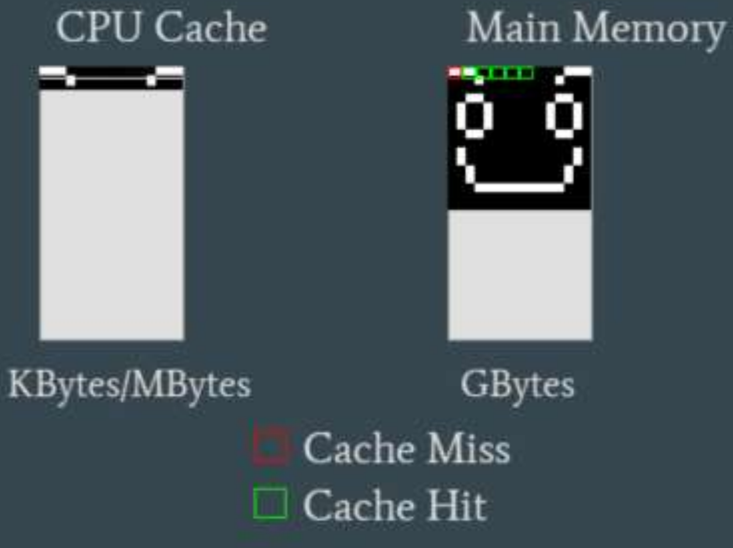
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

```
for( int i=0; i < image.size*image.size; ++i ) {
        //Do something with this pixel...
        DoSomething(image.pixel[i]);
}
```

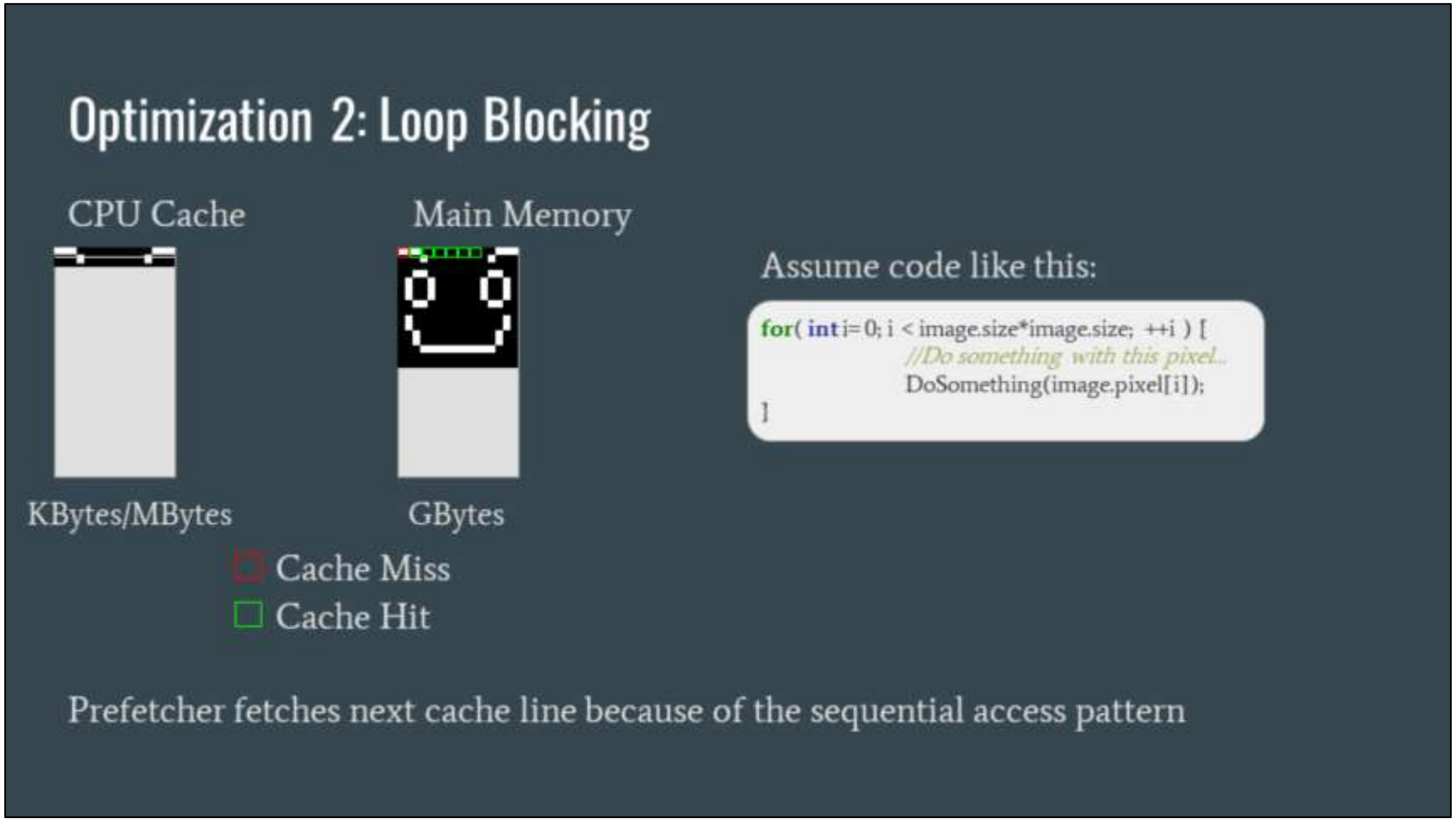
Prefetcher fetches next cache line because of the sequential access pattern

So let's look at this example again to understand how the prefetcher works.

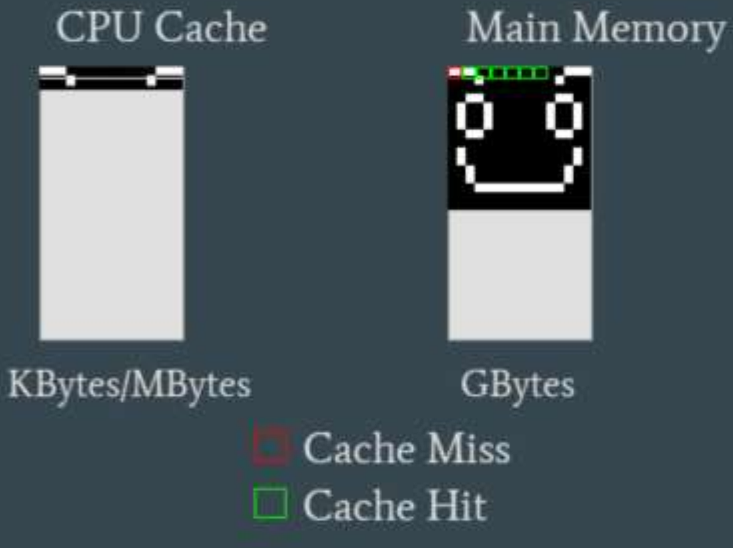
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we'll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we'll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

```
for( int i=0; i < image.size*image.size; ++i ) {
            //Do something with this pixel...
            DoSomething(image.pixel[i]);
}
```

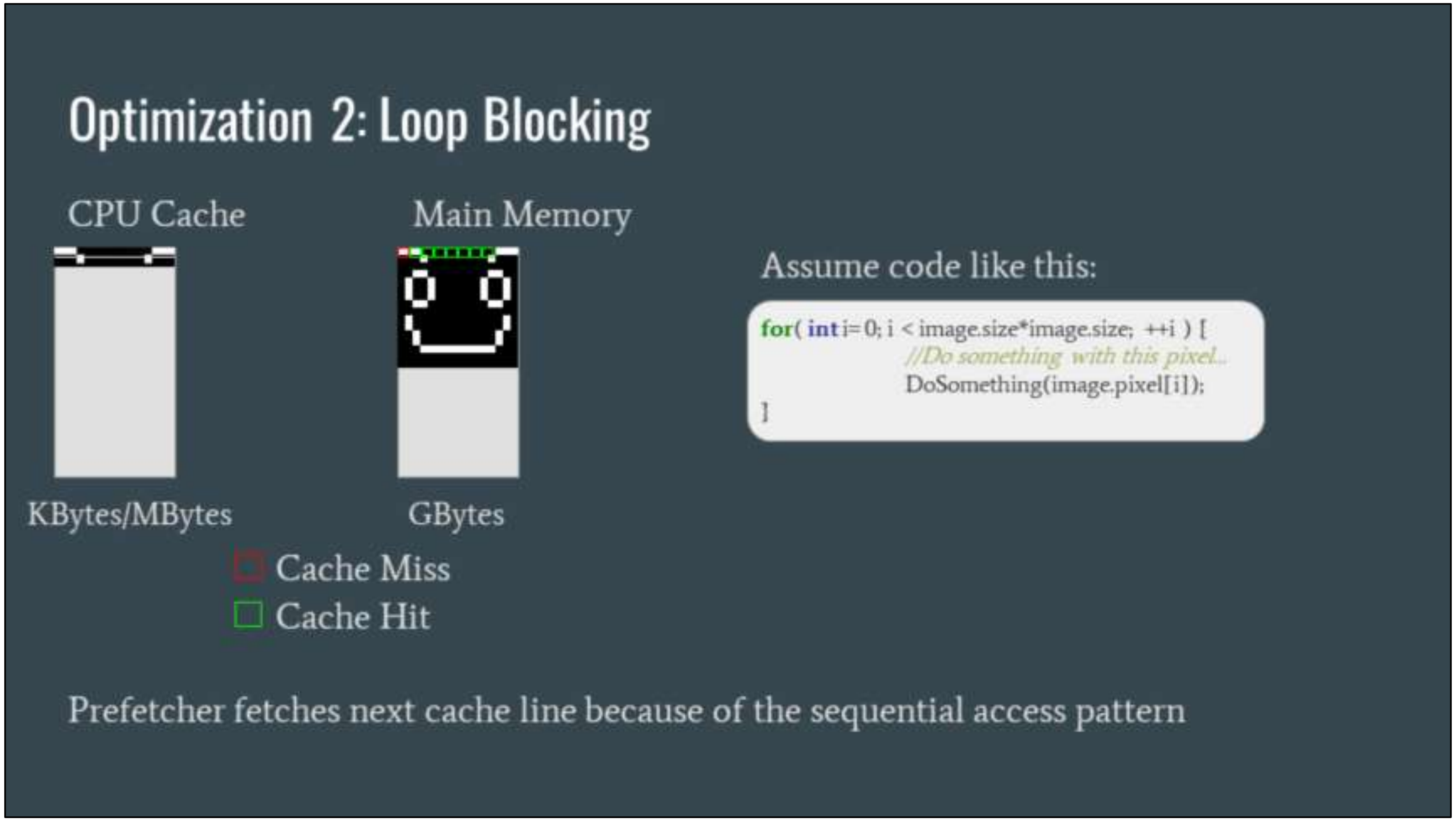
Prefetcher fetches next cache line because of the sequential access pattern

So let’s look at this example again to understand how the prefetcher works.

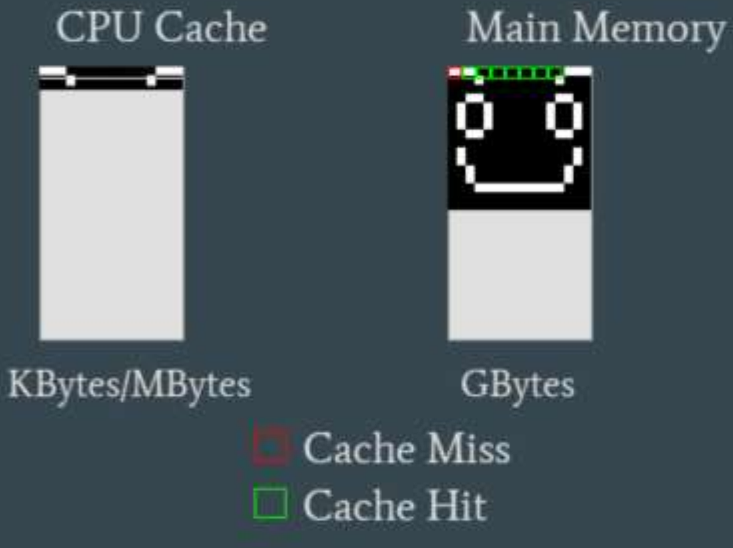
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

```
for( int i=0; i < image.size*image.size; ++i ) {
        //Do something with this pixel...
        DoSomething(image.pixel[i]);
}
```

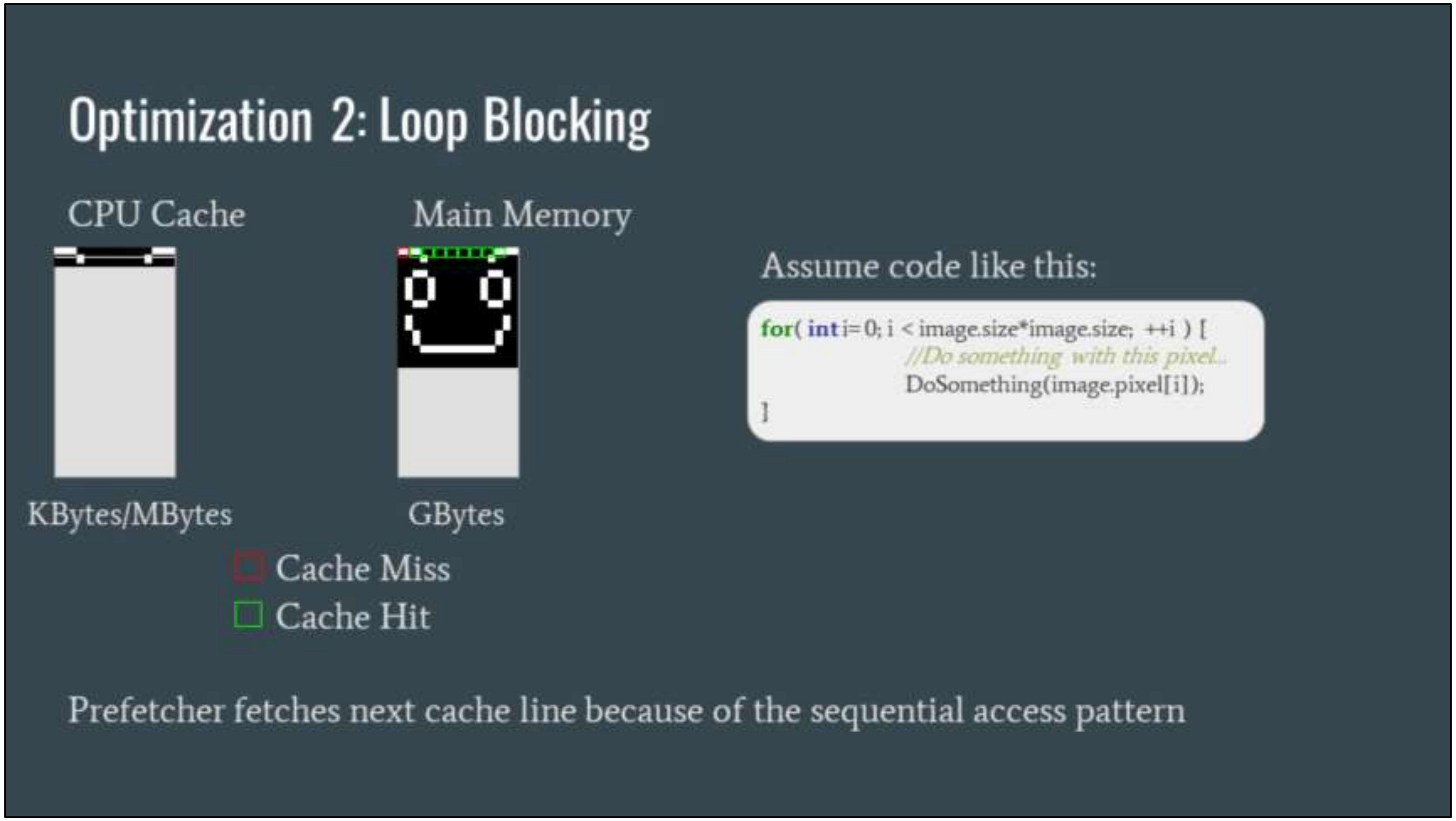
Prefetcher fetches next cache line because of the sequential access pattern

So let’s look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking

Assume code like this:

```
for( int i=0; i < image.size*image.size;  ++i ) {
                //Do something with this pixel...
                DoSomething(image.pixel[i]);
}
```

Prefetcher fetches next cache line because of the sequential access pattern

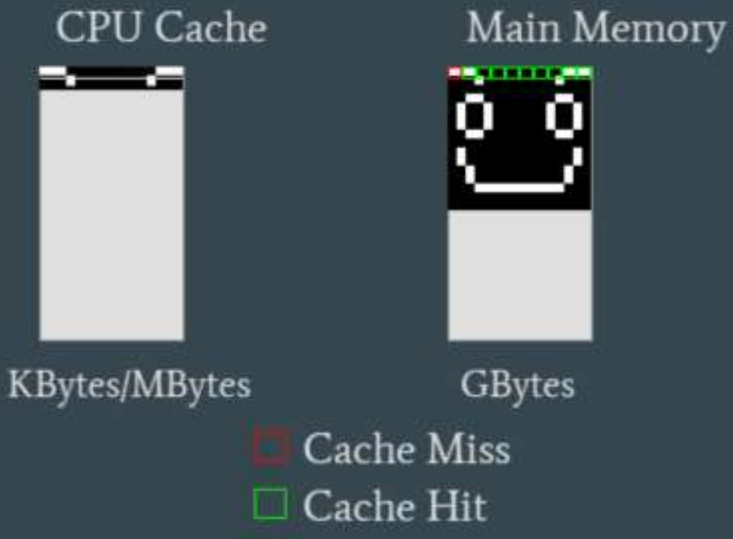
So let’s look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.

Optimization 2: Loop Blocking



Assume code like this:

```
for( int i=0; i < image.size*image.size;  ++i ) {
                //Do something with this pixel...
                DoSomething(image.pixel[i]);
}
```

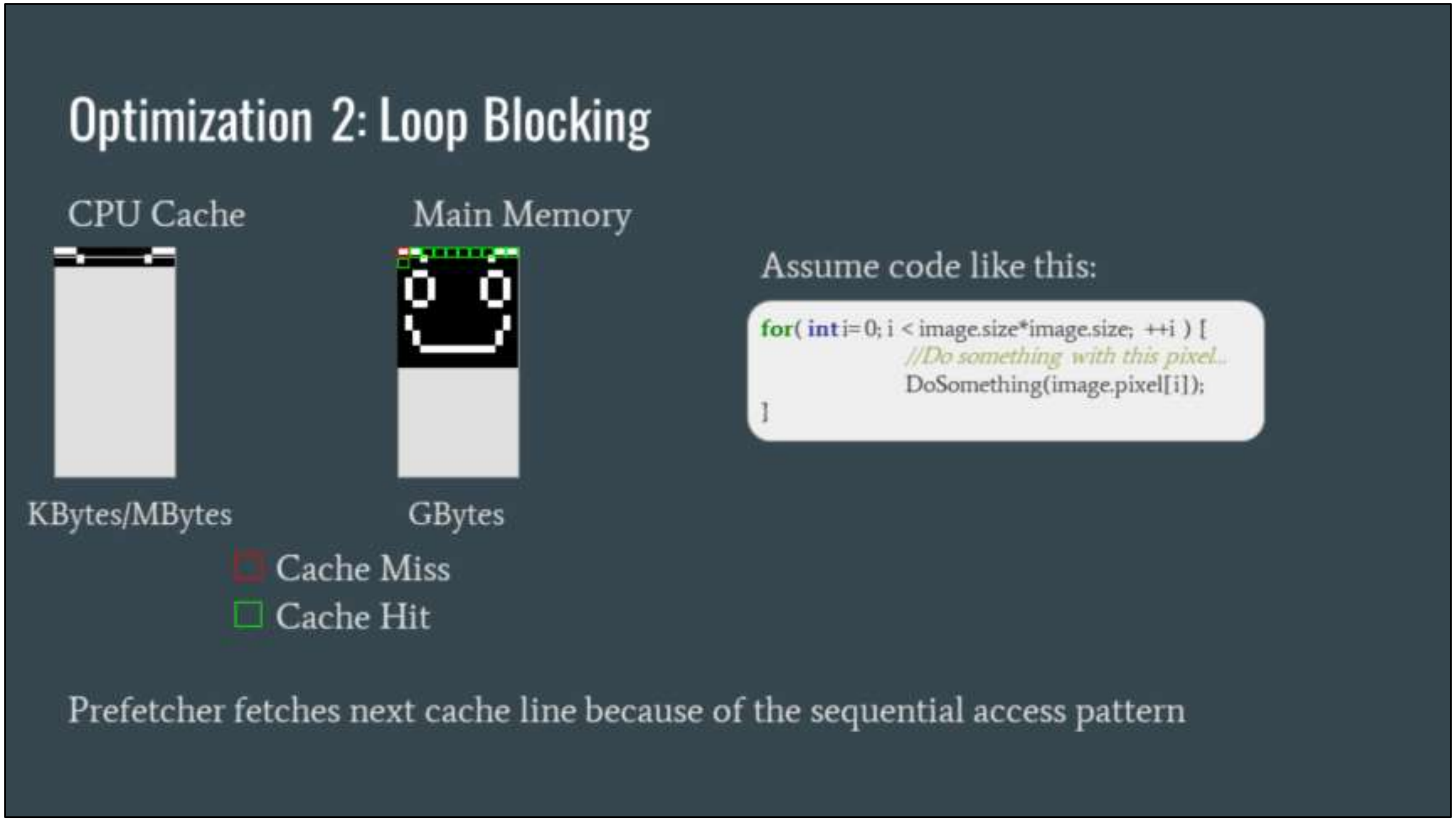
Prefetcher fetches next cache line because of the sequential access pattern

So let’s look at this example again to understand how the prefetcher works.

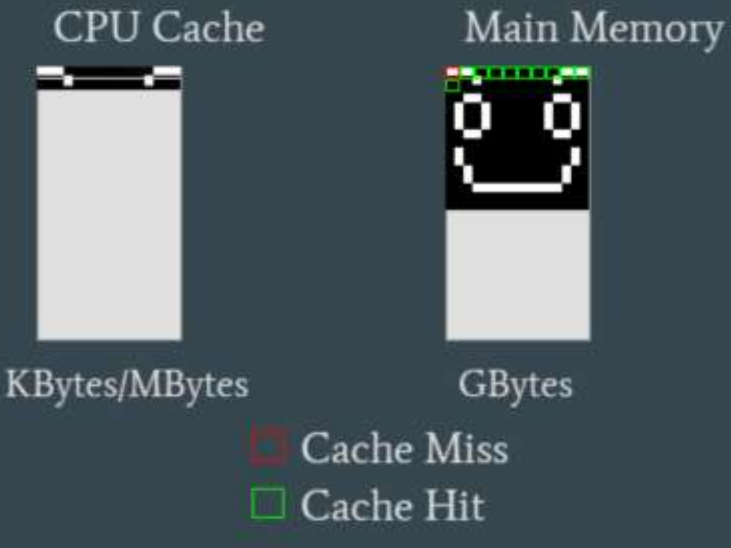
Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking



Assume code like this:

```
for( int i=0; i < image.size*image.size;  ++i ) {
                //Do something with this pixel...
                DoSomething(image.pixel[i]);
}
```

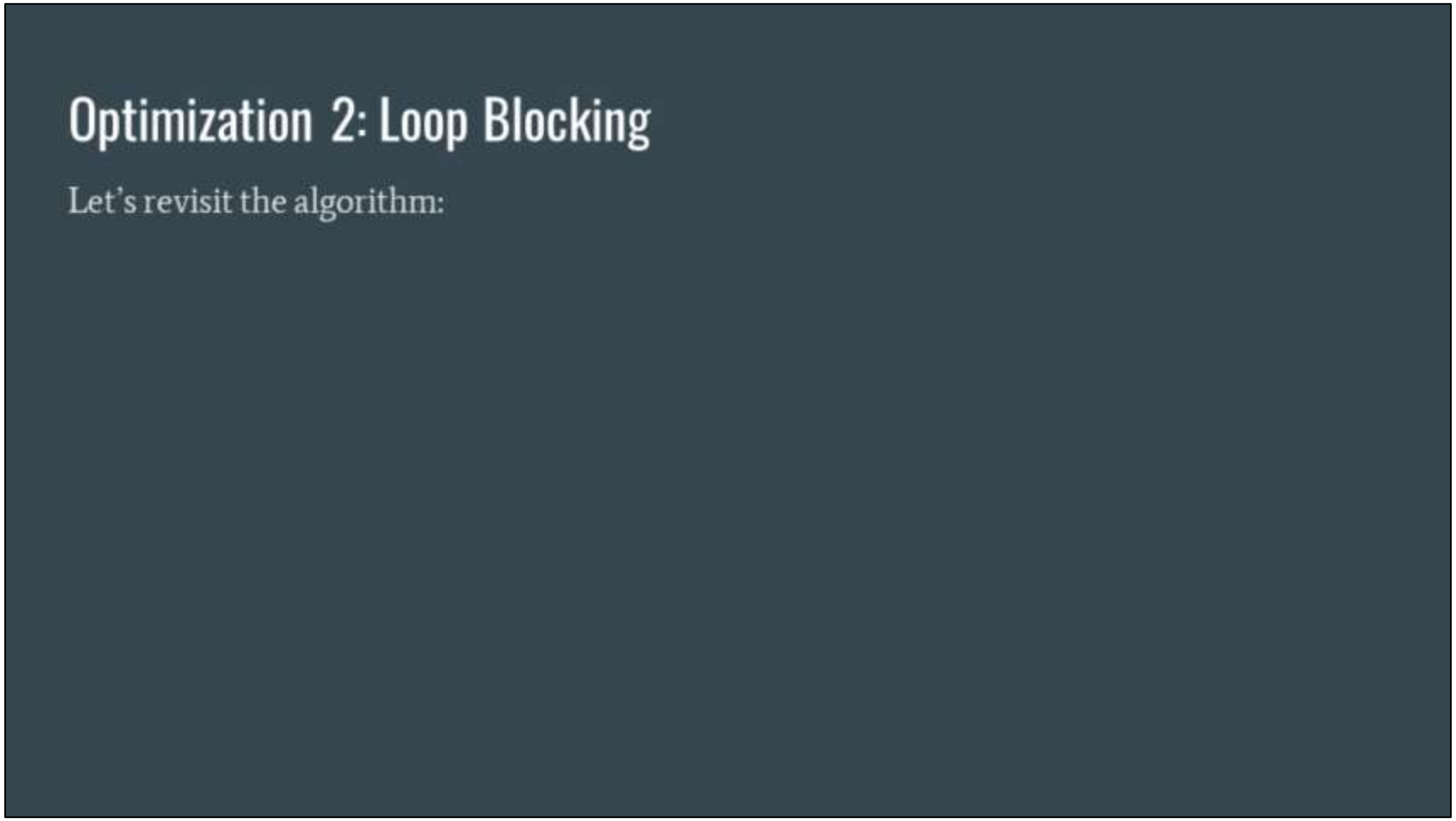
Prefetcher fetches next cache line because of the sequential access pattern

So let’s look at this example again to understand how the prefetcher works.

Assume again that this image just got loaded into main memory and we have this loop that iterates over each pixel and does *something* with that pixel.

For the first access, we’ll get a cache miss since the data is most likely not in the CPU cache.
For successive accesses however, we’ll get cache hits because the whole cache line was loaded into the CPU cache.

At some time during this iteration, the prefetcher will kick in and prefetch the next cache line because a sequential access pattern has been detected.
By the time we exhausted the data of the first cache line, the next cache line has already been loaded into the cache because of this, resulting in cache hits.



Optimization 2: Loop Blocking

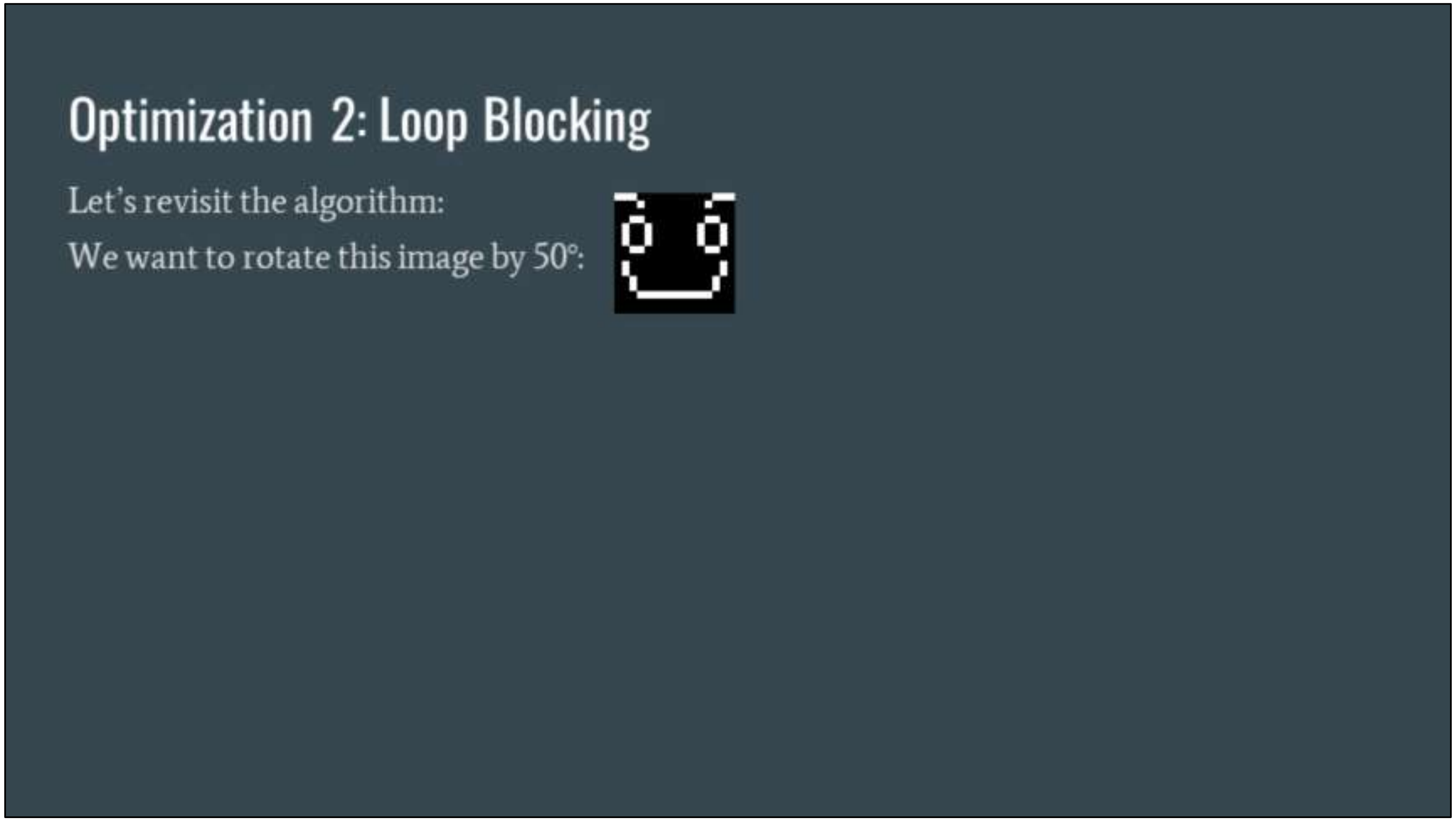
Let's revisit the algorithm:

With all this in mind, let's revisit our algorithm to answer the question "why are we memory bound and have so many LLC misses?"

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Let’s revisit the algorithm:

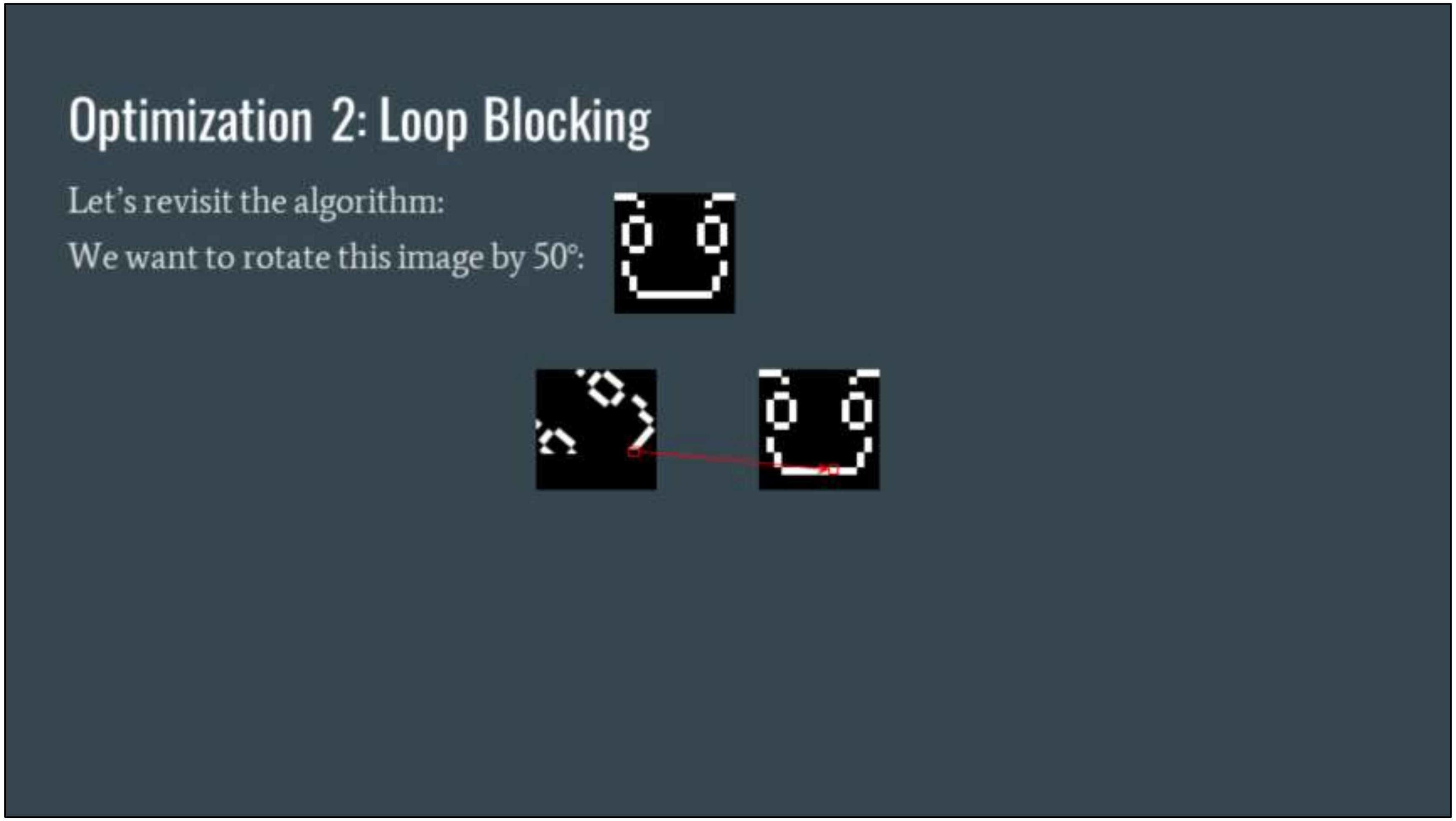
We want to rotate this image by 50°:

With all this in mind, let’s revisit our algorithm to answer the question “why are we memory bound and have so many LLC misses?”

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

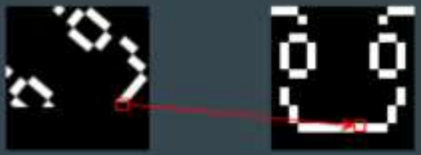
The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Let's revisit the algorithm:

We want to rotate this image by 50°:

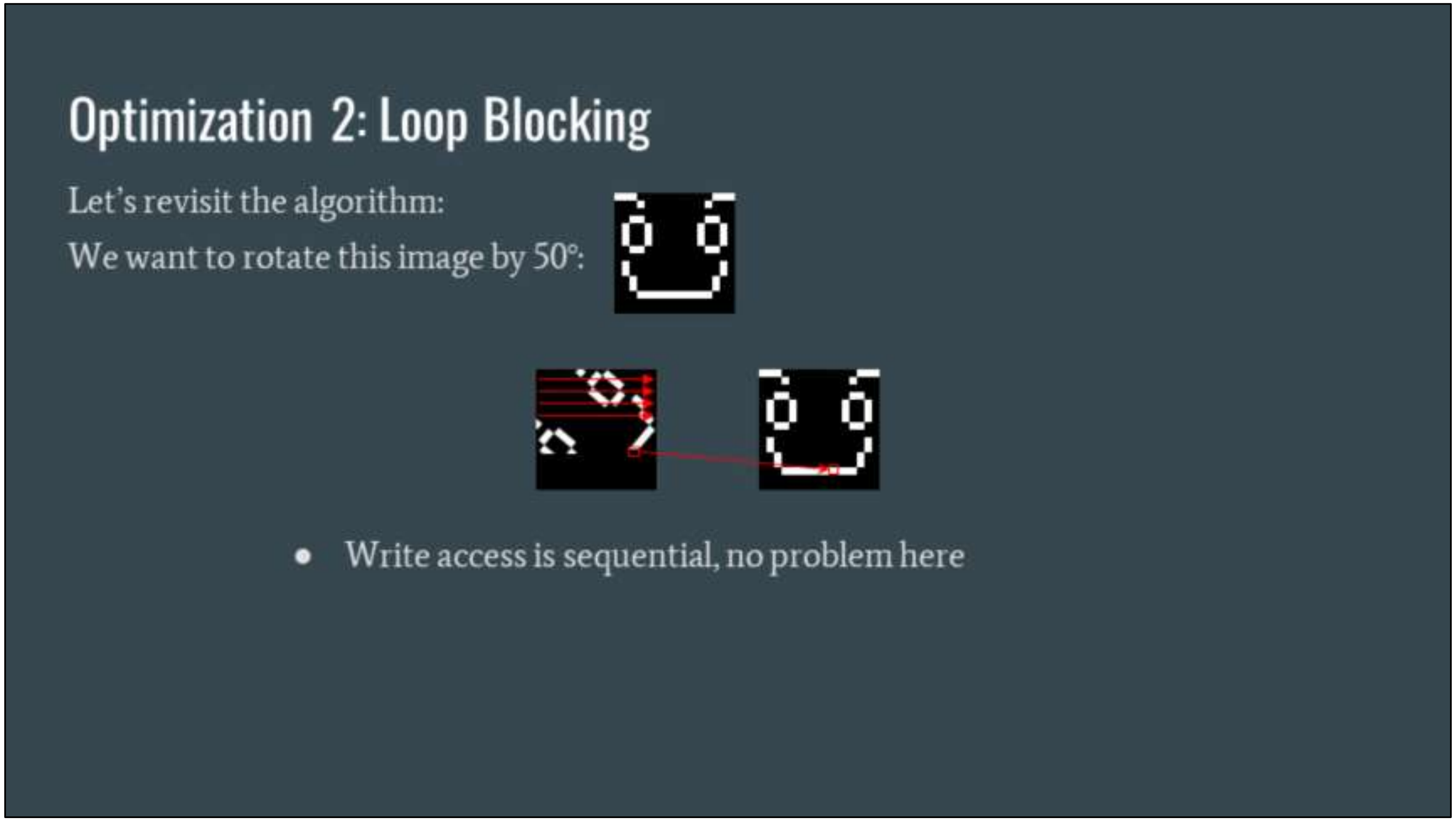


With all this in mind, let's revisit our algorithm to answer the question "why are we memory bound and have so many LLC misses?"

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

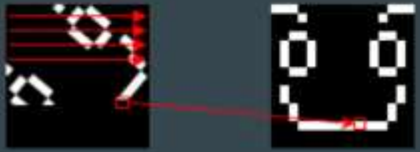
The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Let's revisit the algorithm:

We want to rotate this image by 50°:



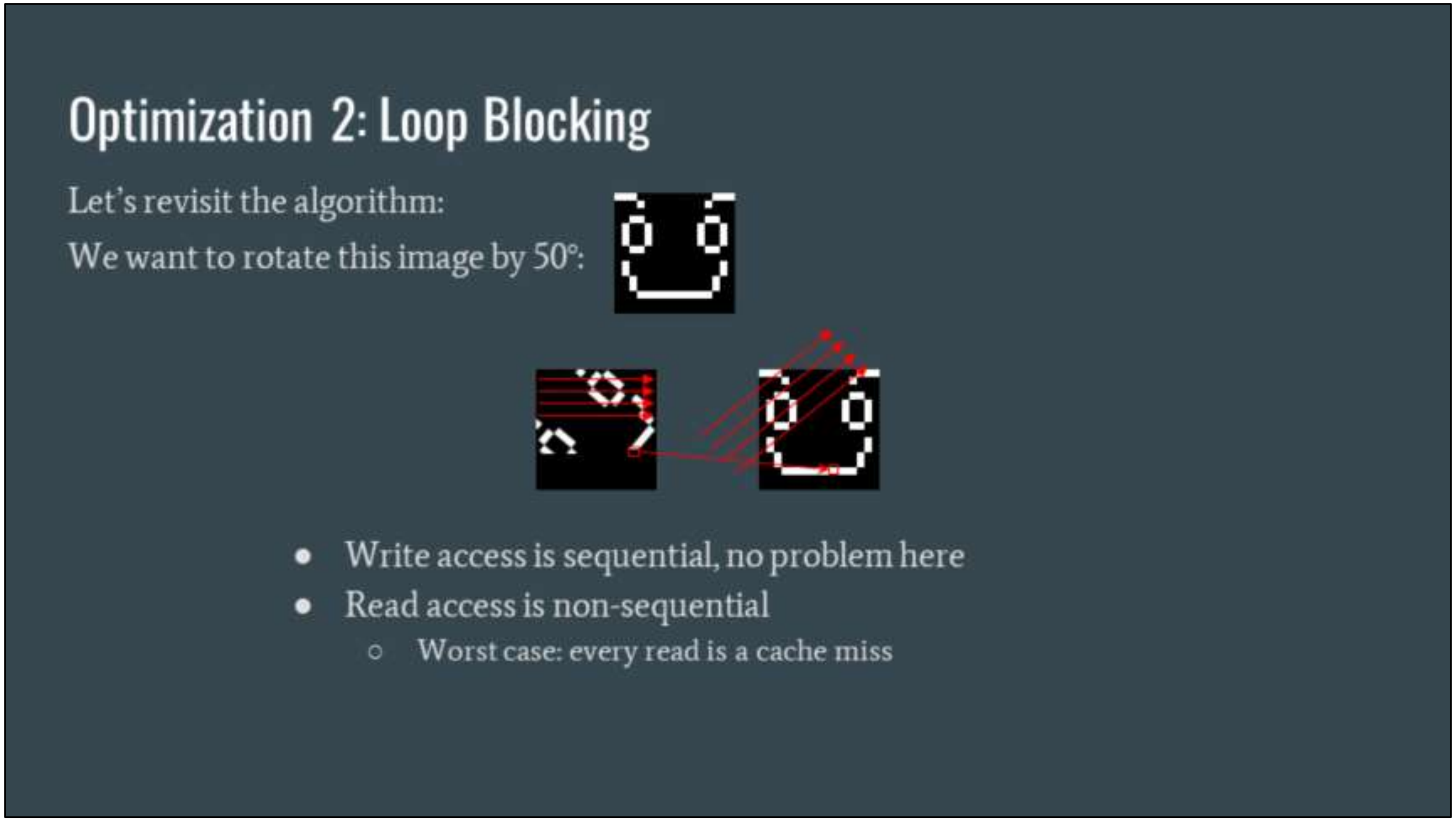
- Write access is sequential, no problem here

With all this in mind, let's revisit our algorithm to answer the question “why are we memory bound and have so many LLC misses?”

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

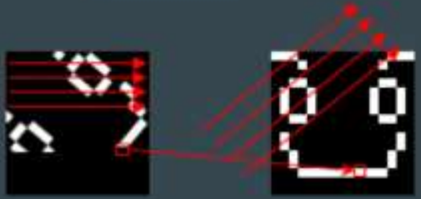
The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Let's revisit the algorithm:

We want to rotate this image by 50°:



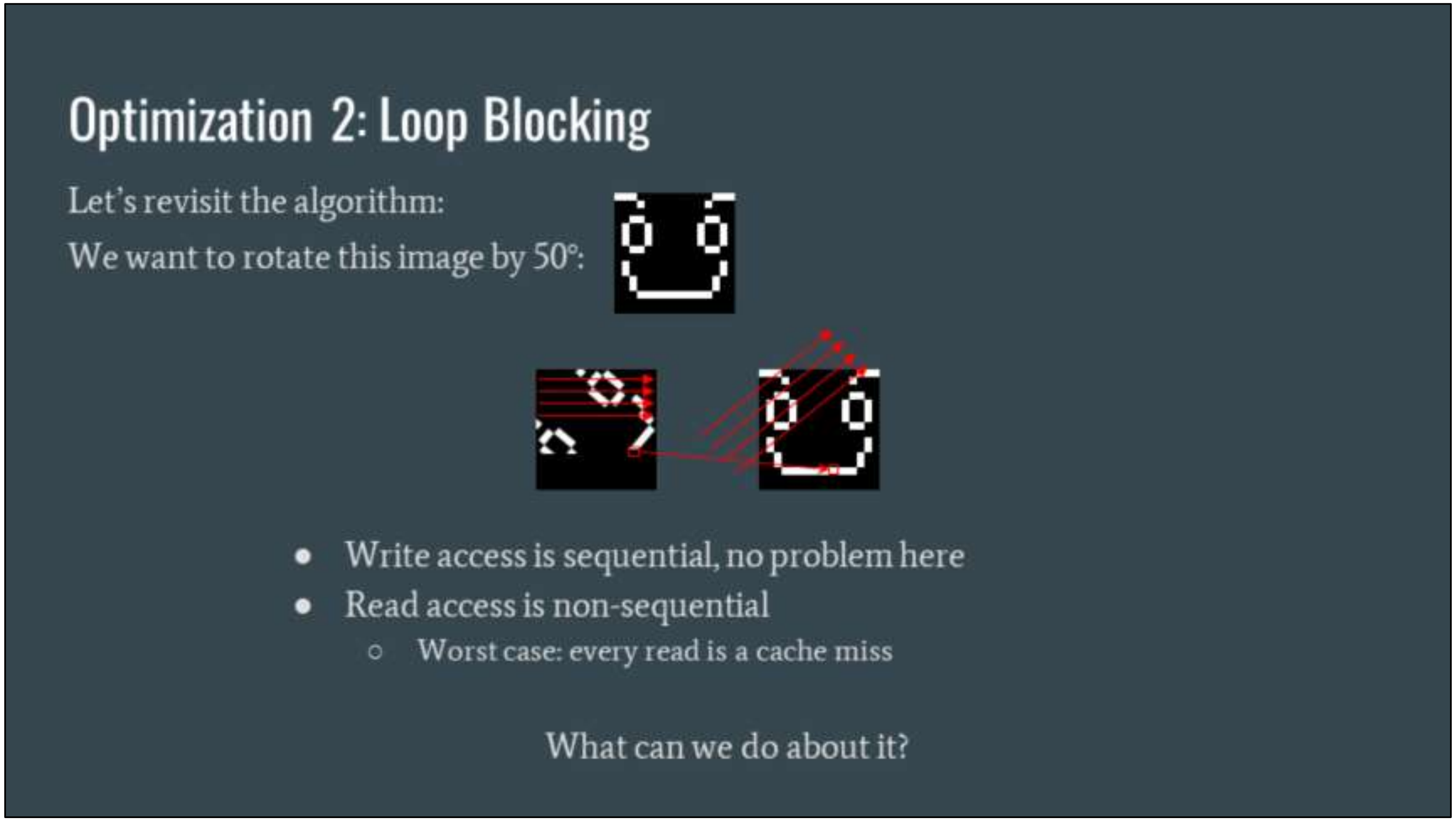
- Write access is sequential, no problem here
- Read access is non-sequential
  - Worst case: every read is a cache miss

With all this in mind, let's revisit our algorithm to answer the question “why are we memory bound and have so many LLC misses?”

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

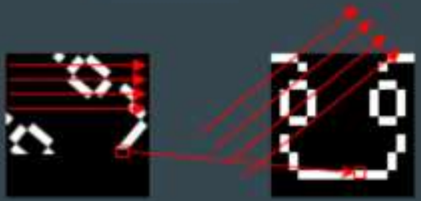
The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Let's revisit the algorithm:

We want to rotate this image by 50°:



- Write access is sequential, no problem here
- Read access is non-sequential
  - Worst case: every read is a cache miss

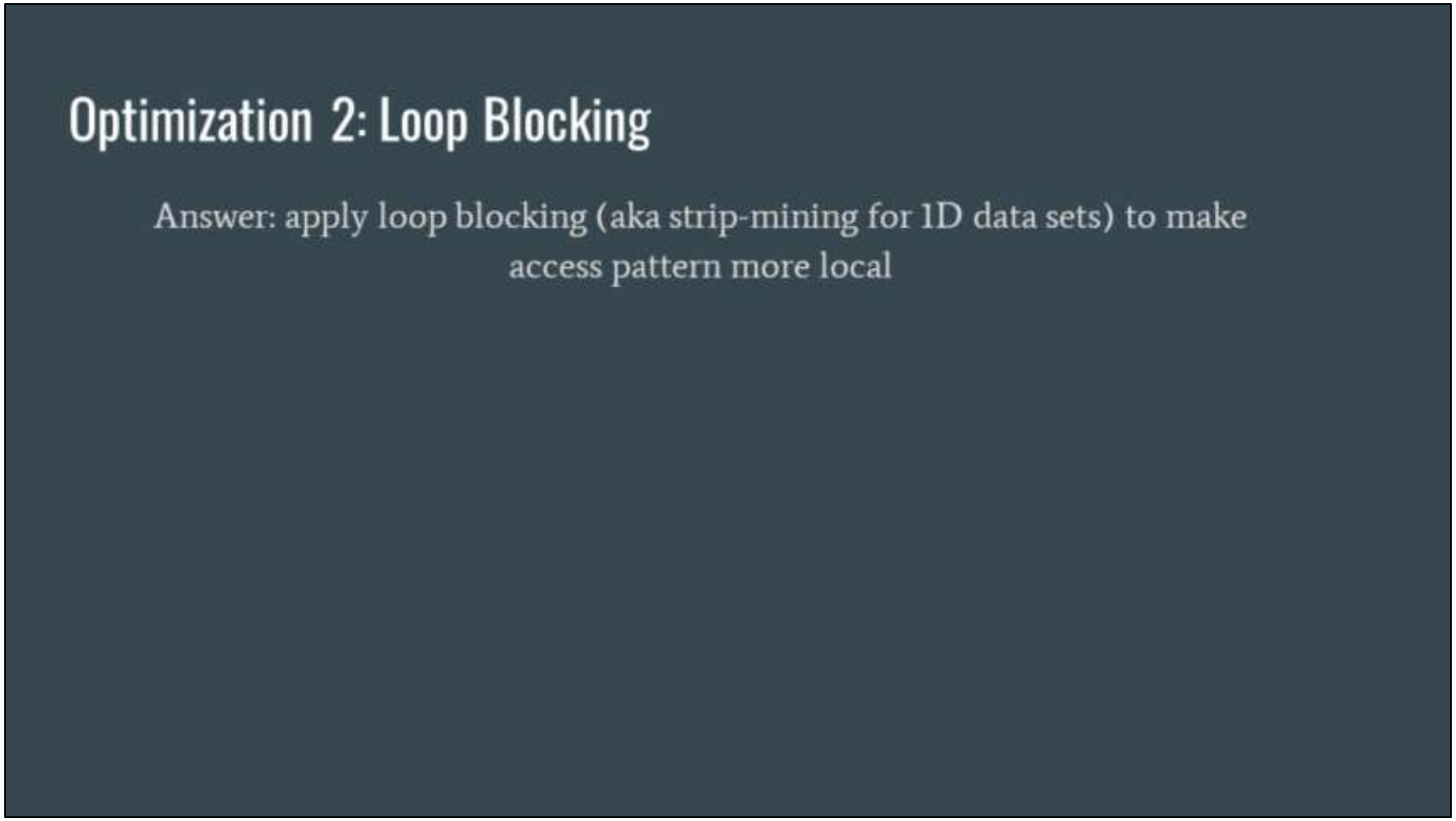
What can we do about it?

With all this in mind, let's revisit our algorithm to answer the question "why are we memory bound and have so many LLC misses?"

Lets say we want to rotate this image by 50°.

Our algorithm works in a way where it iterates sequentially over every pixel of the output image, calculates the index using the transformation matrix, reads the sample at that position and then
Writes the sample back to the output image.

The write access is sequentiel, no problem with this - but the read access is highly non-sequental and could produce a worst case where every read is a cache miss.



Optimization 2: Loop Blocking

Answer: apply loop blocking (aka strip-mining for 1D data sets) to make access pattern more local

The intel architecture optimization reference suggest to apply a loop blocking access pattern to improve memory access locality and reduce the chance of gettings cache misses.
Since our data set is a 2D array of pixels this is called loop blocking, for 1D data the same would be called strip-mining.

Basically all we do is to iterate not over the whole data-set but rather over a fixed-size block within the data-set.

Optimization 2: Loop Blocking

Answer: apply loop blocking (aka strip-mining for 1D data sets) to make access pattern more local

Intel® 64 and IA-32 Architectures Optimization Reference Manual Chapter 5.5.3

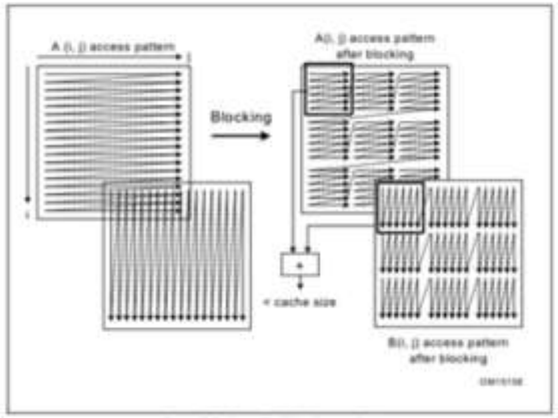
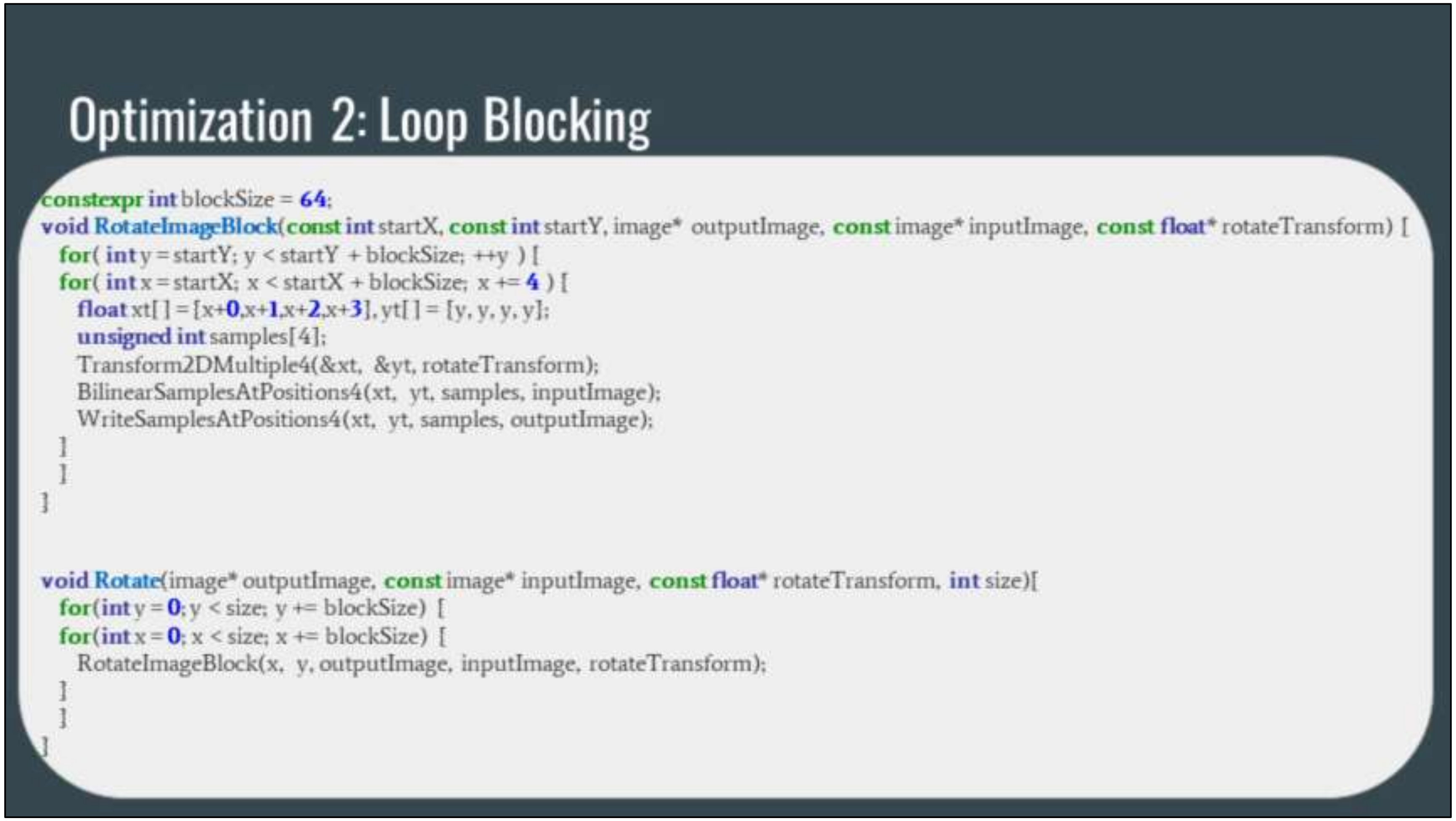

A (i, j) access pattern
A(i, j) access pattern after blocking
Blocking
< cache size
B(i, j) access pattern after blocking

Figure 5-5. Loop Blocking Access Pattern

The intel architecture optimization reference suggest to apply a loop blocking access pattern to improve memory access locality and reduce the chance of gettings cache misses.
Since our data set is a 2D array of pixels this is called loop blocking, for 1D data the same would be called strip-mining.

Basically all we do is to iterate not over the whole data-set but rather over a fixed-size block within the data-set.

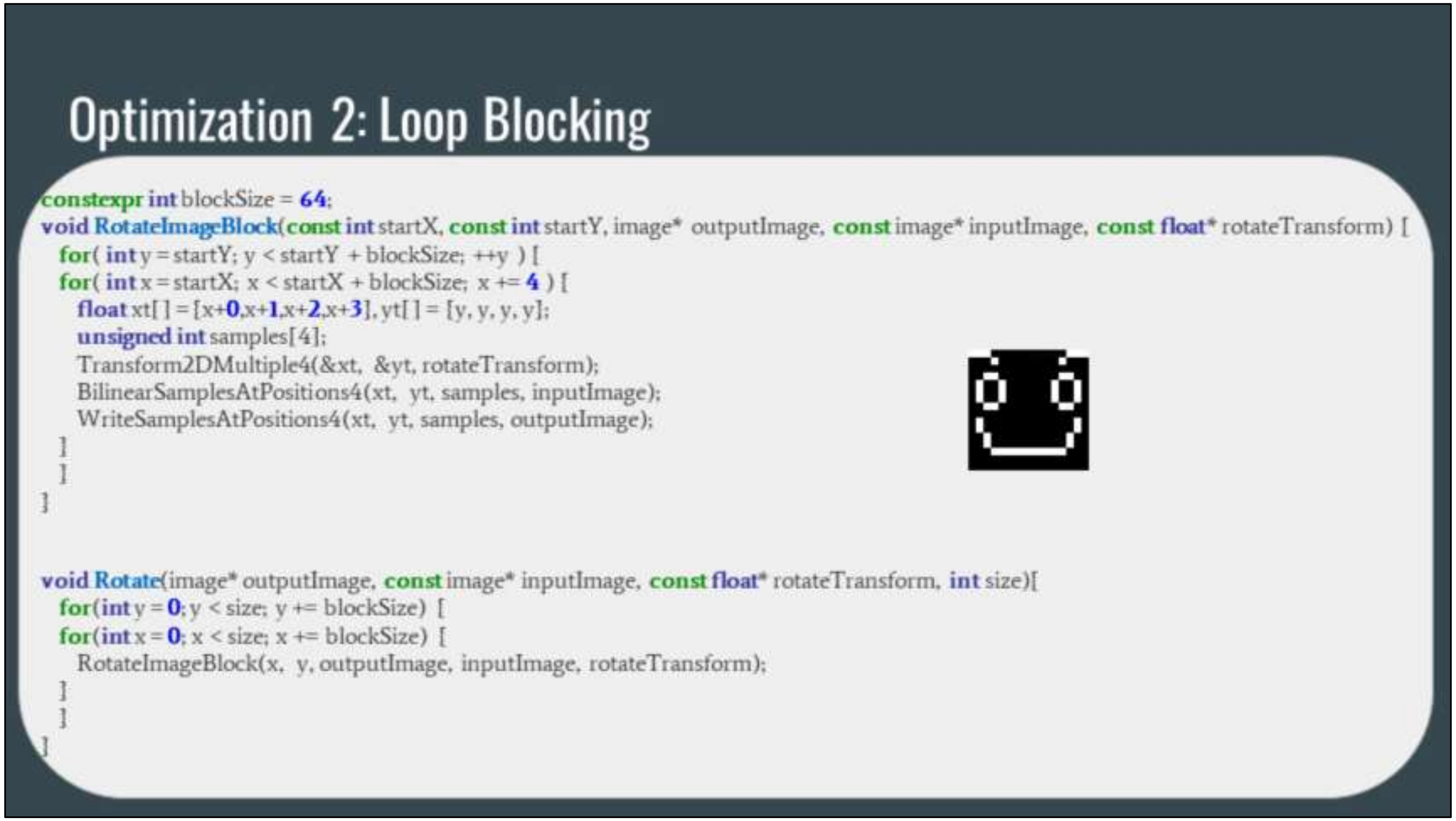


Optimization 2: Loop Blocking

```
constexpr int blockSize = 64;
void RotateImageBlock(const int startX, const int startY, image* outputImage, const image* inputImage, const float* rotateTransform) [
  for( int y = startY; y < startY + blockSize; ++y ) [
  for( int x = startX; x < startX + blockSize; x += 4 ) [
    float xt[] = [x+0,x+1,x+2,x+3], yt[] = [y, y, y, y];
    unsigned int samples[4];
    Transform2DMultiple4(&xt, &yt, rotateTransform);
    BilinearSamplesAtPositions4(xt, yt, samples, inputImage);
    WriteSamplesAtPositions4(xt, yt, samples, outputImage);
  ]
  ]
]


void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform, int size)[
  for(int y = 0; y < size; y += blockSize) [
  for(int x = 0; x < size; x += blockSize) [
    RotateImageBlock(x, y, outputImage, inputImage, rotateTransform);
  ]
  ]
]
```

If we now apply this to our existing code, we'd get something like this.

We basically split the image into 64x64 blocks and then only iterate over the pixels within each block with the already known

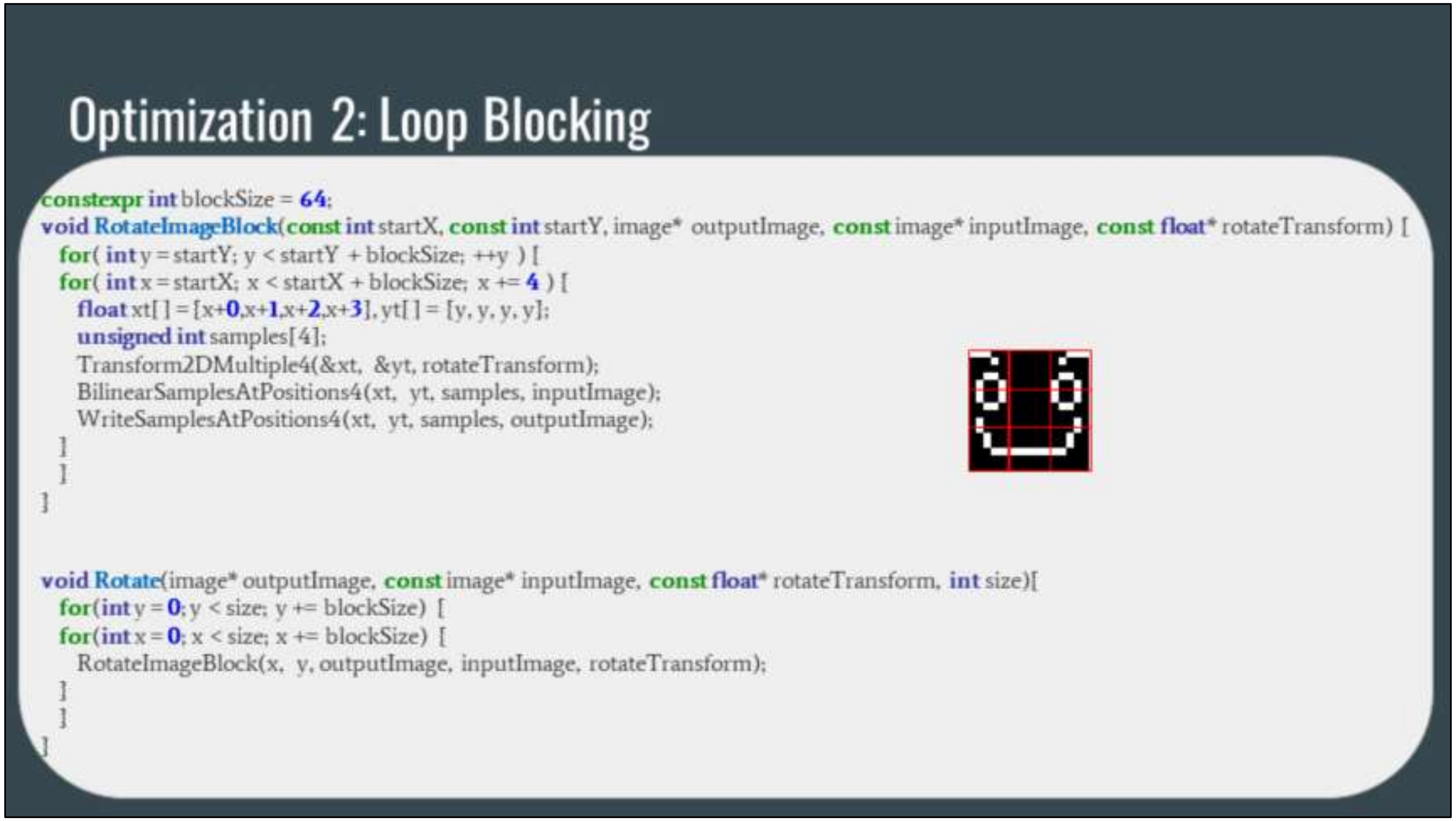


Optimization 2: Loop Blocking

```
constexpr int blockSize = 64;
void RotateImageBlock(const int startX, const int startY, image* outputImage, const image* inputImage, const float* rotateTransform) {
  for( int y = startY; y < startY + blockSize; ++y ) {
  for( int x = startX; x < startX + blockSize; x += 4 ) {
    float xt[] = {x+0,x+1,x+2,x+3}, yt[] = {y, y, y, y};
    unsigned int samples[4];
    Transform2DMultiple4(&xt, &yt, rotateTransform);
    BilinearSamplesAtPositions4(xt, yt, samples, inputImage);
    WriteSamplesAtPositions4(xt, yt, samples, outputImage);
  }
  }
}


void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform, int size){
  for(int y = 0; y < size; y += blockSize) {
  for(int x = 0; x < size; x += blockSize) {
    RotateImageBlock(x, y, outputImage, inputImage, rotateTransform);
  }
  }
}
```

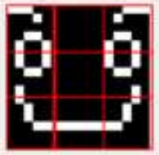
If we now apply this to our existing code, we'd get something like this.

We basically split the image into 64x64 blocks and then only iterate over the pixels within each block with the already known



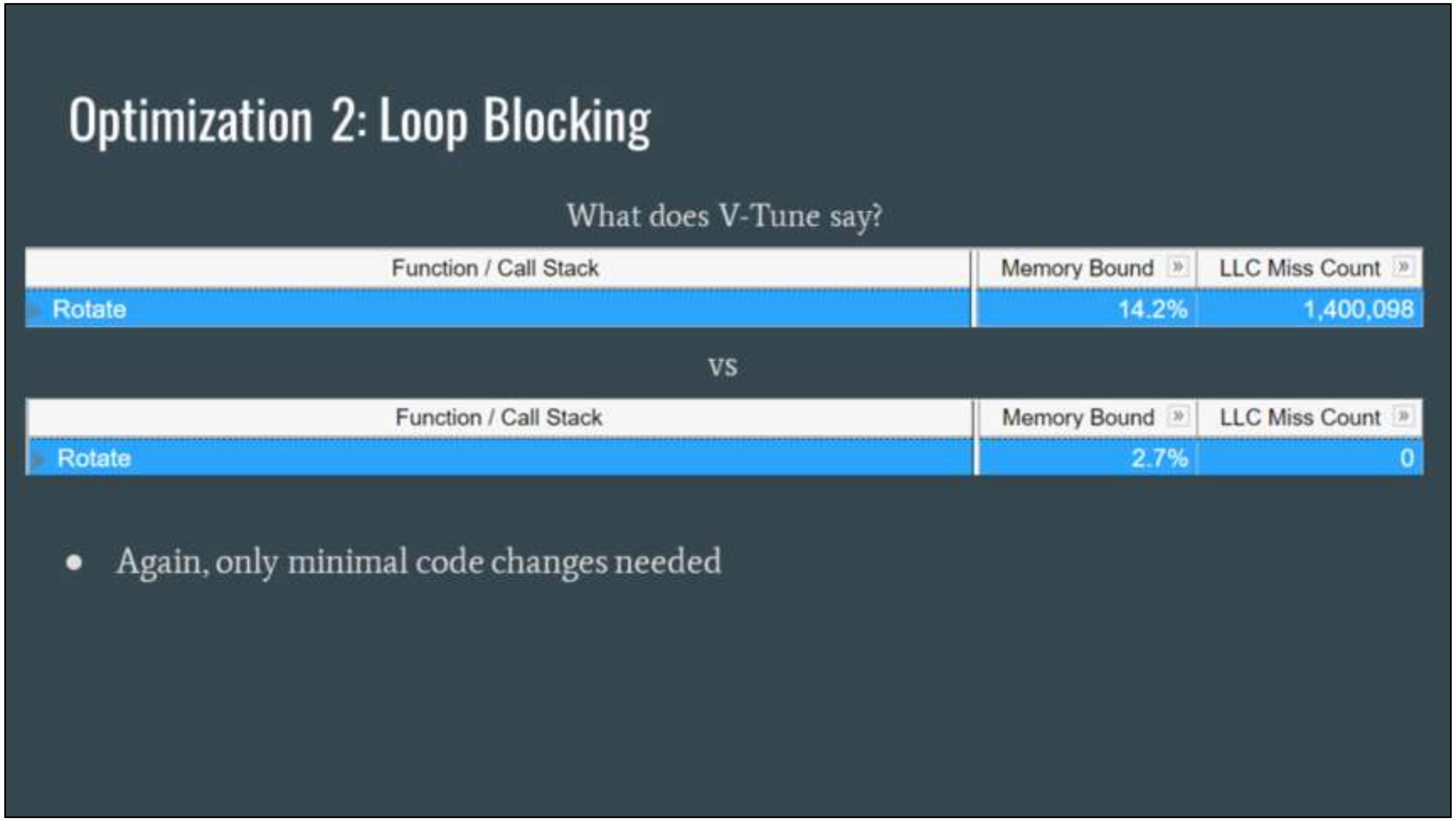
Optimization 2: Loop Blocking

```
constexpr int blockSize = 64;
void RotateImageBlock(const int startX, const int startY, image* outputImage, const image* inputImage, const float* rotateTransform) {
  for( int y = startY; y < startY + blockSize; ++y ) {
  for( int x = startX; x < startX + blockSize; x += 4 ) {
    float xt[] = {x+0,x+1,x+2,x+3}, yt[] = {y, y, y, y};
    unsigned int samples[4];
    Transform2DMultiple4(&xt, &yt, rotateTransform);
    BilinearSamplesAtPositions4(xt, yt, samples, inputImage);
    WriteSamplesAtPositions4(xt, yt, samples, outputImage);
  }
  }
}


void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform, int size){
  for(int y = 0; y < size; y += blockSize) {
  for(int x = 0; x < size; x += blockSize) {
    RotateImageBlock(x, y, outputImage, inputImage, rotateTransform);
  }
  }
}
```



If we now apply this to our existing code, we'd get something like this.

We basically split the image into 64x64 blocks and then only iterate over the pixels within each block with the already known



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

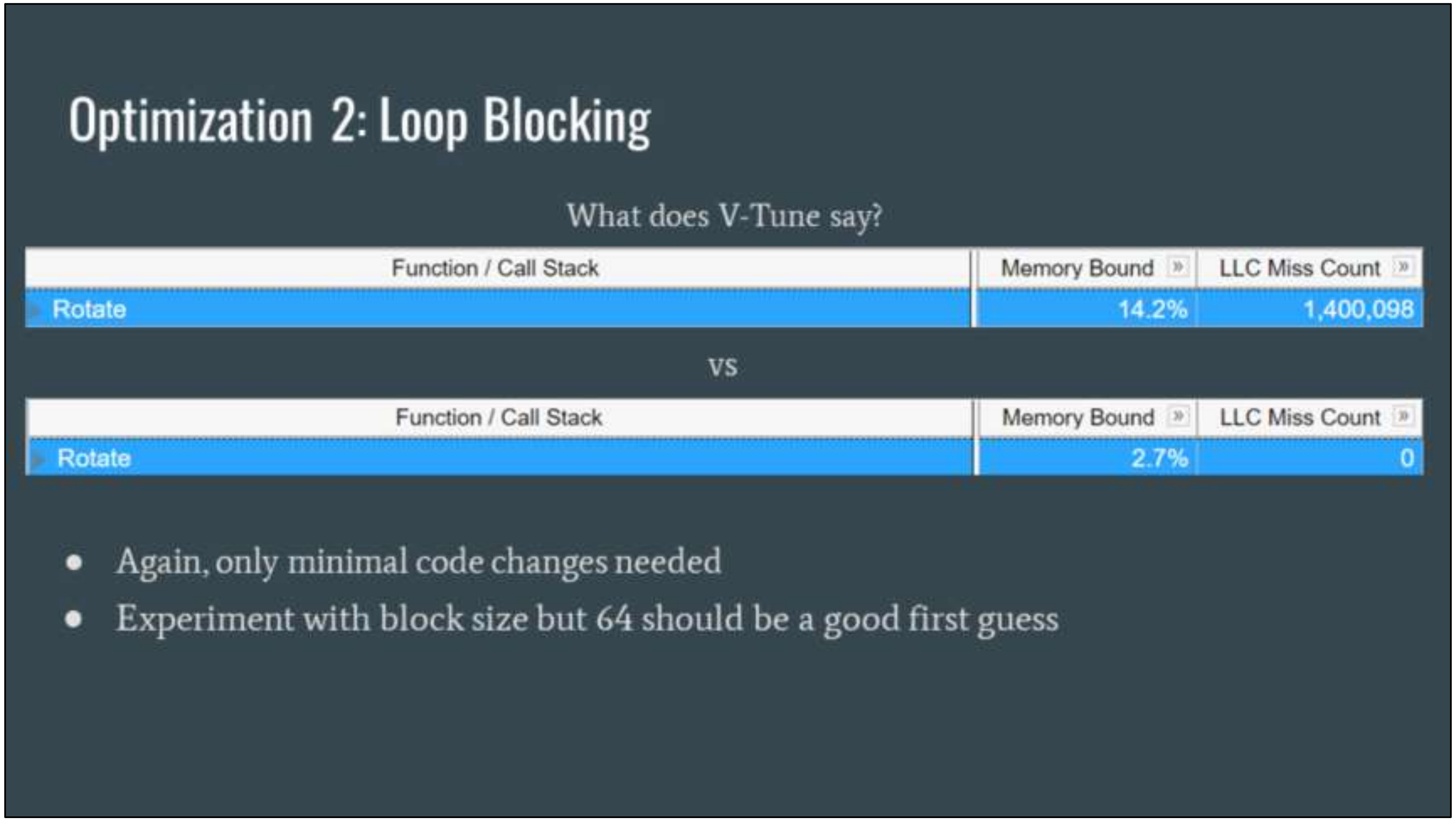
Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

- Again, only minimal code changes needed

The result of this is that we’re not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

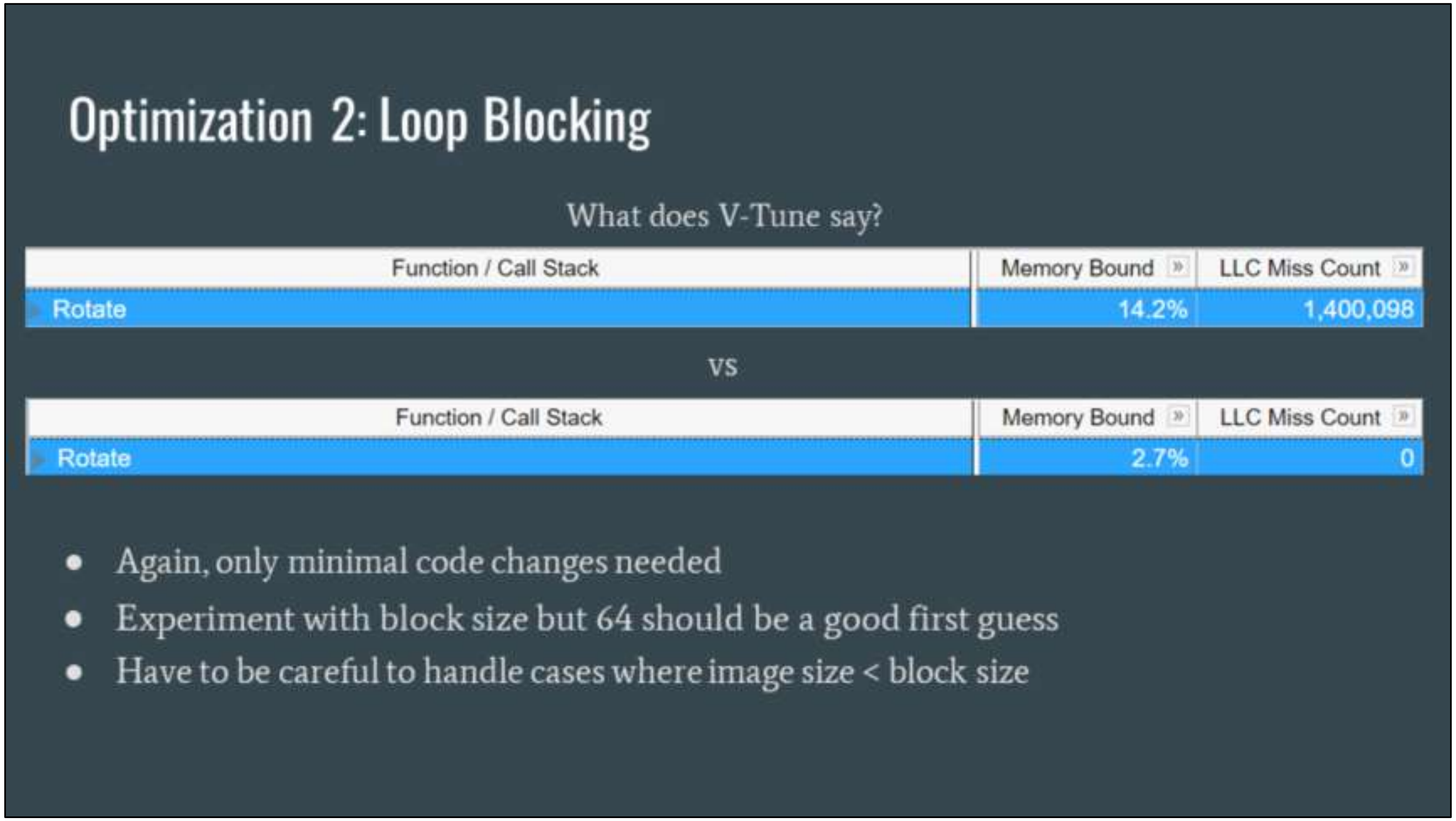
Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

- Again, only minimal code changes needed
- Experiment with block size but 64 should be a good first guess

The result of this is that we're not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

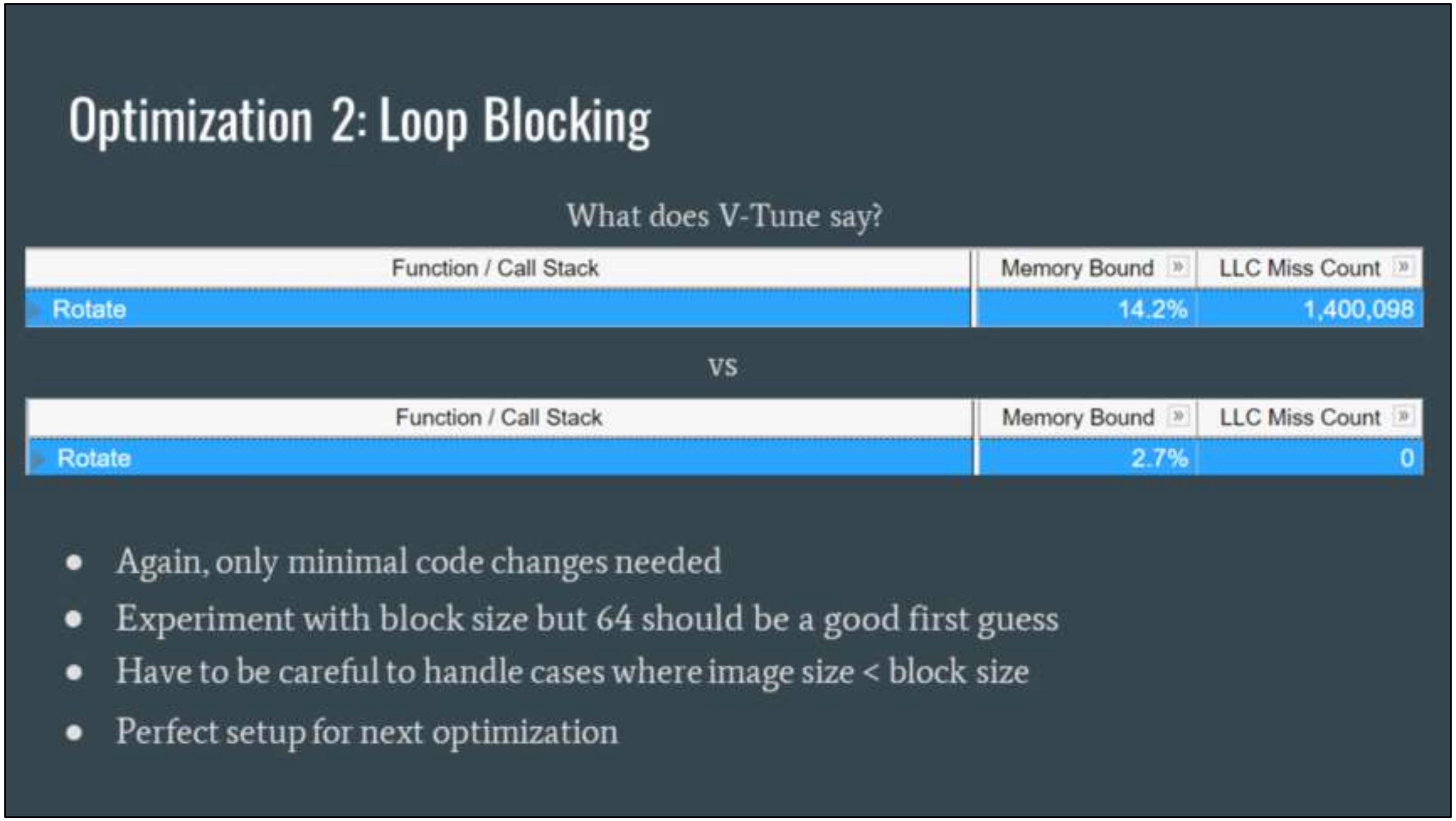
Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

- Again, only minimal code changes needed
- Experiment with block size but 64 should be a good first guess
- Have to be careful to handle cases where image size < block size

The result of this is that we’re not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

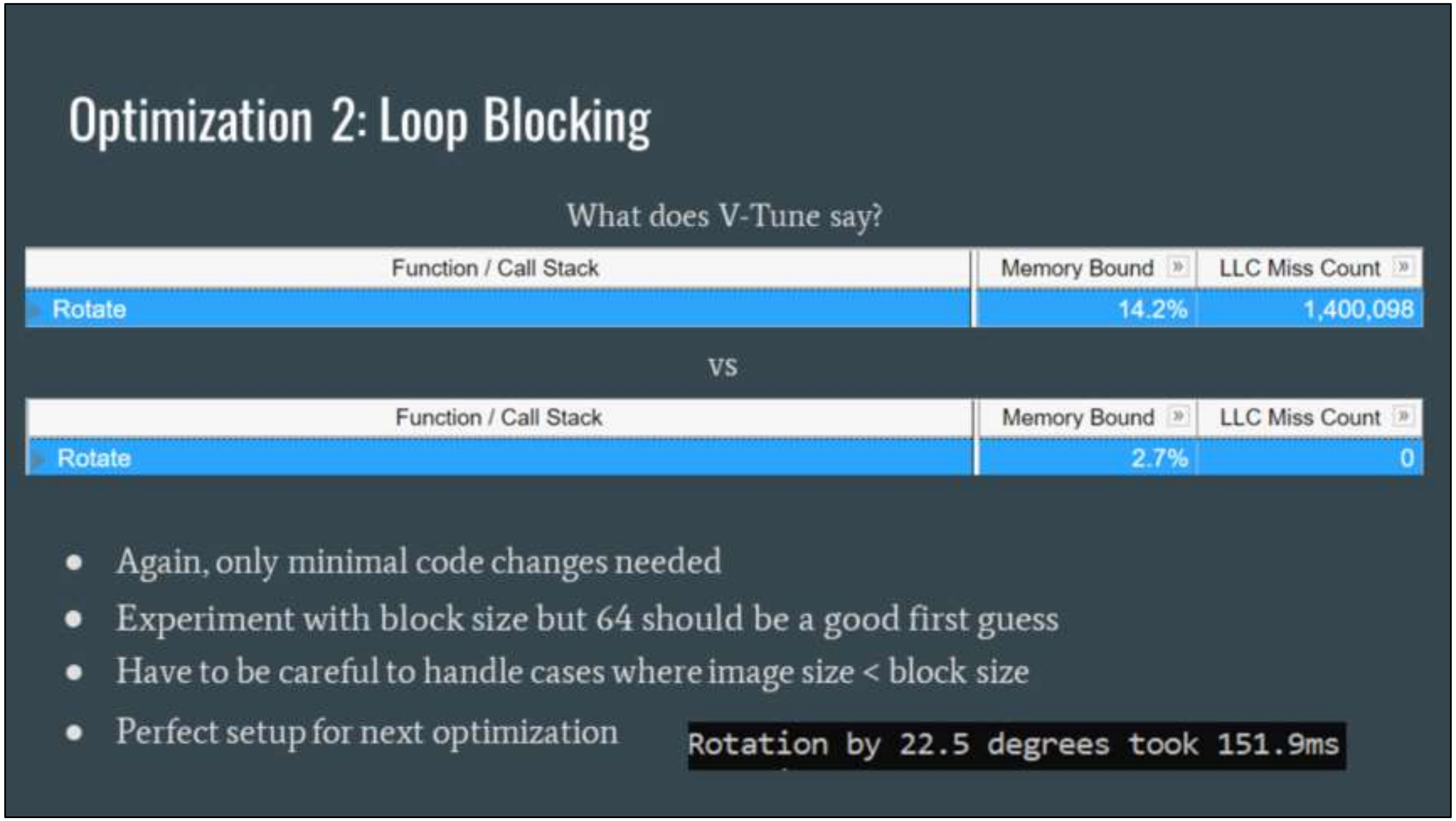
Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

- Again, only minimal code changes needed
- Experiment with block size but 64 should be a good first guess
- Have to be careful to handle cases where image size < block size
- Perfect setup for next optimization

The result of this is that we're not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

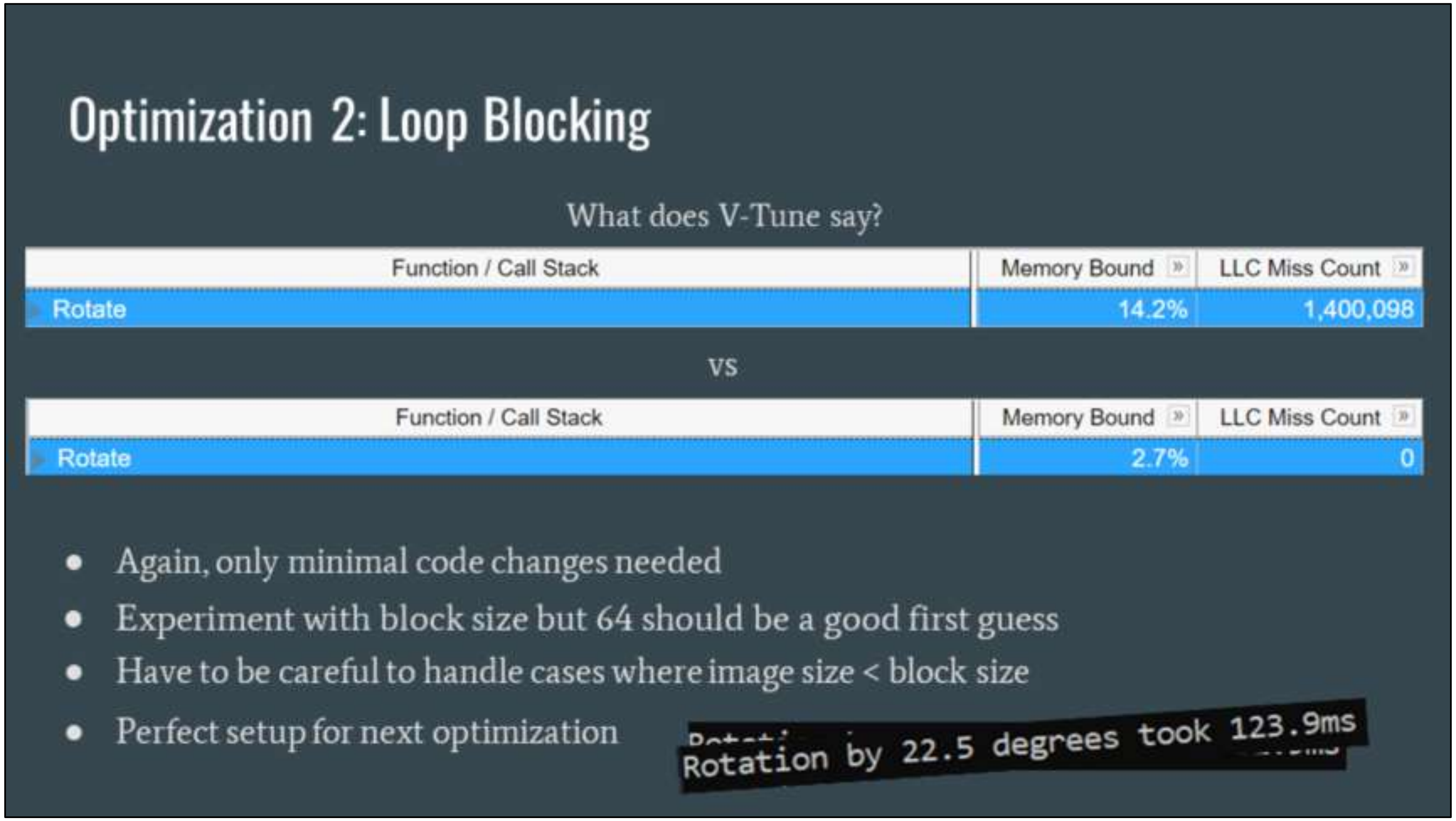
- Again, only minimal code changes needed
- Experiment with block size but 64 should be a good first guess
- Have to be careful to handle cases where image size < block size
- Perfect setup for next optimization

```
Rotation by 22.5 degrees took 151.9ms
```

The result of this is that we’re not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



Optimization 2: Loop Blocking

What does V-Tune say?

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	14.2%	1,400,098

vs

Function / Call Stack	Memory Bound	LLC Miss Count
Rotate	2.7%	0

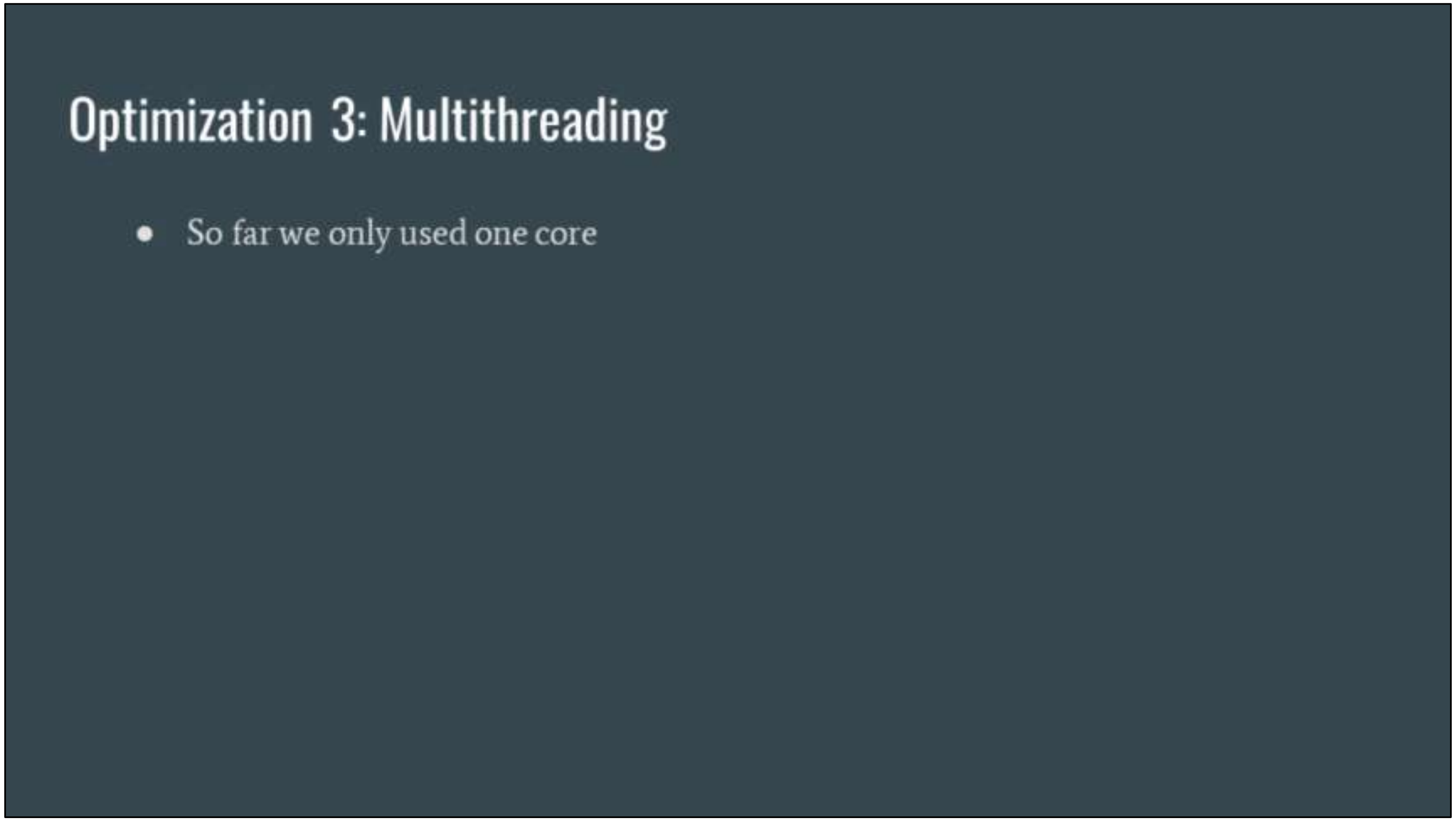
- Again, only minimal code changes needed
- Experiment with block size but 64 should be a good first guess
- Have to be careful to handle cases where image size < block size
- Perfect setup for next optimization

```
Rotation by 22.5 degrees took 123.9ms
```

The result of this is that we're not as heavily memory bound anymore and also completely reduced the LLC misses (which is kind of surprising that there are no cache misses anymore).

This is again an optimization that was done using minimal code changes and nicely builds on top of the loop unrolling.
It might be worth experimenting with different block sizes since the gains depend on the size of the cpu cache and the size of the data set.

Also code must be added to handle cases where the image is smaller then a block.
Doing this optimization improved the runtime by another 30ms.



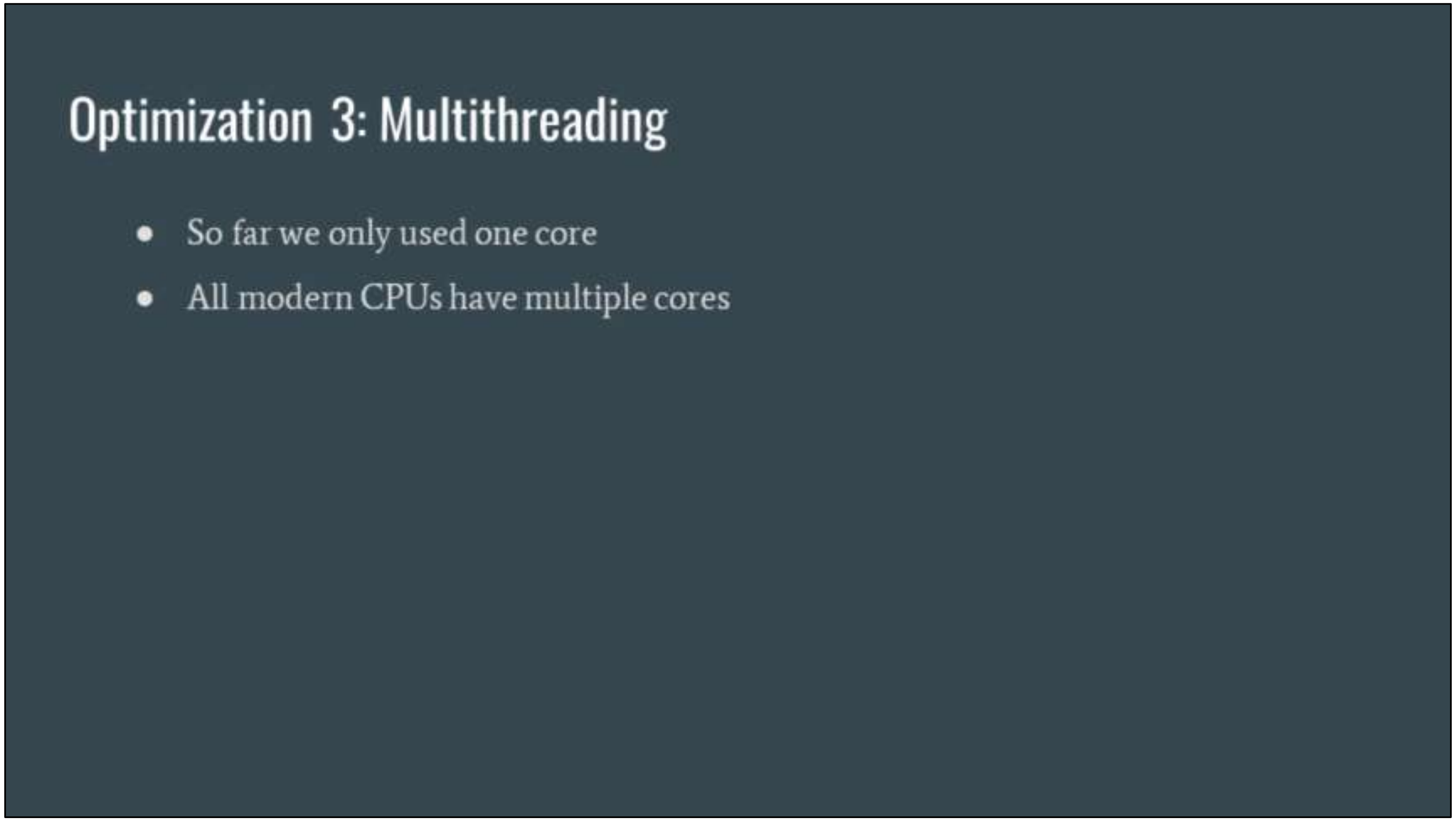
Optimization 3: Multithreading

- So far we only used one core

The next optimization is maybe not super low level but still worth talking about.

All modern CPUs are multi-core processors and so far our code is completely single threaded.
If you intend to work on performance critical software (eg games) it’s important to understand how to utilize all the cores in your CPU.

There are lots of traps to fall into, that we’ll go over in a bit but in general I’d say that it’s better to have multithreading code that is easy to reason about compared to having something that works but nobody quite knows why. This is most likely the code that will blow up during the final pushes of the software and that you definitely will spent much time on to debug and to understand.



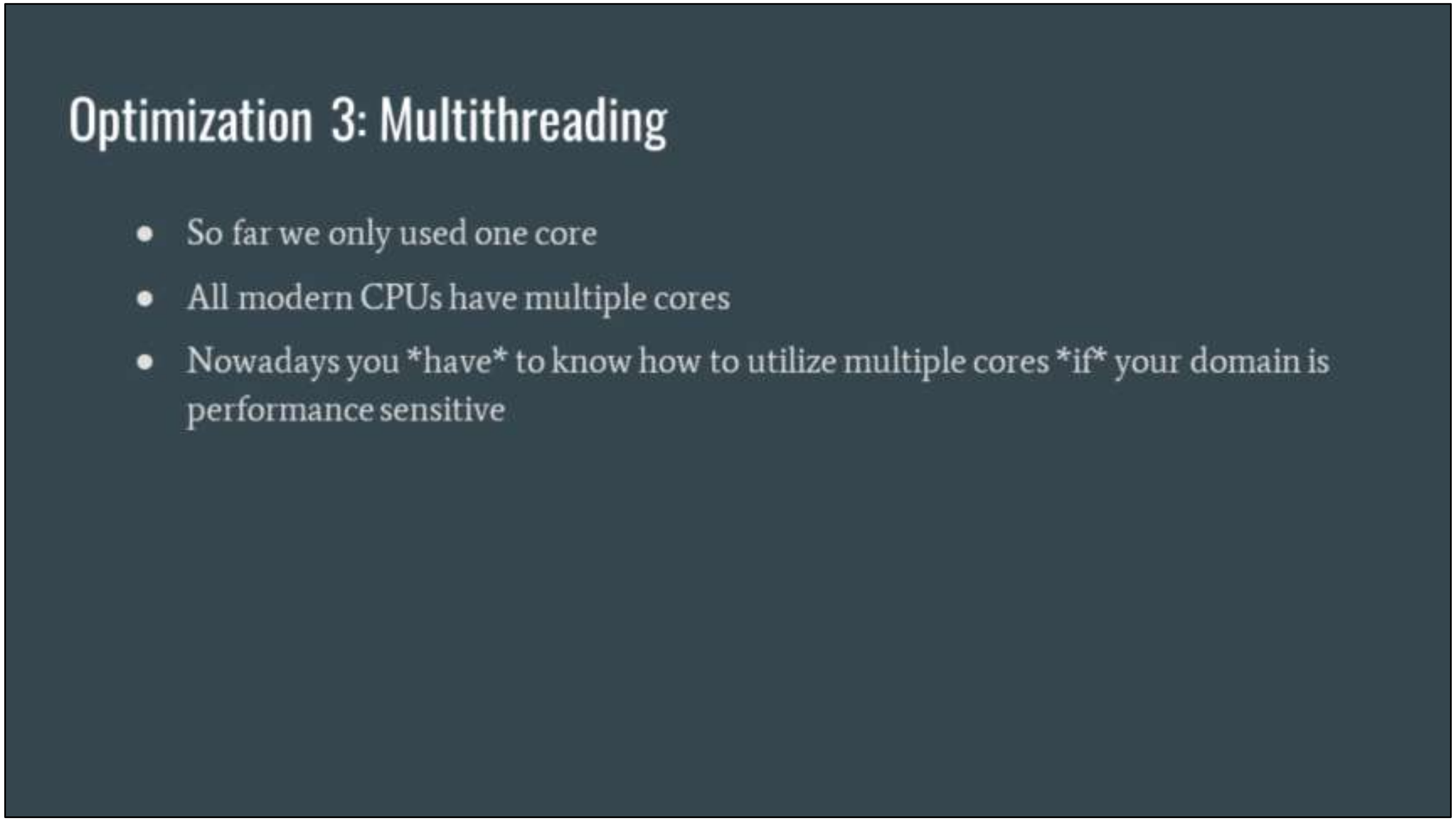
Optimization 3: Multithreading

- So far we only used one core
- All modern CPUs have multiple cores

The next optimization is maybe not super low level but still worth talking about.

All modern CPUs are multi-core processors and so far our code is completely single threaded.
If you intend to work on performance critical software (eg games) it's important to understand how to utilize all the cores in your CPU.

There are lots of traps to fall into, that we'll go over in a bit but in general I'd say that it's better to have multithreading code that is easy to reason about compared to having something that works but nobody quite knows why. This is most likely the code that will blow up during the final pushes of the software and that you definitely will spent much time on to debug and to understand.



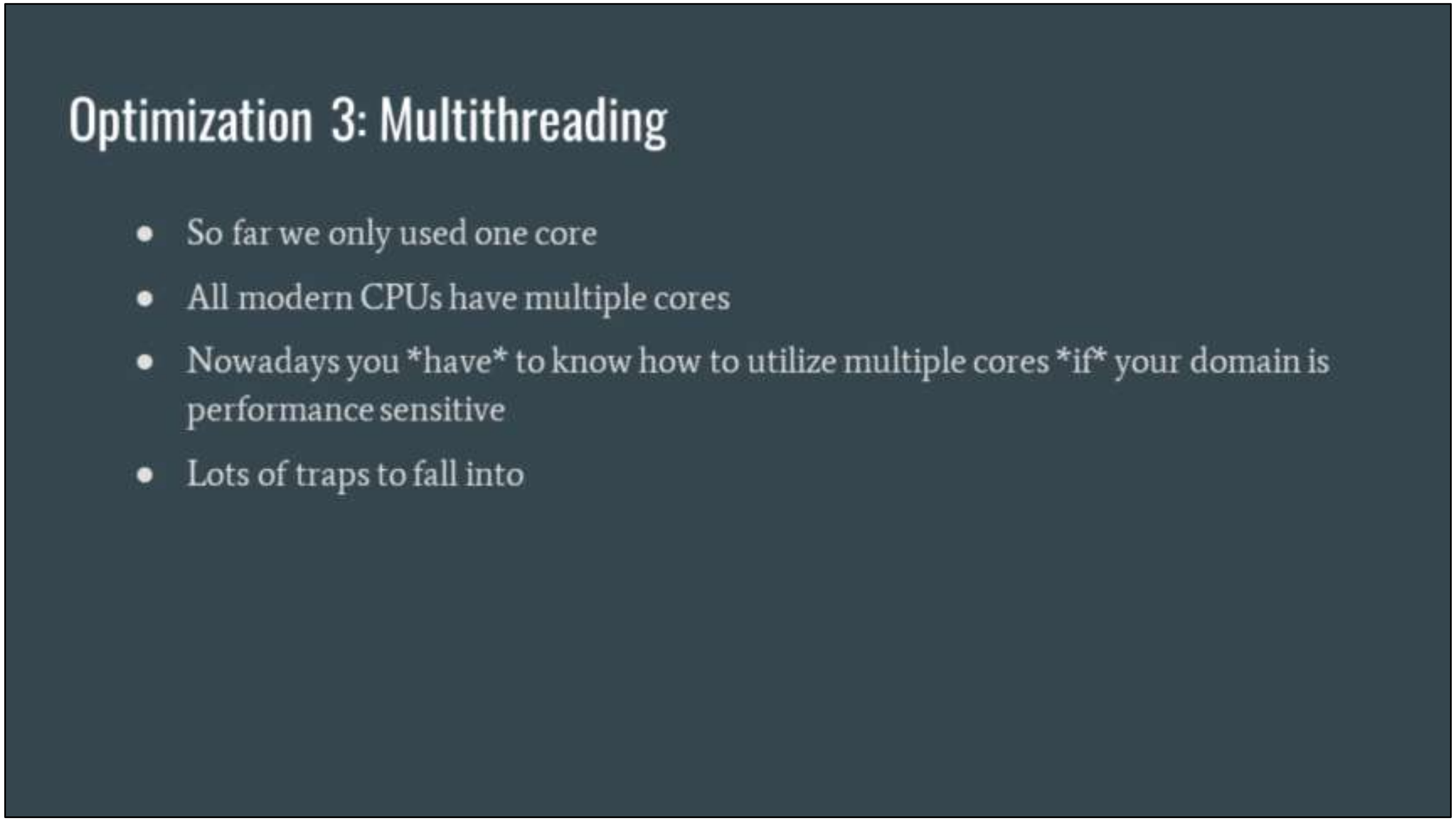
Optimization 3: Multithreading

- So far we only used one core
- All modern CPUs have multiple cores
- Nowadays you *have* to know how to utilize multiple cores *if* your domain is performance sensitive

The next optimization is maybe not super low level but still worth talking about.

All modern CPUs are multi-core processors and so far our code is completely single threaded.
If you intend to work on performance critical software (eg games) it’s important to understand how to utilize all the cores in your CPU.

There are lots of traps to fall into, that we’ll go over in a bit but in general I’d say that it’s better to have multithreading code that is easy to reason about compared to having something that works but nobody quite knows why. This is most likely the code that will blow up during the final pushes of the software and that you definitely will spent much time on to debug and to understand.



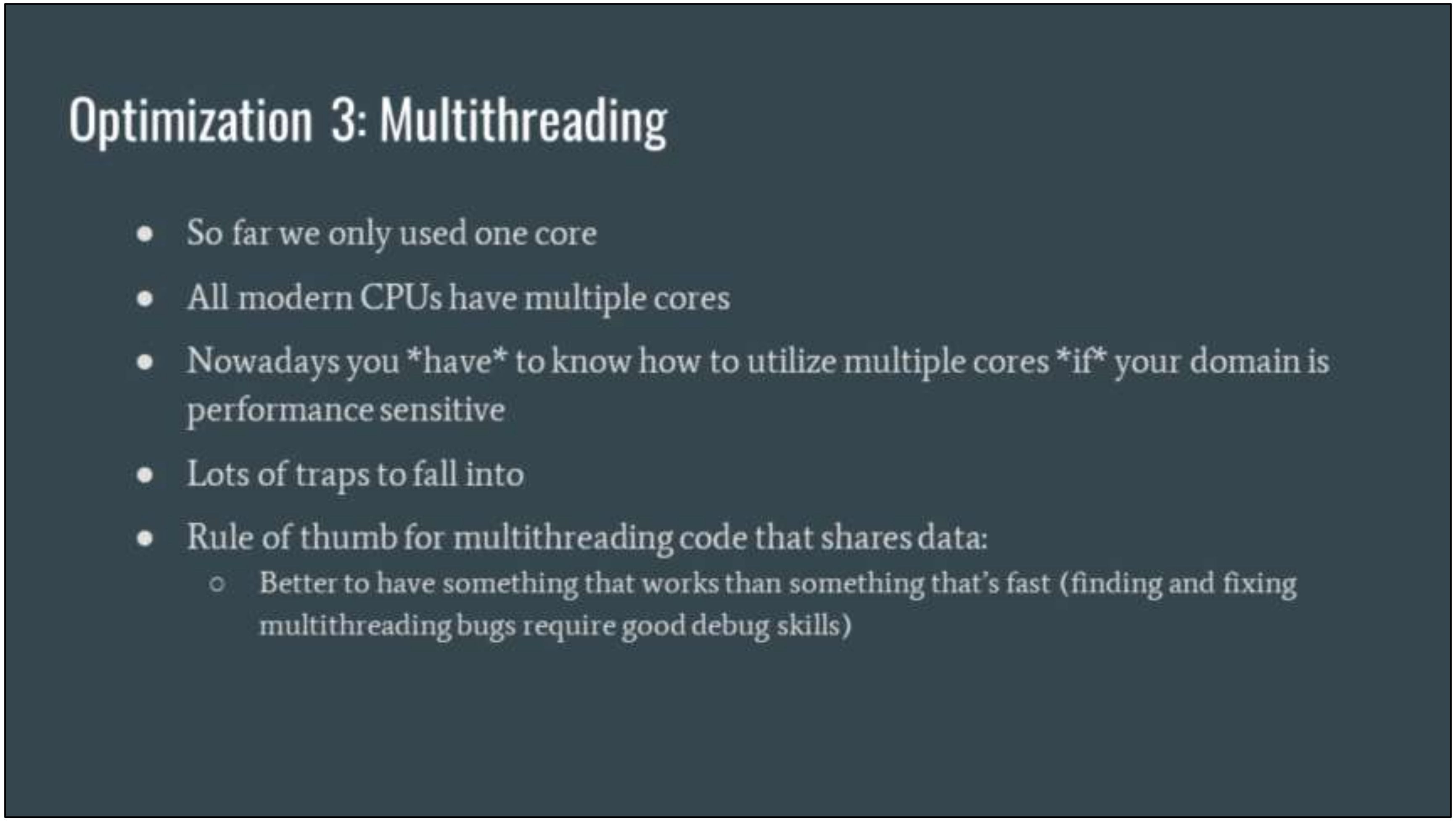
Optimization 3: Multithreading

- So far we only used one core
- All modern CPUs have multiple cores
- Nowadays you *have* to know how to utilize multiple cores *if* your domain is performance sensitive
- Lots of traps to fall into

The next optimization is maybe not super low level but still worth talking about.

All modern CPUs are multi-core processors and so far our code is completely single threaded.
If you intend to work on performance critical software (eg games) it’s important to understand how to utilize all the cores in your CPU.

There are lots of traps to fall into, that we’ll go over in a bit but in general I’d say that it’s better to have multithreading code that is easy to reason about compared to having something that works but nobody quite knows why. This is most likely the code that will blow up during the final pushes of the software and that you definitely will spent much time on to debug and to understand.



Optimization 3: Multithreading

- So far we only used one core
- All modern CPUs have multiple cores
- Nowadays you *have* to know how to utilize multiple cores *if* your domain is performance sensitive
- Lots of traps to fall into
- Rule of thumb for multithreading code that shares data:
  - Better to have something that works than something that’s fast (finding and fixing multithreading bugs require good debug skills)

The next optimization is maybe not super low level but still worth talking about.

All modern CPUs are multi-core processors and so far our code is completely single threaded.
If you intend to work on performance critical software (eg games) it’s important to understand how to utilize all the cores in your CPU.

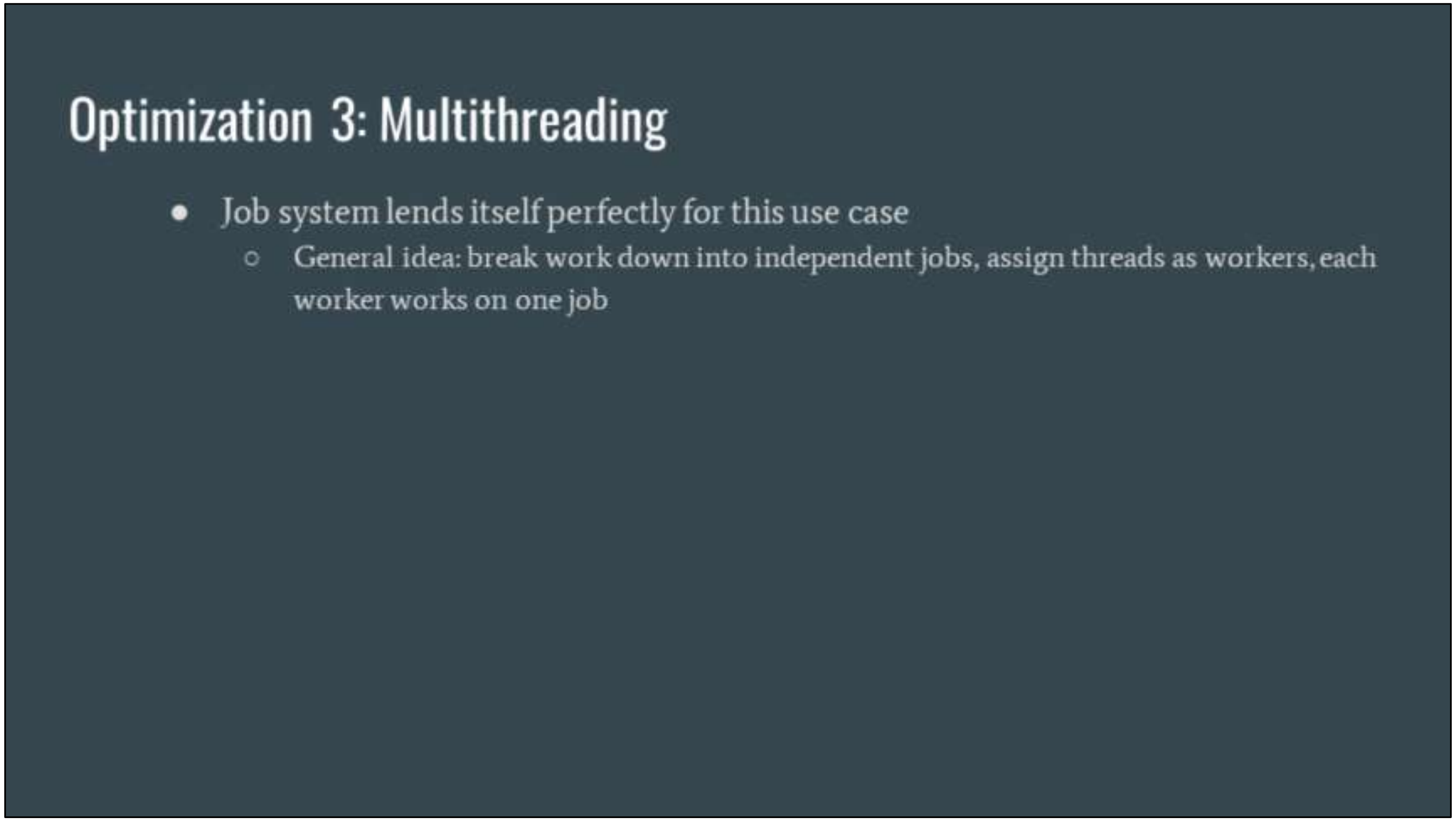
There are lots of traps to fall into, that we’ll go over in a bit but in general I’d say that it’s better to have multithreading code that is easy to reason about compared to having something that works but nobody quite knows why. This is most likely the code that will blow up during the final pushes of the software and that you definitely will spent much time on to debug and to understand.

Optimization 3: Multithreading

The concept of a job system lends itself perfectly for our use case.

A job is defined as an independent piece of work that is consumed by something known as workers.
Workers are mostly 1 or more threads that work on any given job in isolation.

Generally a job system is implemented as a single producer, multiple consumer concept where the main thread is the producer (produces jobs) and the worker are the consumer (consuming jobs). If the main thread is idle, it can also act as a consumer and take over work that has been assigned to a consumer yet.



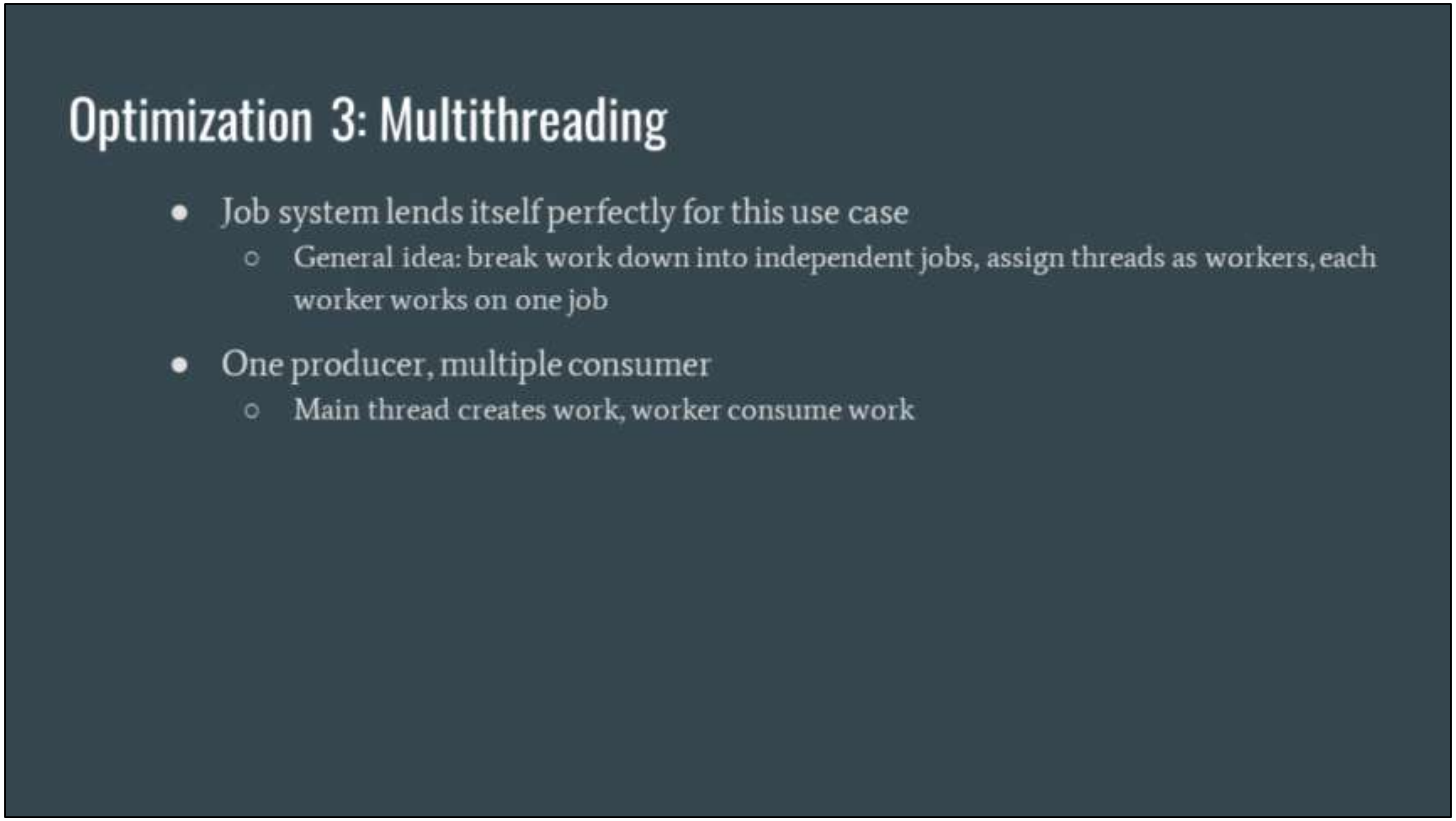
Optimization 3: Multithreading

- Job system lends itself perfectly for this use case
  - General idea: break work down into independent jobs, assign threads as workers, each worker works on one job

The concept of a job system lends itself perfectly for our use case.

A job is defined as an independent piece of work that is consumed by something known as workers.
Workers are mostly 1 or more threads that work on any given job in isolation.

Generally a job system is implemented as a single producer, multiple consumer concept where the main thread is the producer (produces jobs) and the worker are the consumer (consuming jobs). If the main thread is idle, it can also act as a consumer and take over work that has been assigned to a consumer yet.



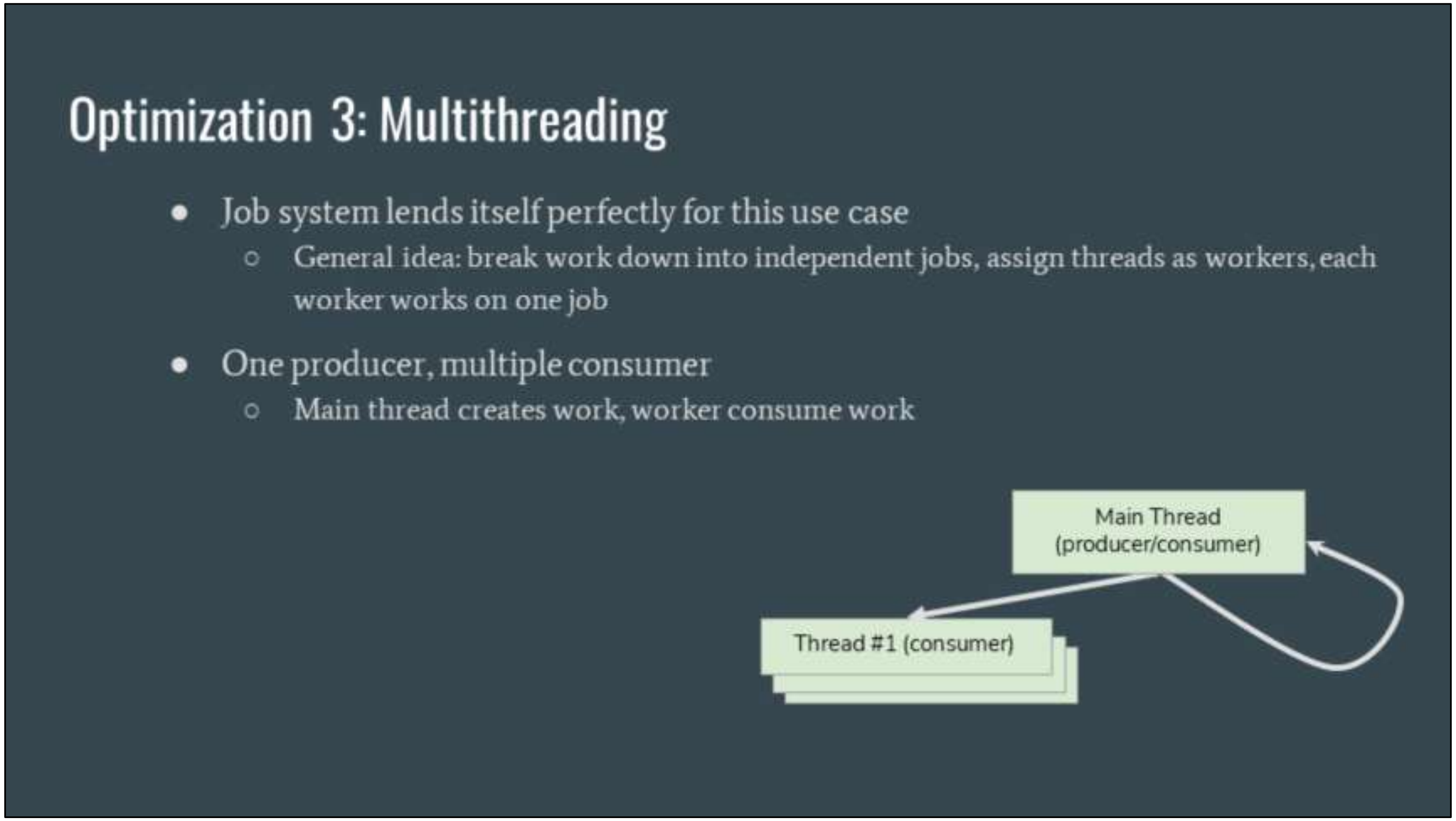
Optimization 3: Multithreading

- Job system lends itself perfectly for this use case
  - General idea: break work down into independent jobs, assign threads as workers, each worker works on one job
- One producer, multiple consumer
  - Main thread creates work, worker consume work

The concept of a job system lends itself perfectly for our use case.

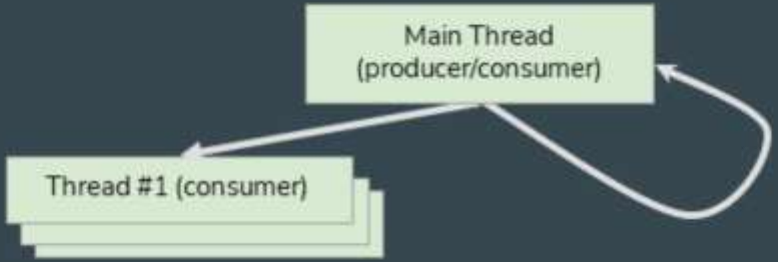
A job is defined as an independent piece of work that is consumed by something known as workers.
Workers are mostly 1 or more threads that work on any given job in isolation.

Generally a job system is implemented as a single producer, multiple consumer concept where the main thread is the producer (produces jobs) and the worker are the consumer (consuming jobs). If the main thread is idle, it can also act as a consumer and take over work that has been assigned to a consumer yet.



Optimization 3: Multithreading

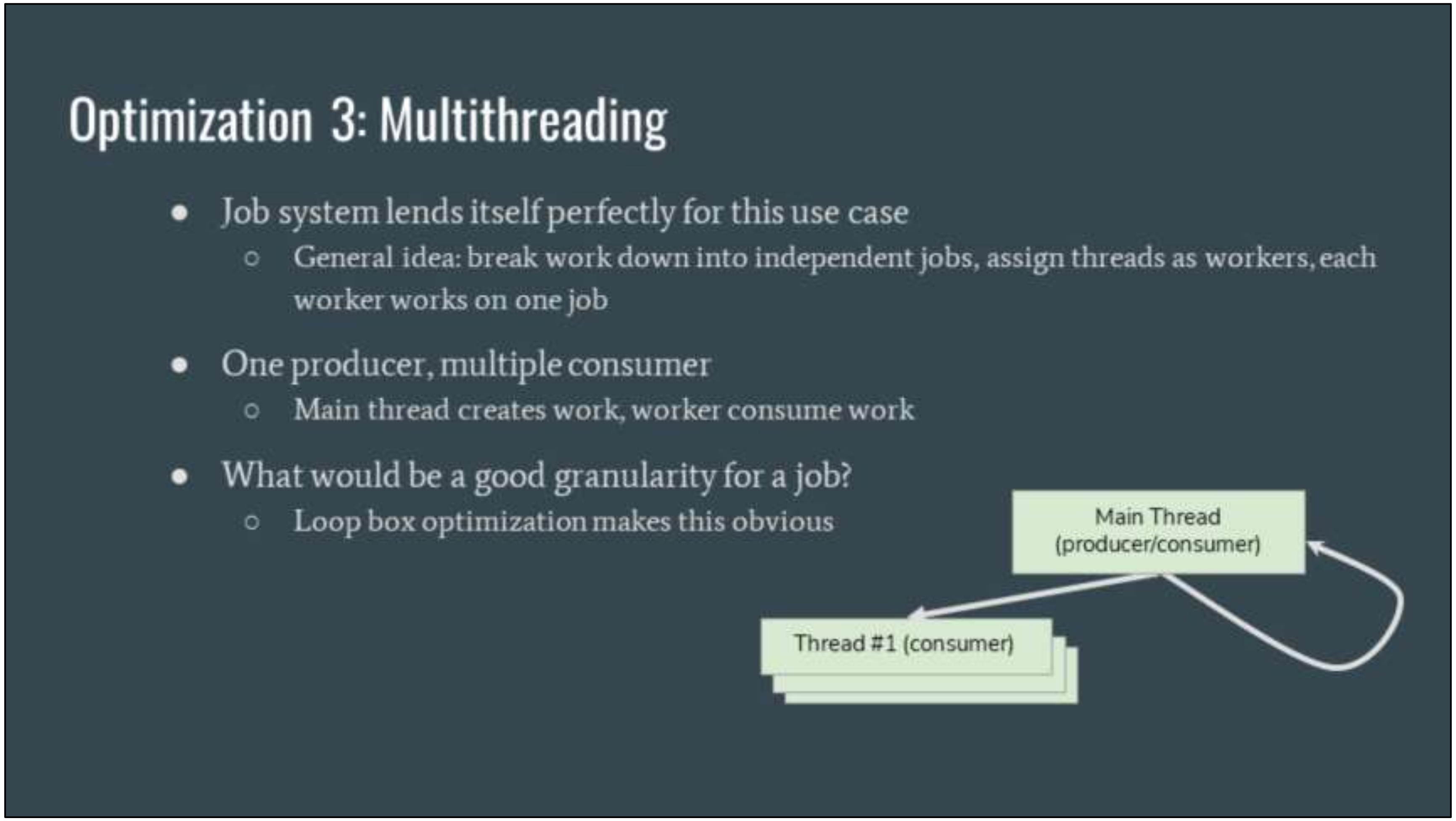
- Job system lends itself perfectly for this use case
  - General idea: break work down into independent jobs, assign threads as workers, each worker works on one job
- One producer, multiple consumer
  - Main thread creates work, worker consume work



The concept of a job system lends itself perfectly for our use case.

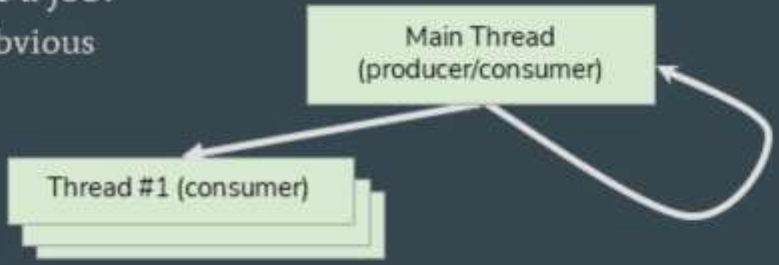
A job is defined as an independent piece of work that is consumed by something known as workers.
Workers are mostly 1 or more threads that work on any given job in isolation.

Generally a job system is implemented as a single producer, multiple consumer concept where the main thread is the producer (produces jobs) and the worker are the consumer (consuming jobs). If the main thread is idle, it can also act as a consumer and take over work that has been assigned to a consumer yet.



Optimization 3: Multithreading

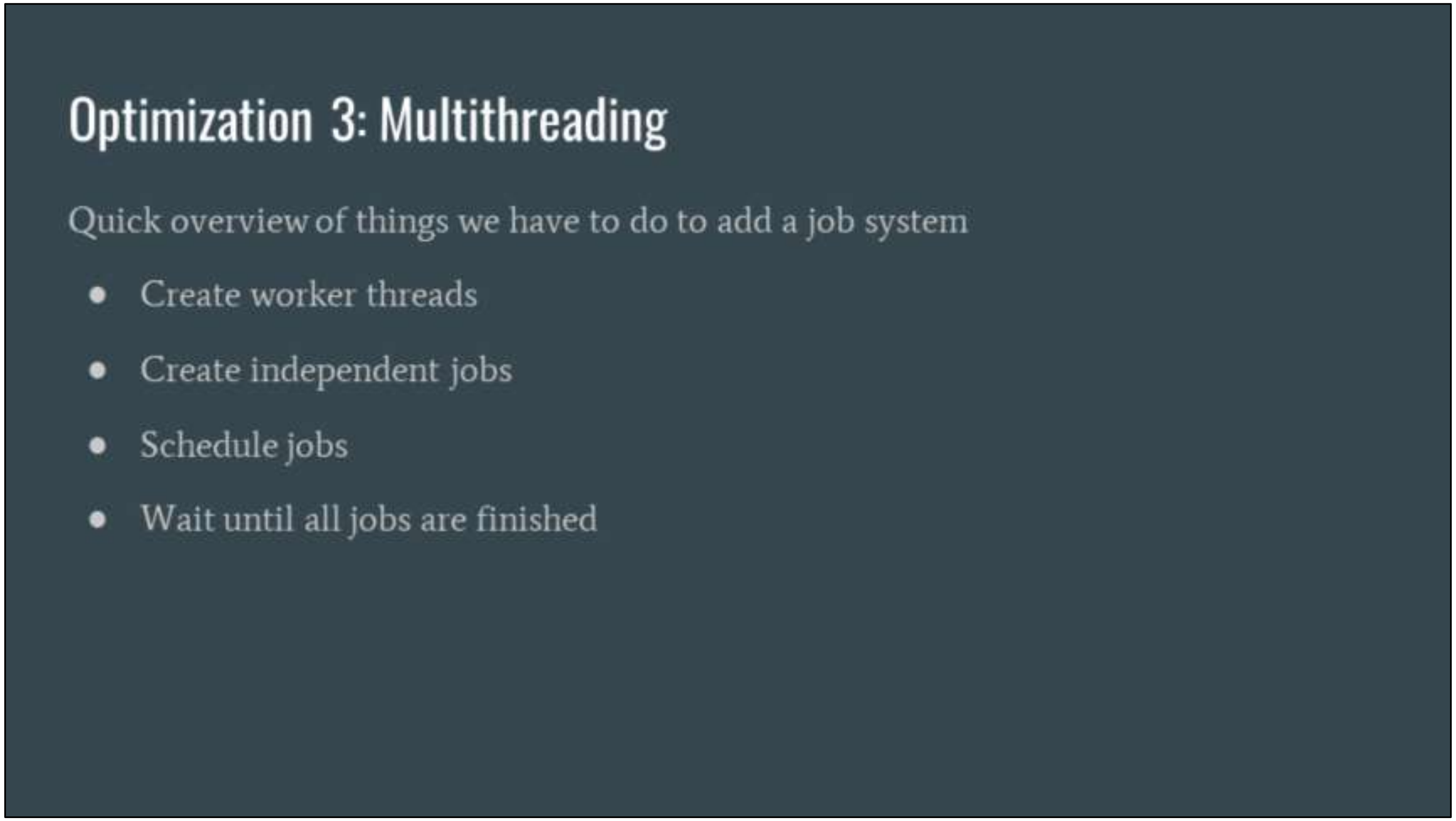
- Job system lends itself perfectly for this use case
  - General idea: break work down into independent jobs, assign threads as workers, each worker works on one job
- One producer, multiple consumer
  - Main thread creates work, worker consume work
- What would be a good granularity for a job?
  - Loop box optimization makes this obvious



The concept of a job system lends itself perfectly for our use case.

A job is defined as an independent piece of work that is consumed by something known as workers.
Workers are mostly 1 or more threads that work on any given job in isolation.

Generally a job system is implemented as a single producer, multiple consumer concept where the main thread is the producer (produces jobs) and the worker are the consumer (consuming jobs). If the main thread is idle, it can also act as a consumer and take over work that has been assigned to a consumer yet.



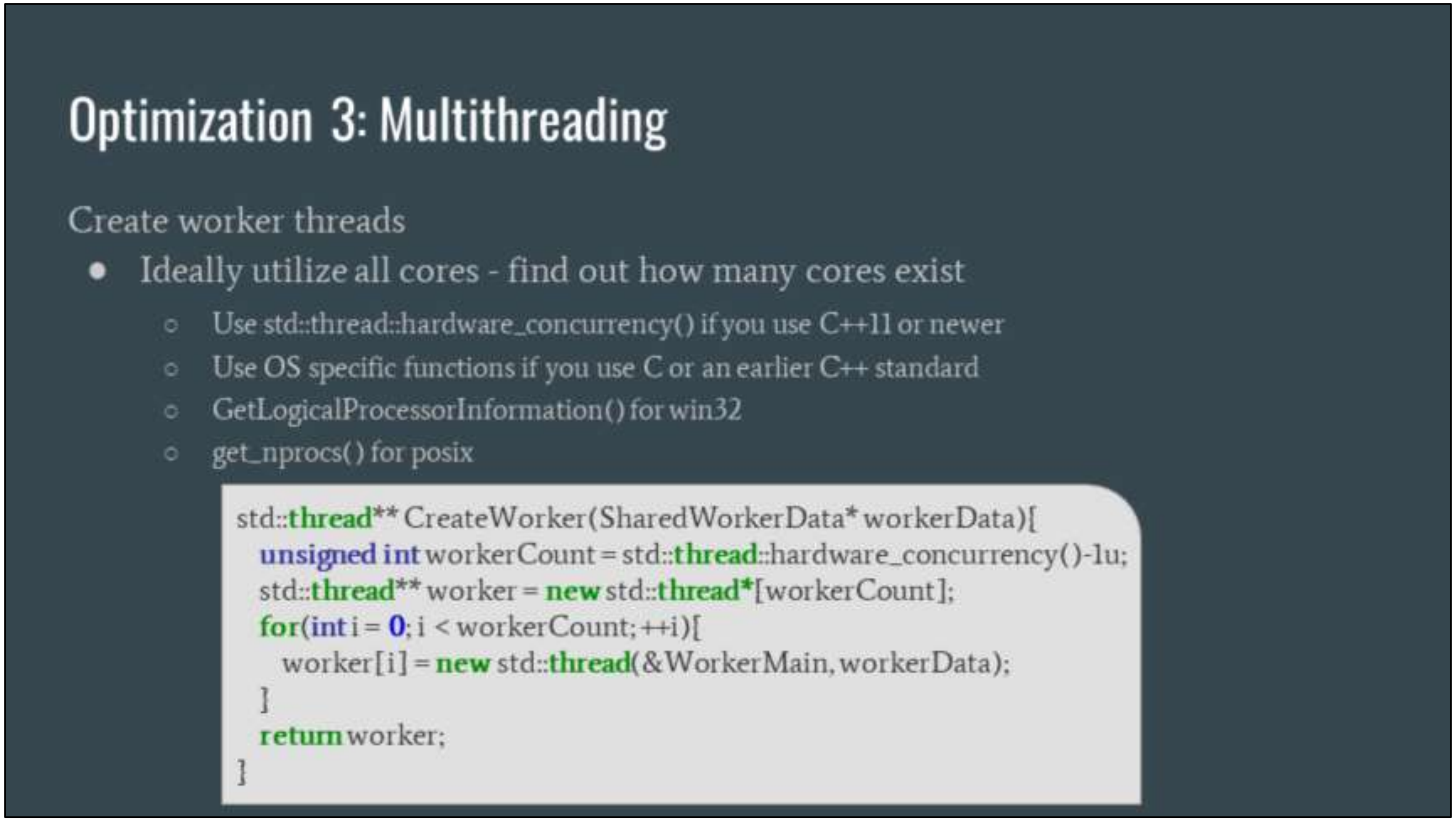
Optimization 3: Multithreading

Quick overview of things we have to do to add a job system

- Create worker threads
- Create independent jobs
- Schedule jobs
- Wait until all jobs are finished

So to get this to work is unfortunately not as easy as changing a few lines. For this to work we have to do the following things.

We have to find out how many cores the cpu has and create worker threads.
We have to create the data for the jobs.
We have to schedule the jobs - this is basically where the tell the workers that there are jobs to do.
After that we have to wait until all jobs are finished - at this time the image is rotated.



Optimization 3: Multithreading

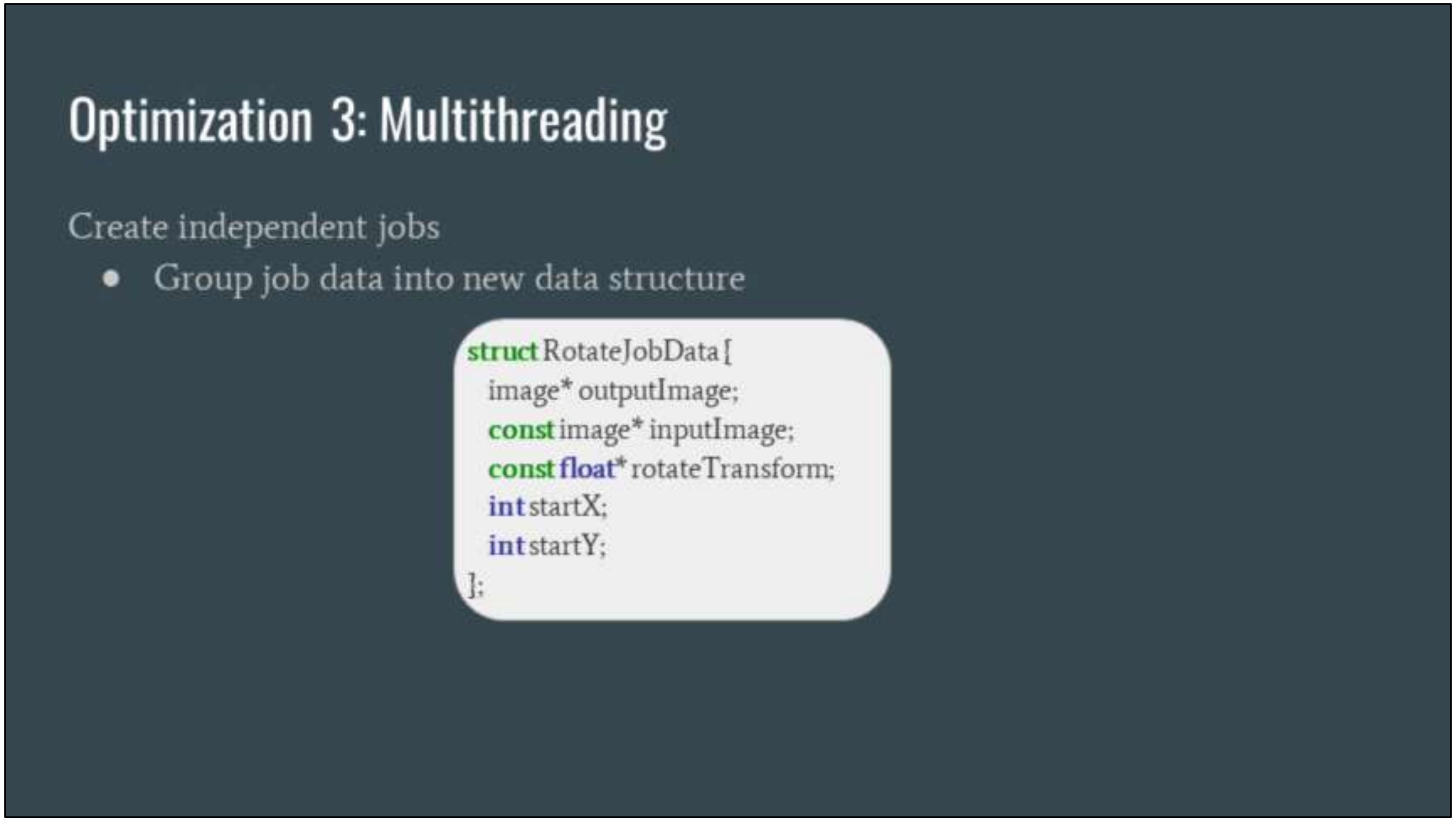
Create worker threads

- Ideally utilize all cores - find out how many cores exist
  - Use std::thread::hardware_concurrency() if you use C++11 or newer
  - Use OS specific functions if you use C or an earlier C++ standard
  - GetLogicalProcessorInformation() for win32
  - get_nprocs() for posix

```
std::thread** CreateWorker(SharedWorkerData* workerData){
  unsigned int workerCount = std::thread::hardware_concurrency()-1u;
  std::thread** worker = new std::thread*[workerCount];
  for(int i = 0; i < workerCount; ++i){
    worker[i] = new std::thread(&WorkerMain, workerData);
  }
  return worker;
}
```

To find out how many cores the current CPU has, we can either use the C++11 STL or use OS specific functions, like GetLogicalProcessorInformation() on win32.

A function based on the STL thread api could look like this.
Note that we subtract 1 from the core count since we also have the main thread that is already running on one of the cores.



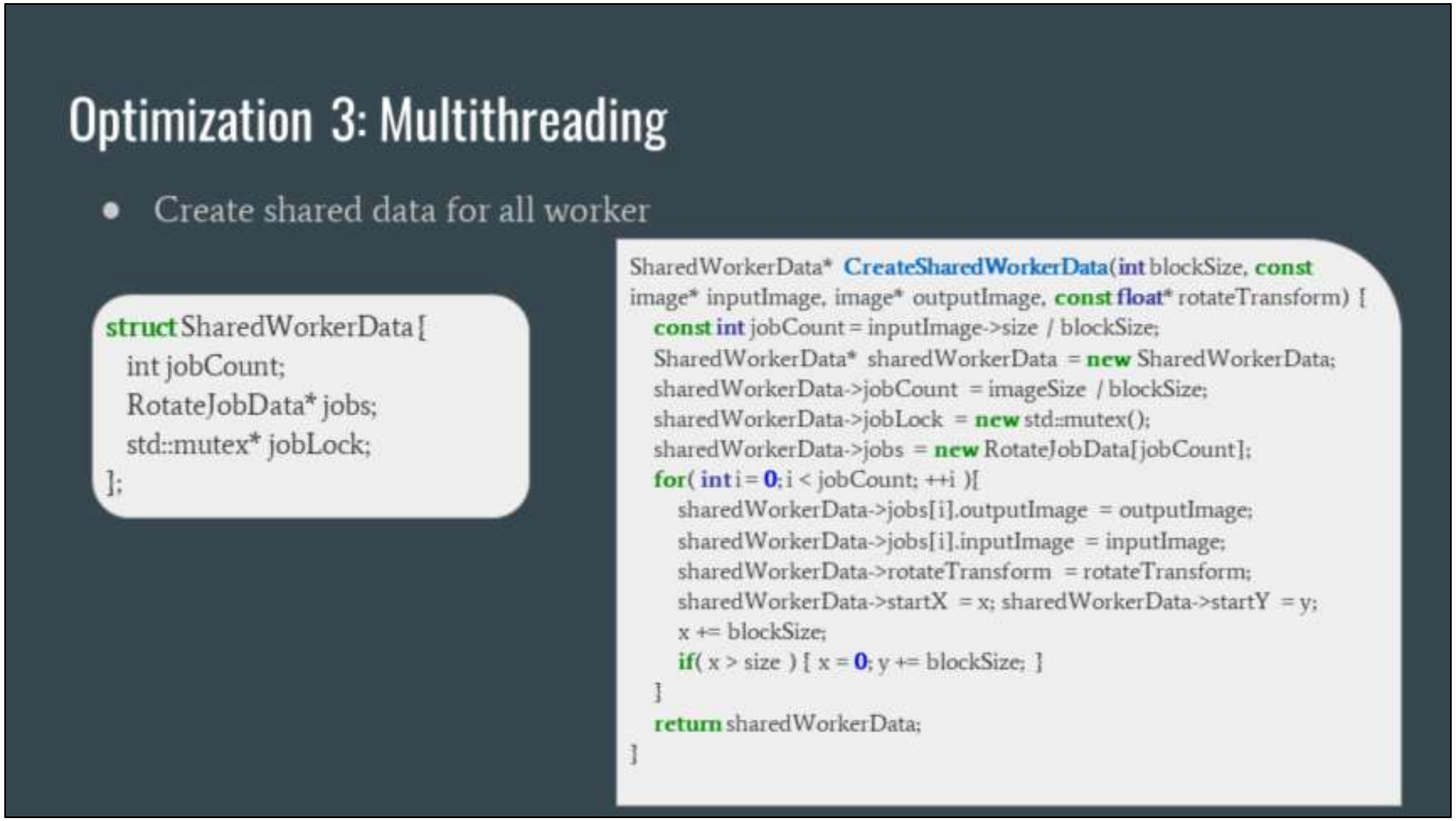
Optimization 3: Multithreading

Create independent jobs

- Group job data into new data structure

```
struct RotateJobData {
  image* outputImage;
  const image* inputImage;
  const float* rotateTransform;
  int startX;
  int startY;
};
```

For creating the independent jobs, we first create a new data structure that encapsulates all the data that a given jobs needs.



Optimization 3: Multithreading

- Create shared data for all worker

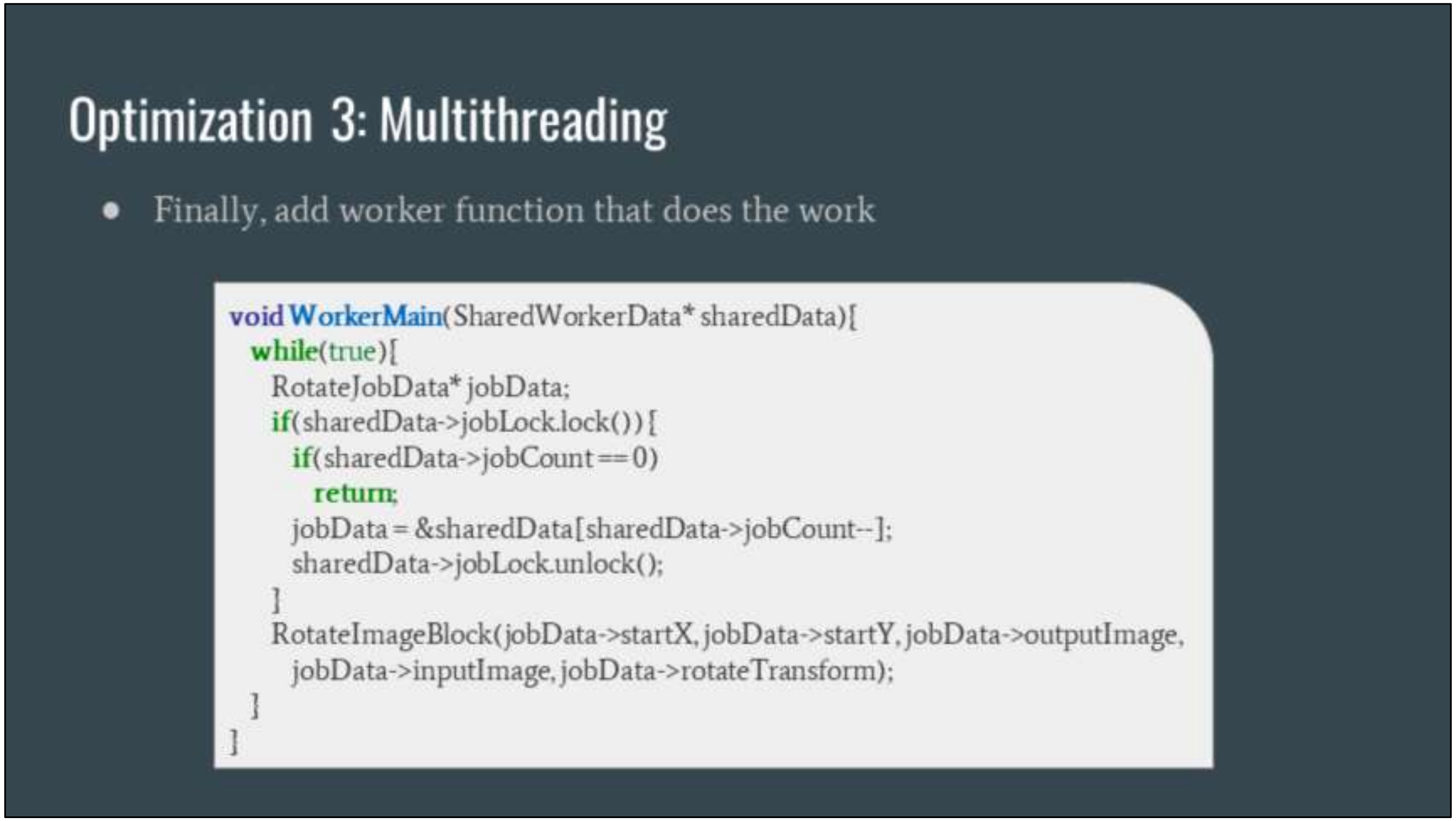
```
struct SharedWorkerData {
  int jobCount;
  RotateJobData* jobs;
  std::mutex* jobLock;
};
```

```
SharedWorkerData*  CreateSharedWorkerData(int blockSize, const
image* inputImage, image* outputImage, const float* rotateTransform) {
  const int jobCount = inputImage->size / blockSize;
  SharedWorkerData*  sharedWorkerData  = new SharedWorkerData;
  sharedWorkerData->jobCount  = imageSize  / blockSize;
  sharedWorkerData->jobLock  = new std::mutex();
  sharedWorkerData->jobs  = new RotateJobData[jobCount];
  for( int i = 0; i < jobCount; ++i ){
    sharedWorkerData->jobs[i].outputImage  = outputImage;
    sharedWorkerData->jobs[i].inputImage  = inputImage;
    sharedWorkerData->rotateTransform  = rotateTransform;
    sharedWorkerData->startX  = x; sharedWorkerData->startY  = y;
    x += blockSize;
    if( x > size ) { x = 0; y += blockSize; }
  }
  return sharedWorkerData;
}
```

Then we have the data that is shared between all workers.

This is: the amount of jobs available, the job data and a mutex, which we'll talk about in more detail later.

The function that creates and populates this shared data structure looks like this.



Optimization 3: Multithreading

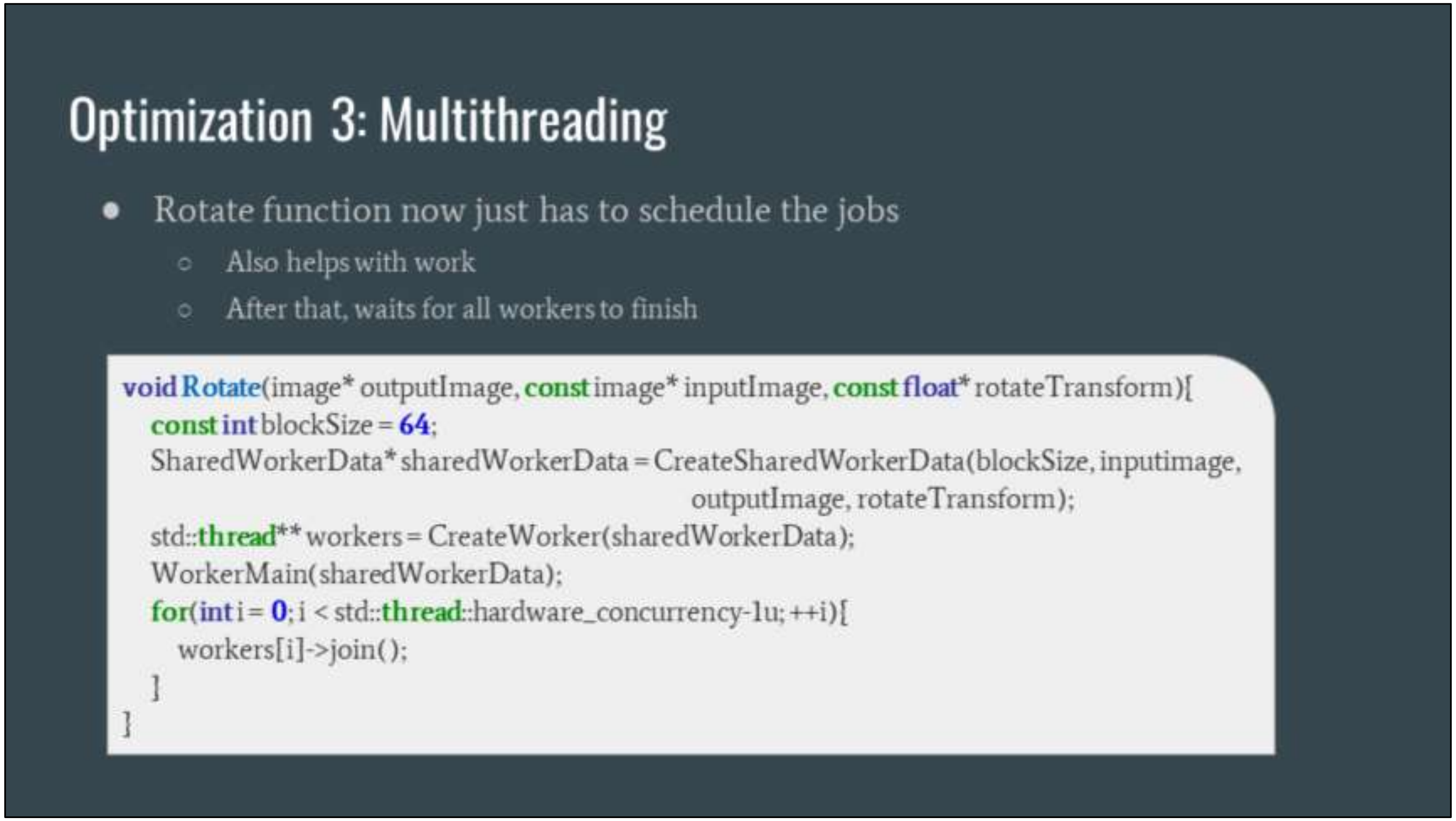
- Finally, add worker function that does the work

```
void WorkerMain(SharedWorkerData* sharedData){
  while(true){
    RotateJobData* jobData;
    if(sharedData->jobLock.lock()) {
      if(sharedData->jobCount == 0)
        return;
      jobData = &sharedData[sharedData->jobCount--];
      sharedData->jobLock.unlock();
    }
    RotateImageBlock(jobData->startX, jobData->startY, jobData->outputImage,
      jobData->inputImage, jobData->rotateTransform);
  }
}
```

And then we have the entry point for each worker thread.
This function basically runs as long as there are jobs available.

We have to make sure that the write access (during the decrement) is guarded by a mutex. Is a data structure that makes sure that only one thread can lock the mutex at any given point in time.
If a thread tries to lock a mutex that is already locked by another thread, it'll wait until the mutex is unlocked.

Once we have the job data, we just call the already known RotateImageBlock function from the loop blocking optimization.



Optimization 3: Multithreading

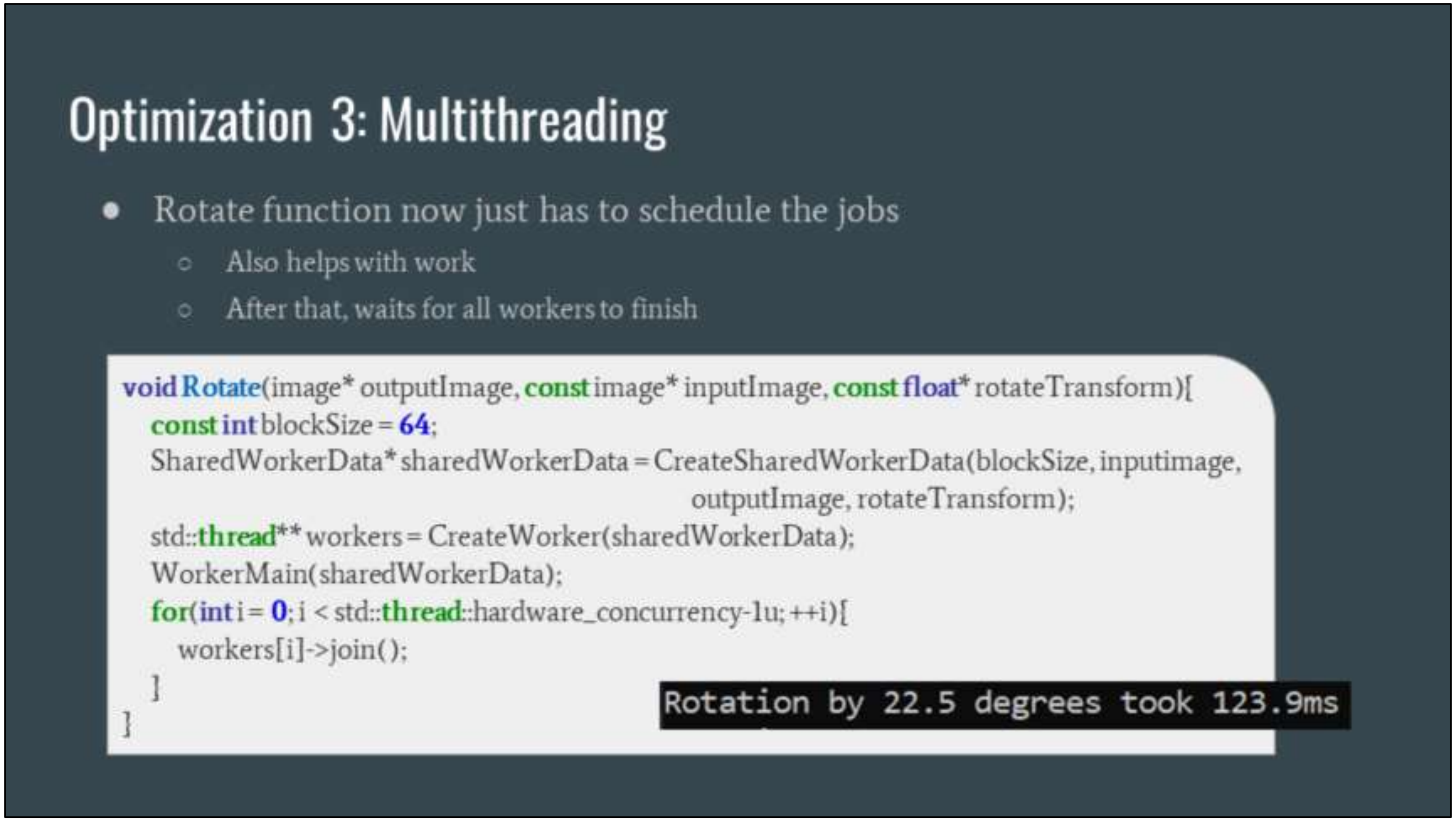
- Rotate function now just has to schedule the jobs
  - Also helps with work
  - After that, waits for all workers to finish

```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform){
   const int blockSize = 64;
   SharedWorkerData* sharedWorkerData = CreateSharedWorkerData(blockSize, inputimage,
                                                               outputImage, rotateTransform);
   std::thread** workers = CreateWorker(sharedWorkerData);
   WorkerMain(sharedWorkerData);
   for(int i = 0; i < std::thread::hardware_concurrency-1u; ++i){
      workers[i]->join();
   }
}
```

The complete, new Rotate function now looks like this.
We first create the shared worker data, which also creates all the job data.
We then create all the worker and kick them off and after that we either work on non scheduled jobs or wait until all threads are finish.

After that the rotation of the image is done.

Looking at the performance result we see a big improvement - but be aware that this scales with the number of cores in the target CPU.



Optimization 3: Multithreading

- Rotate function now just has to schedule the jobs
  - Also helps with work
  - After that, waits for all workers to finish

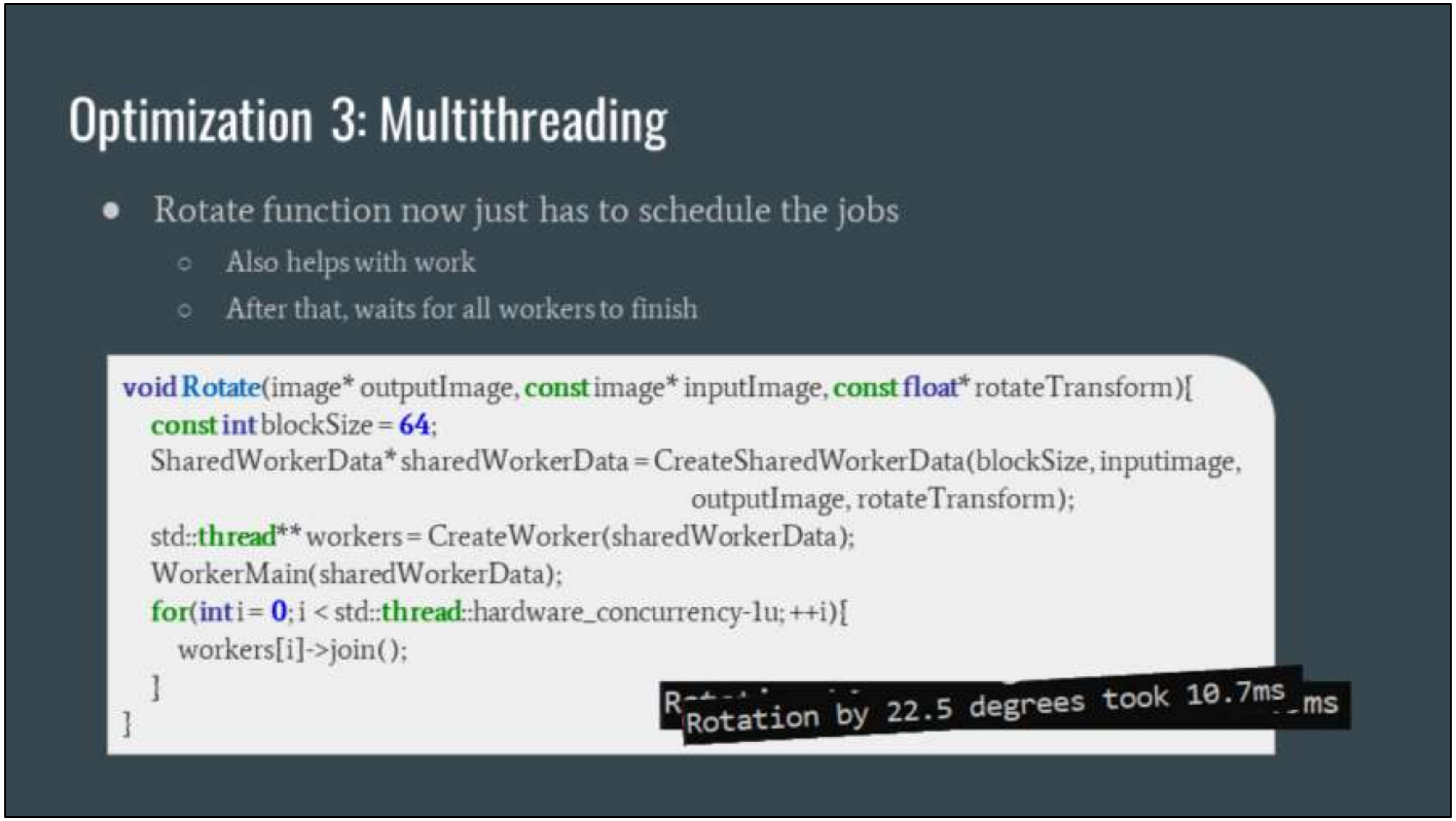
```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform)[
   const int blockSize = 64;
   SharedWorkerData* sharedWorkerData = CreateSharedWorkerData(blockSize, inputimage,
                                                          outputImage, rotateTransform);
   std::thread** workers = CreateWorker(sharedWorkerData);
   WorkerMain(sharedWorkerData);
   for(int i = 0; i < std::thread::hardware_concurrency-1u; ++i)[
      workers[i]->join();
   ]
]
```

```
Rotation by 22.5 degrees took 123.9ms
```

The complete, new Rotate function now looks like this.
We first create the shared worker data, which also creates all the job data.
We then create all the worker and kick them off and after that we either work on non scheduled jobs or wait until all threads are finish.

After that the rotation of the image is done.

Looking at the performance result we see a big improvement - but be aware that this scales with the number of cores in the target CPU.



Optimization 3: Multithreading

- Rotate function now just has to schedule the jobs
  - Also helps with work
  - After that, waits for all workers to finish

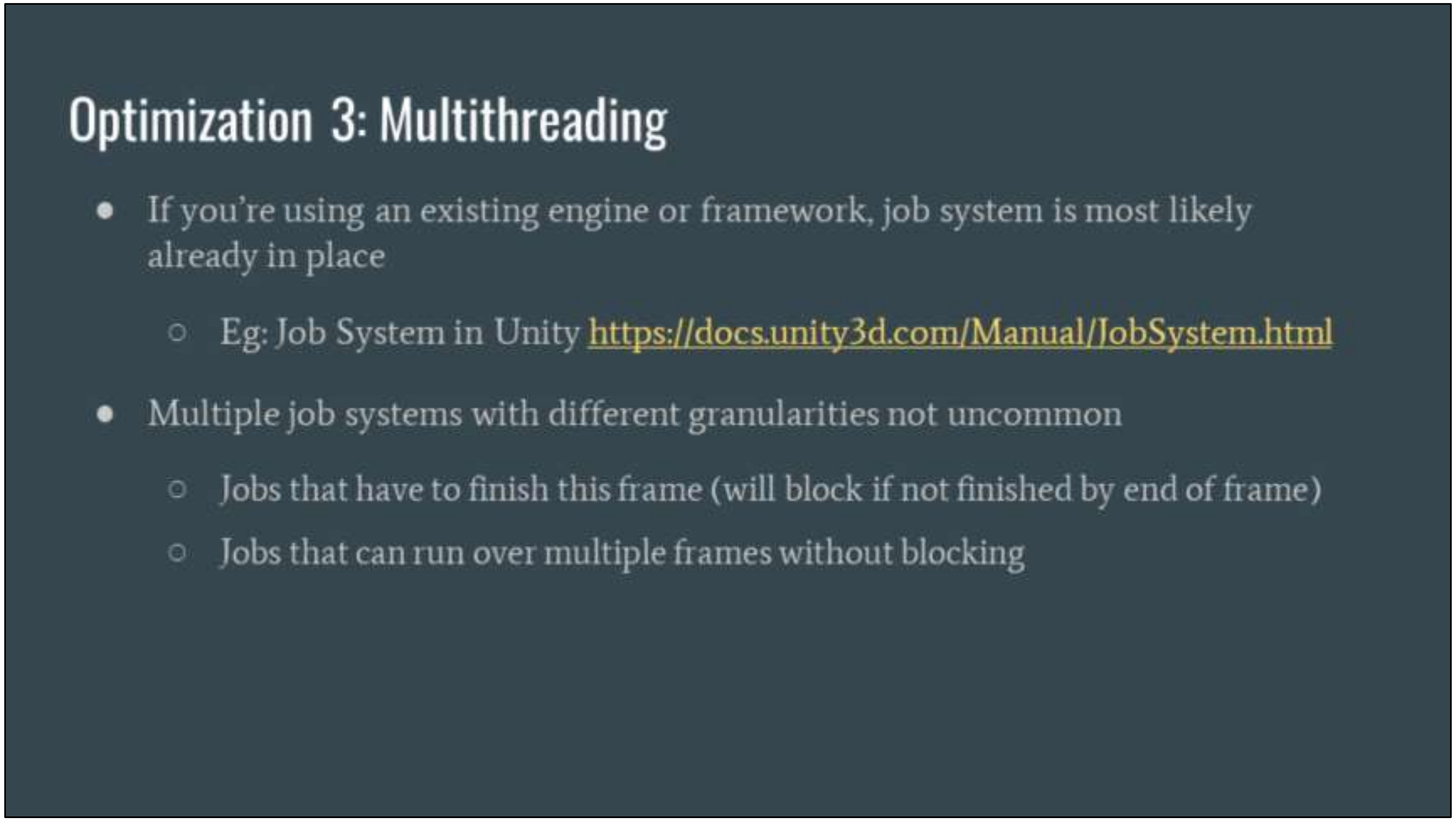
```
void Rotate(image* outputImage, const image* inputImage, const float* rotateTransform){
   const int blockSize = 64;
   SharedWorkerData* sharedWorkerData = CreateSharedWorkerData(blockSize, inputimage,
                                                               outputImage, rotateTransform);
   std::thread** workers = CreateWorker(sharedWorkerData);
   WorkerMain(sharedWorkerData);
   for(int i = 0; i < std::thread::hardware_concurrency-1u; ++i){
      workers[i]->join();
   }
}
```

```
Rotation by 22.5 degrees took 10.7ms
```

The complete, new Rotate function now looks like this.
We first create the shared worker data, which also creates all the job data.
We then create all the worker and kick them off and after that we either work on non scheduled jobs or wait until all threads are finish.

After that the rotation of the image is done.

Looking at the performance result we see a big improvement - but be aware that this scales with the number of cores in the target CPU.



Optimization 3: Multithreading

- If you're using an existing engine or framework, job system is most likely already in place
  - Eg: Job System in Unity https://docs.unity3d.com/Manual/JobSystem.html
- Multiple job systems with different granularities not uncommon
  - Jobs that have to finish this frame (will block if not finished by end of frame)
  - Jobs that can run over multiple frames without blocking

Note that most engines or game frameworks already have a job system in place that you can just use, without having to create the workers by hand.

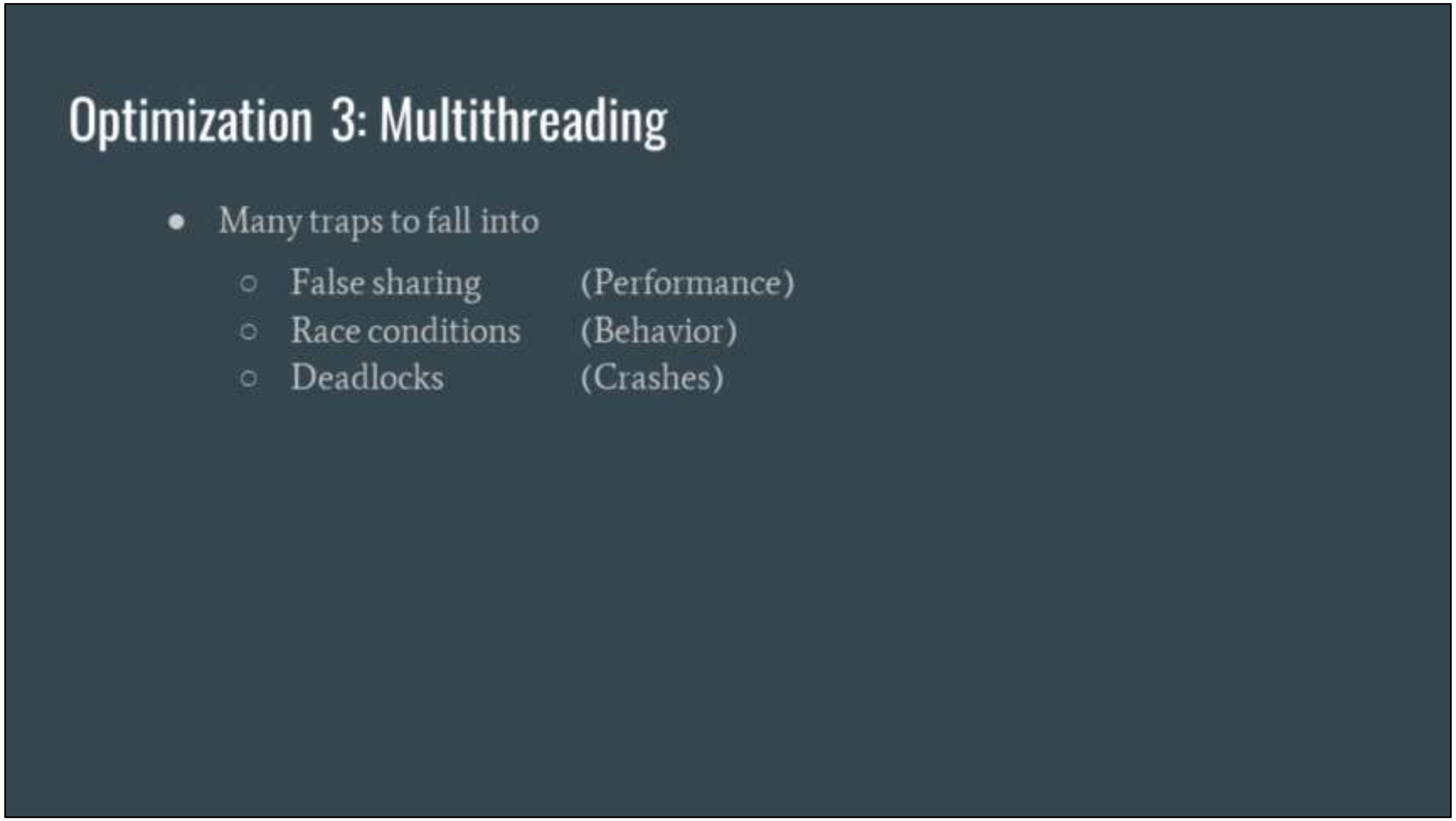
It's also not uncommon to have multiple job systems with different granularities where eg. jobs in one job system have to be done within a frame (the frame will block until all jobs are finished) while jobs in another job system, within the same engine, can run over multiple frames (savegames come to mind).

Optimization 3: Multithreading

- Many traps to fall into

Again, there are many traps to fall into with multithreading code like false sharing, race conditions or dead locks. All of there problems aren't necessarily super obvious and your program might run fine on your machine but crash horribly on another.

That's why I have to re-empathize that it makes sense to make data sharing as simple as possible to not run into these problems. A simple job queue like in our example will fit most use cases.

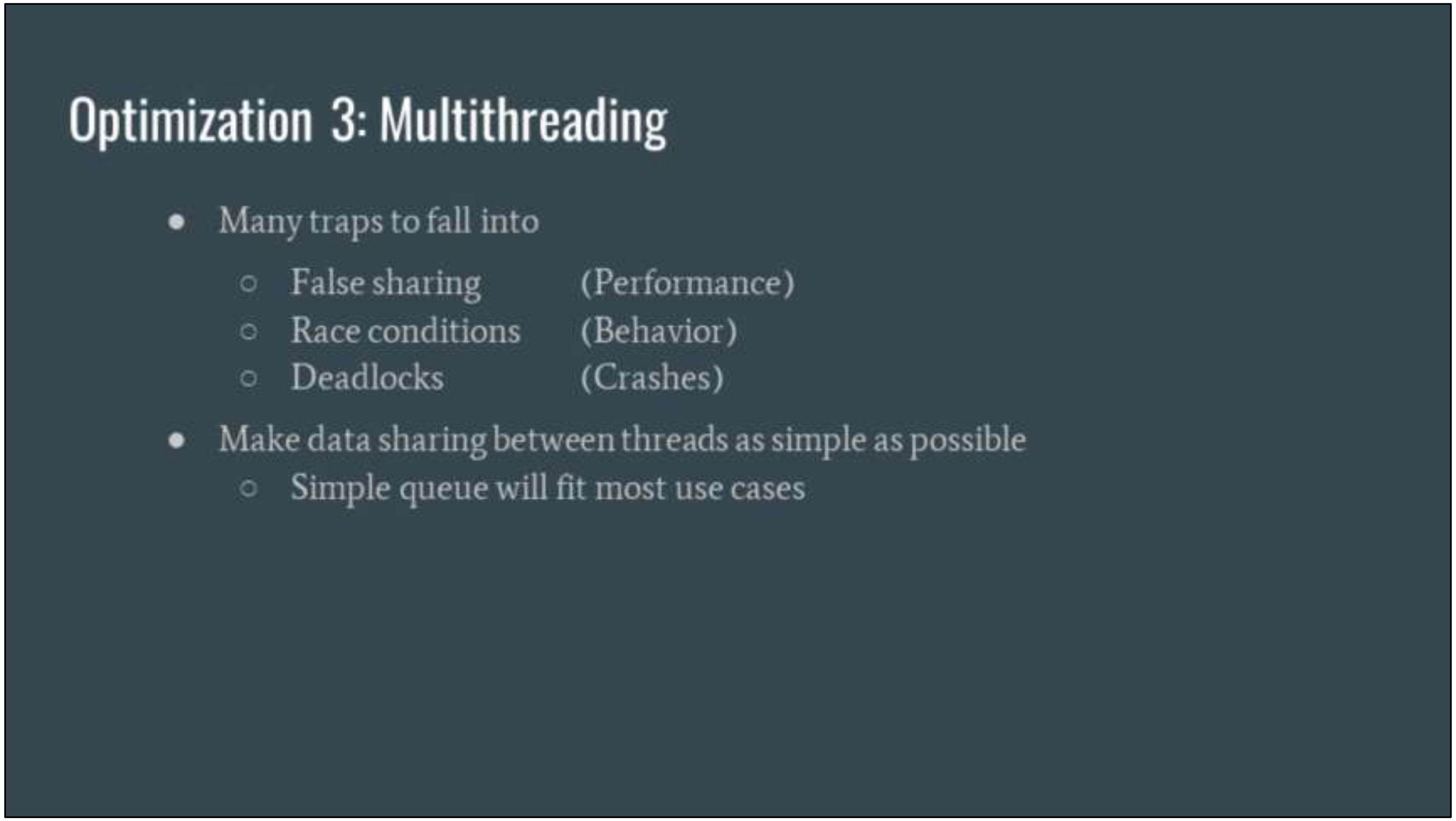


Optimization 3: Multithreading

- Many traps to fall into
  - False sharing (Performance)
  - Race conditions (Behavior)
  - Deadlocks (Crashes)

Again, there are many traps to fall into with multithreading code like false sharing, race conditions or dead locks. All of there problems aren't necessarily super obvious and your program might run fine on your machine but crash horribly on another.

That's why I have to re-empathize that it makes sense to make data sharing as simple as possible to not run into these problems. A simple job queue like in our example will fit most use cases.

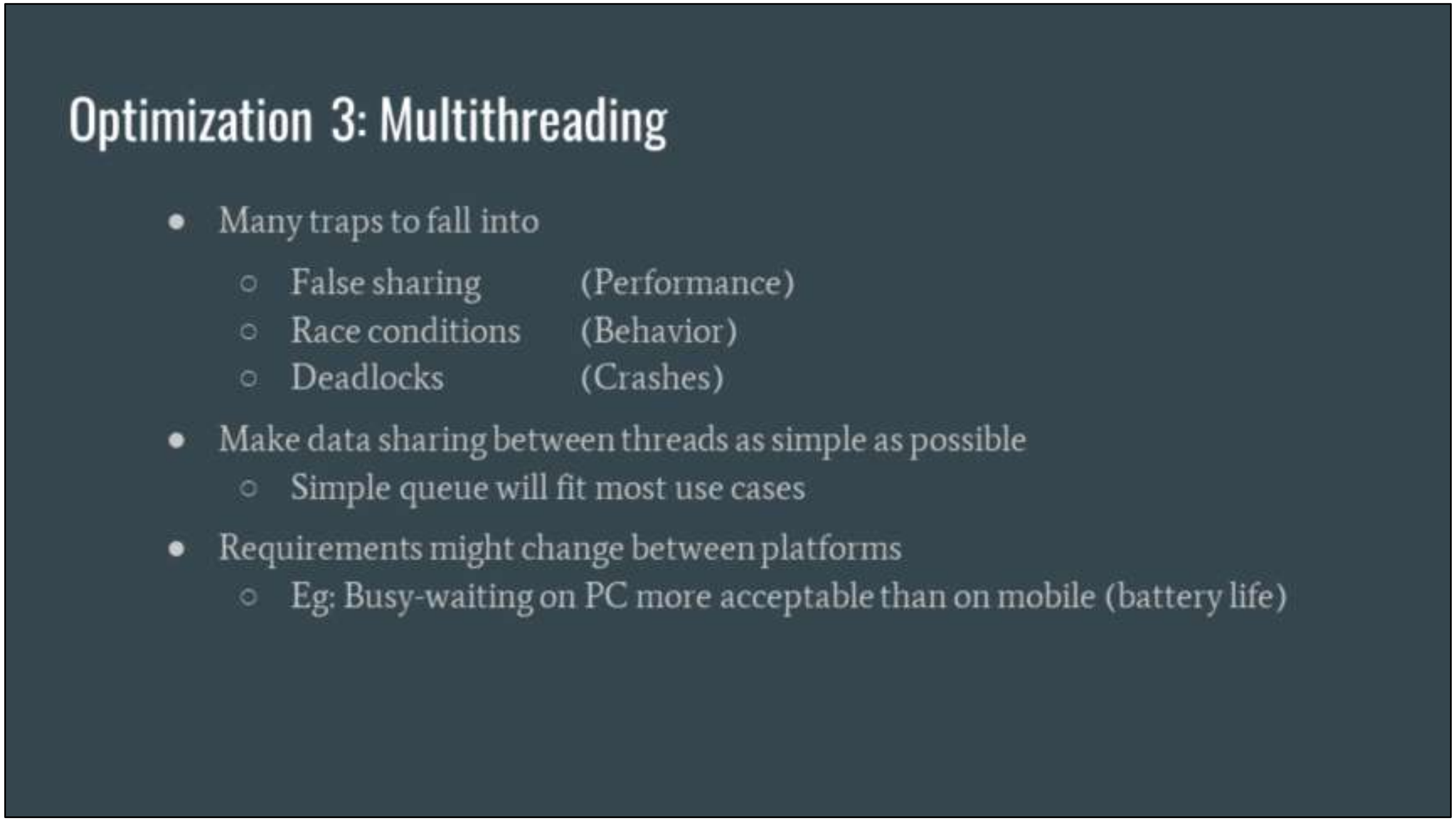


Optimization 3: Multithreading

- Many traps to fall into
  - False sharing (Performance)
  - Race conditions (Behavior)
  - Deadlocks (Crashes)
- Make data sharing between threads as simple as possible
  - Simple queue will fit most use cases

Again, there are many traps to fall into with multithreading code like false sharing, race conditions or dead locks. All of there problems aren’t necessarily super obvious and your program might run fine on your machine but crash horribly on another.

That’s why I have to re-empathize that it makes sense to make data sharing as simple as possible to not run into these problems. A simple job queue like in our example will fit most use cases.



Optimization 3: Multithreading

- Many traps to fall into
  - False sharing (Performance)
  - Race conditions (Behavior)
  - Deadlocks (Crashes)
- Make data sharing between threads as simple as possible
  - Simple queue will fit most use cases
- Requirements might change between platforms
  - Eg: Busy-waiting on PC more acceptable than on mobile (battery life)

Again, there are many traps to fall into with multithreading code like false sharing, race conditions or dead locks. All of there problems aren’t necessarily super obvious and your program might run fine on your machine but crash horribly on another.

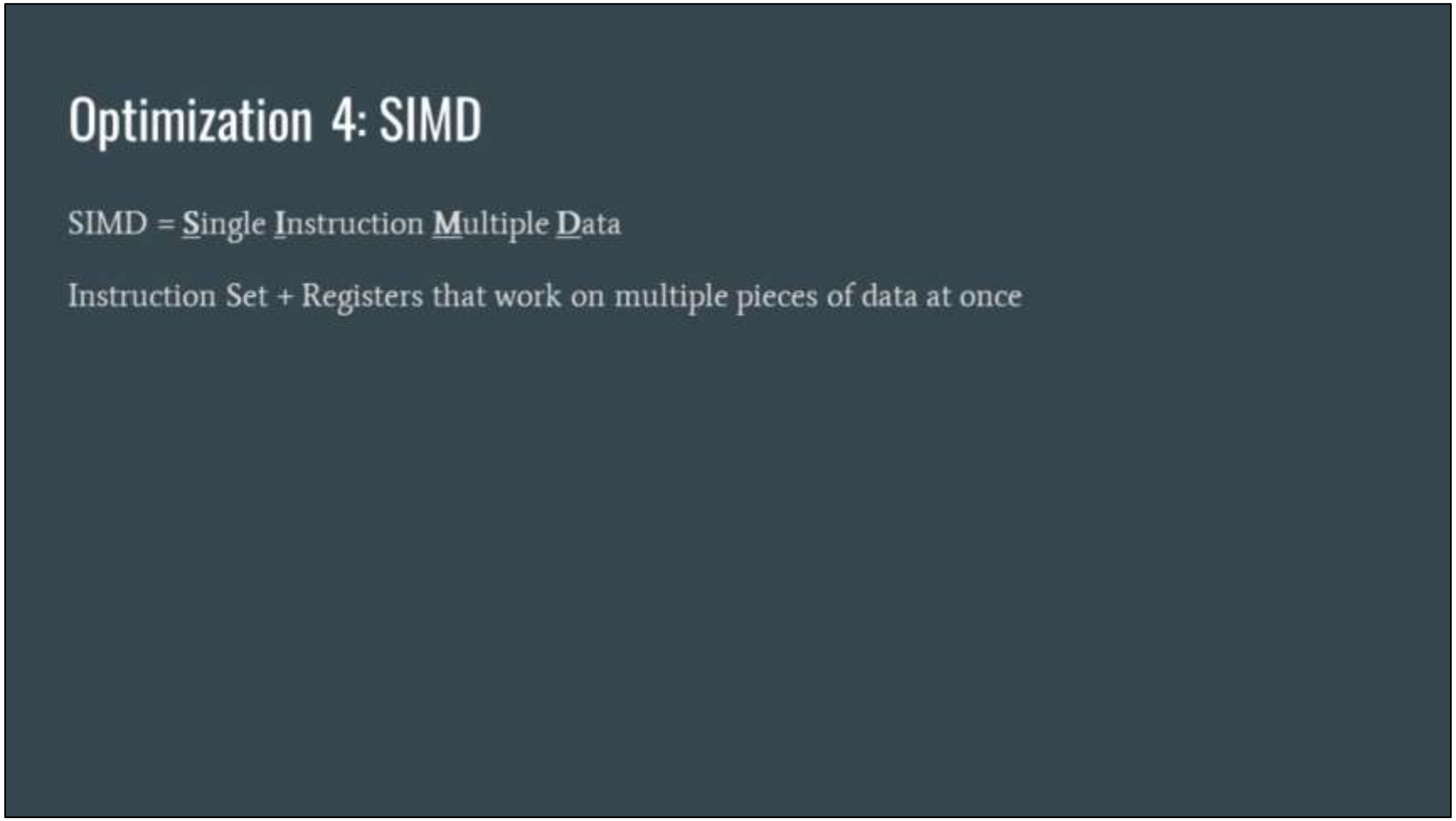
That’s why I have to re-empathize that it makes sense to make data sharing as simple as possible to not run into these problems. A simple job queue like in our example will fit most use cases.

Optimization 4: SIMD

SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

The last optimization that I want to talk about today is SIMD, which stands for Single Instruction Multiple Data.
This is basically one instruction - like mul which works on multiple values at once.

To give you a concrete example consider this function which multiplies 4 floats by a multiplier.
The scalar version of this function, where every multiplication is done one after the other, would look like this.
The SIMD version of this function, where all multiplications are done with a single instruction, would look like this.



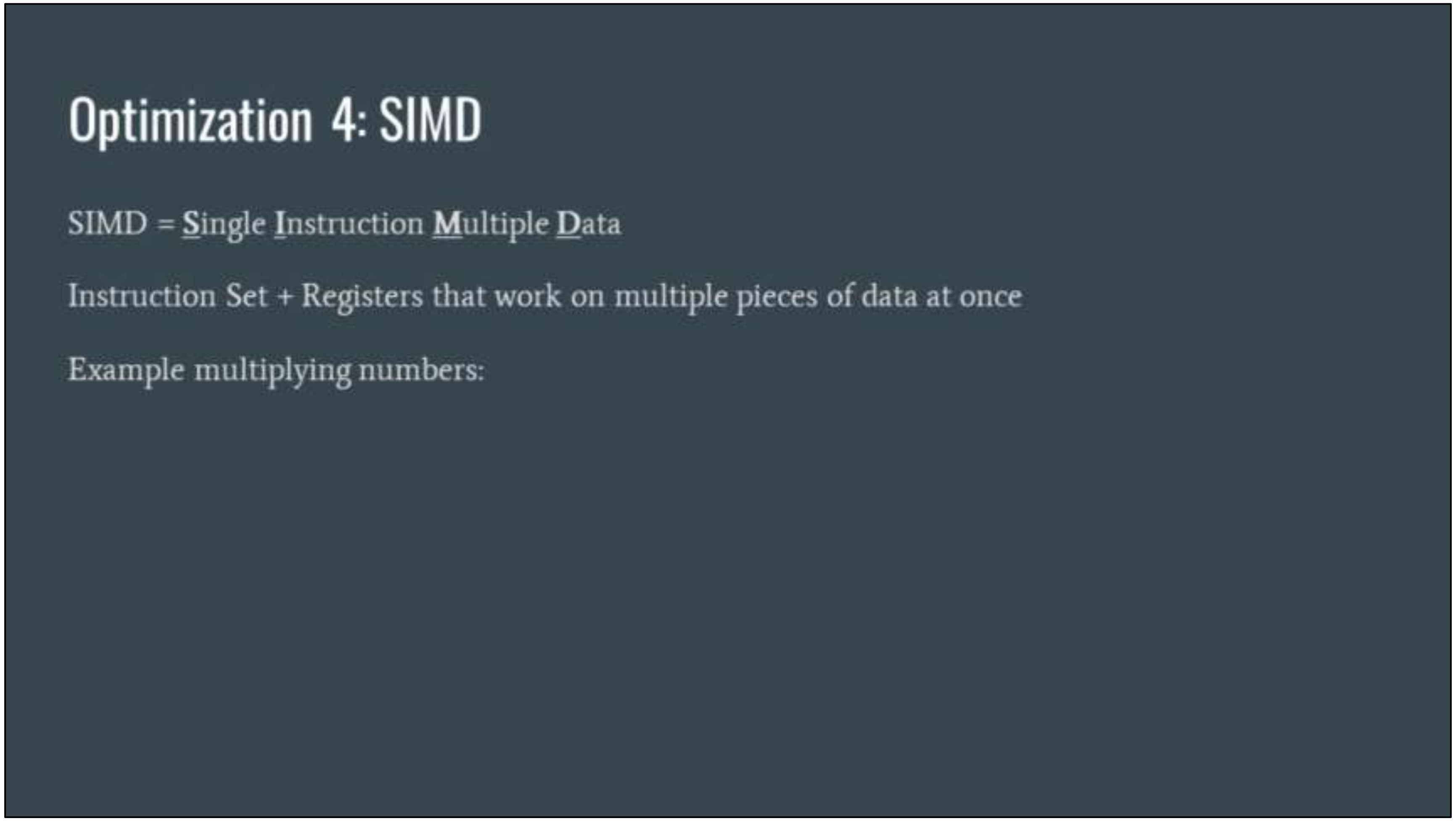
Optimization 4: SIMD

SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

Instruction Set + Registers that work on multiple pieces of data at once

The last optimization that I want to talk about today is SIMD, which stands for Single Instruction Multiple Data.
This is basically one instruction - like mul which works on multiple values at once.

To give you a concrete example consider this function which multiplies 4 floats by a multiplier.
The scalar version of this function, where every multiplication is done one after the other, would look like this.
The SIMD version of this function, where all multiplications are done with a single instruction, would look like this.



Optimization 4: SIMD

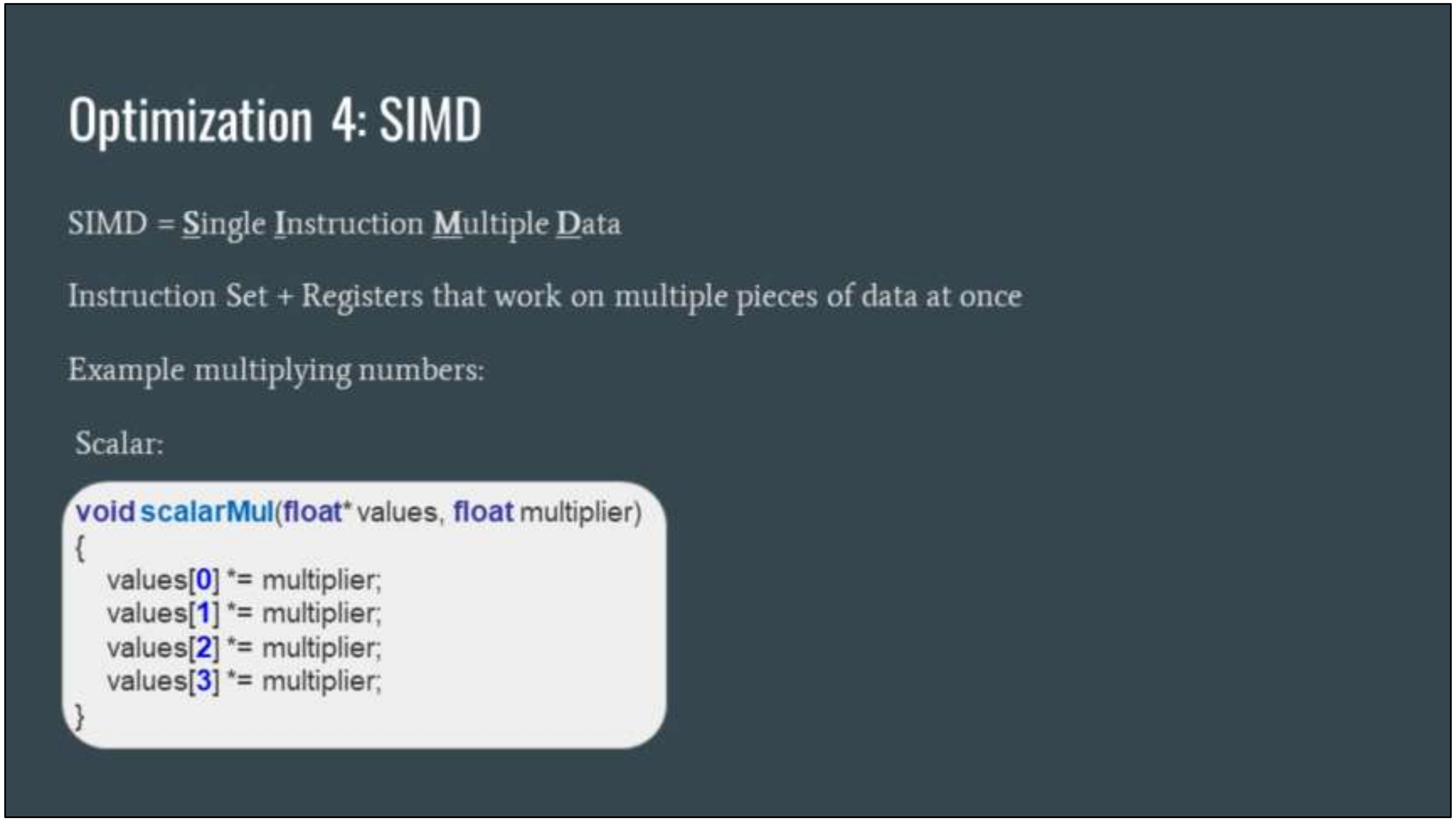
SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

Instruction Set + Registers that work on multiple pieces of data at once

Example multiplying numbers:

The last optimization that I want to talk about today is SIMD, which stands for Single Instruction Multiple Data.
This is basically one instruction - like mul which works on multiple values at once.

To give you a concrete example consider this function which multiplies 4 floats by a multiplier.
The scalar version of this function, where every multiplication is done one after the other, would look like this.
The SIMD version of this function, where all multiplications are done with a single instruction, would look like this.



Optimization 4: SIMD

SIMD = Single Instruction Multiple Data

Instruction Set + Registers that work on multiple pieces of data at once

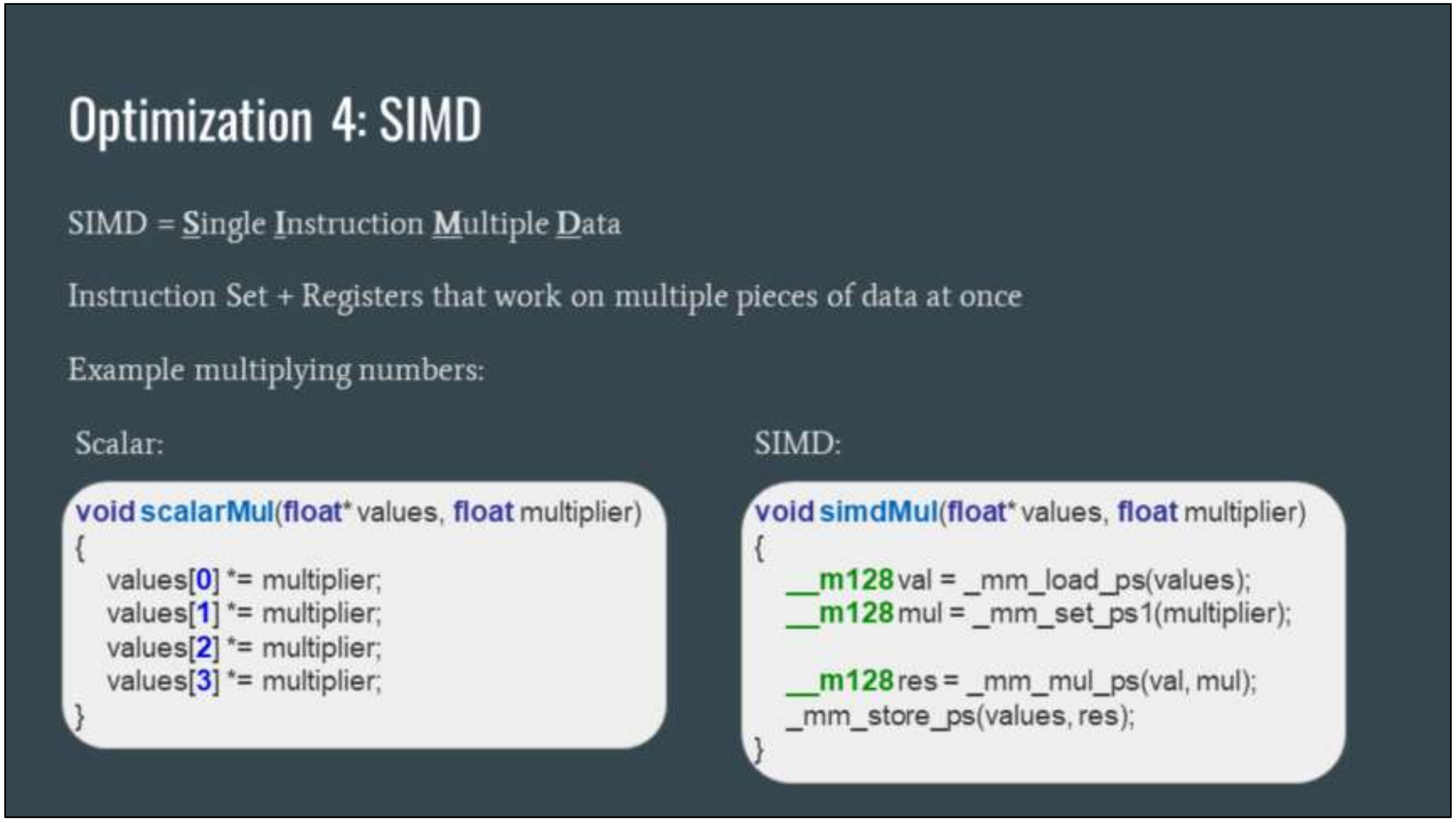
Example multiplying numbers:

Scalar:

```
void scalarMul(float* values, float multiplier)
{
   values[0] *= multiplier;
   values[1] *= multiplier;
   values[2] *= multiplier;
   values[3] *= multiplier;
}
```

The last optimization that I want to talk about today is SIMD, which stands for Single Instruction Multiple Data.
This is basically one instruction - like mul which works on multiple values at once.

To give you a concrete example consider this function which multiplies 4 floats by a multiplier.
The scalar version of this function, where every multiplication is done one after the other, would look like this.
The SIMD version of this function, where all multiplications are done with a single instruction, would look like this.



Optimization 4: SIMD

SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

Instruction Set + Registers that work on multiple pieces of data at once

Example multiplying numbers:

Scalar:

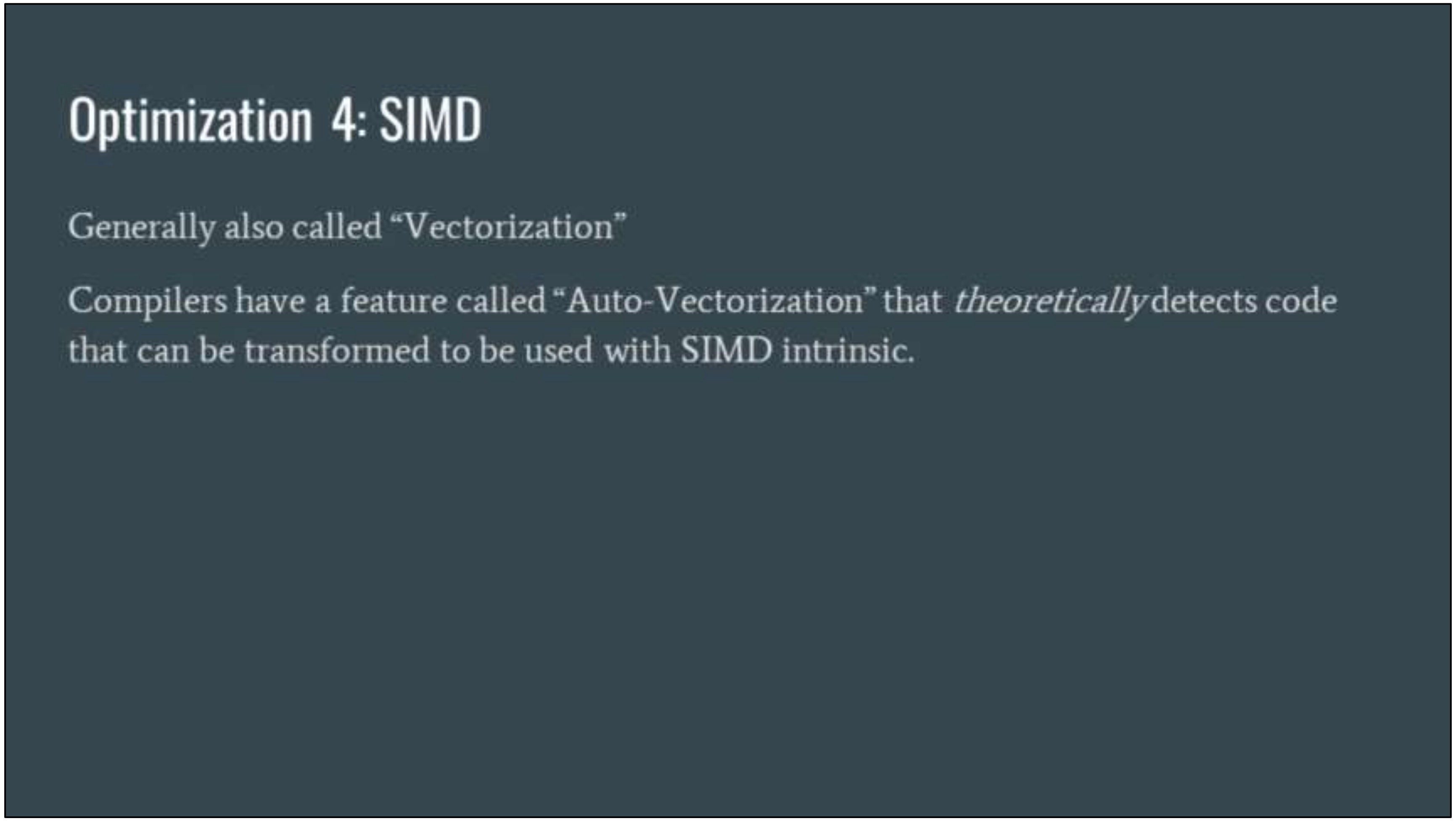
```
void scalarMul(float* values, float multiplier)
{
   values[0] *= multiplier;
   values[1] *= multiplier;
   values[2] *= multiplier;
   values[3] *= multiplier;
}
```

SIMD:

```
void simdMul(float* values, float multiplier)
{
   __m128 val = _mm_load_ps(values);
   __m128 mul = _mm_set_ps1(multiplier);

   __m128 res = _mm_mul_ps(val, mul);
   _mm_store_ps(values, res);
}
```

The last optimization that I want to talk about today is SIMD, which stands for Single Instruction Multiple Data.
This is basically one instruction - like mul which works on multiple values at once.

To give you a concrete example consider this function which multiplies 4 floats by a multiplier.
The scalar version of this function, where every multiplication is done one after the other, would look like this.
The SIMD version of this function, where all multiplications are done with a single instruction, would look like this.



Optimization 4: SIMD

Generally also called "Vectorization"

Compilers have a feature called "Auto-Vectorization" that *theoretically* detects code that can be transformed to be used with SIMD intrinsic.

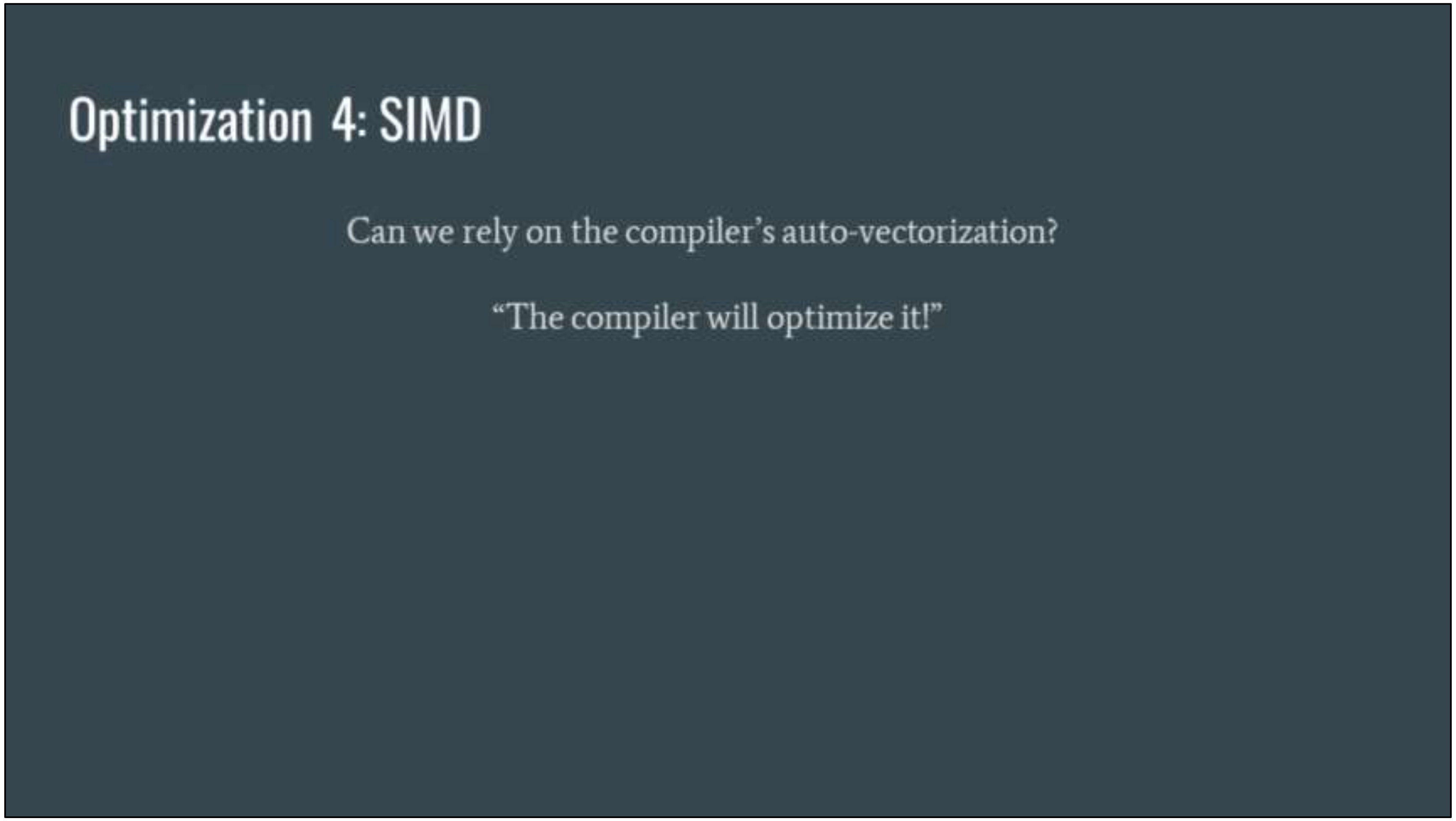
Programming using SIMD is also known as "vectorization".
There's a feature called "auto-vectorization" that is part of the optimization step of most compilers. This step tries to analyze the code and to find use cases where the compiler can identify a "SIMD-Pattern" and generate SIMD instructions instead of scalar instruction.

Optimization 4: SIMD

Can we rely on the compiler's auto-vectorization?

But can we really rely on this?
There are many people out there in forums or chat groups that will just blindly trust the compiler.

But can we really do that?



Optimization 4: SIMD

Can we rely on the compiler's auto-vectorization?

"The compiler will optimize it!"

But can we really rely on this?
There are many people out there in forums or chat groups that will just blindly trust the compiler.

But can we really do that?

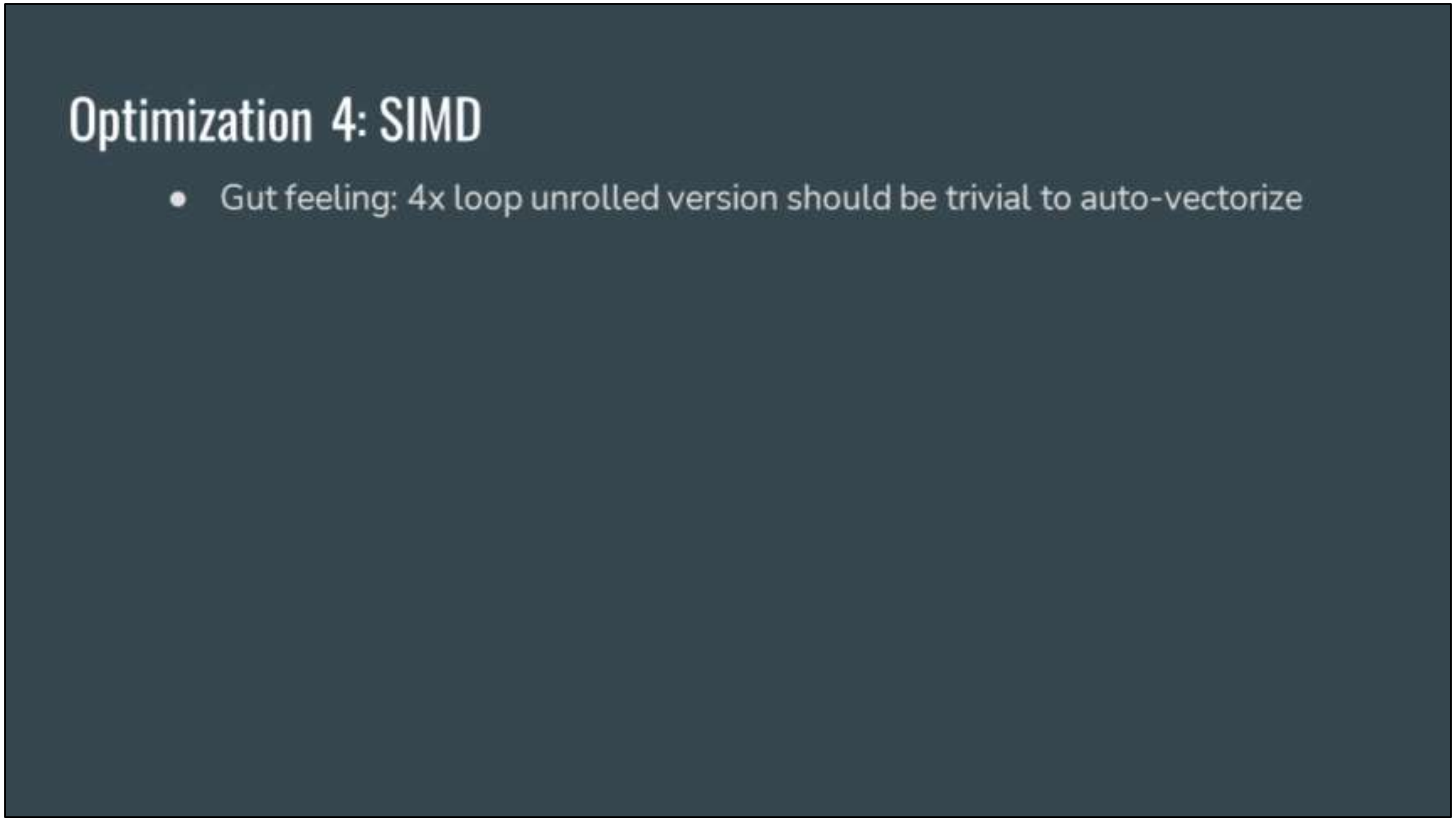
Optimization 4: SIMD

Can we rely on the compiler's auto-vectorization?

"The compiler will optimize it!"

But can we really rely on this?
There are many people out there in forums or chat groups that will just blindly trust the compiler.

But can we really do that?

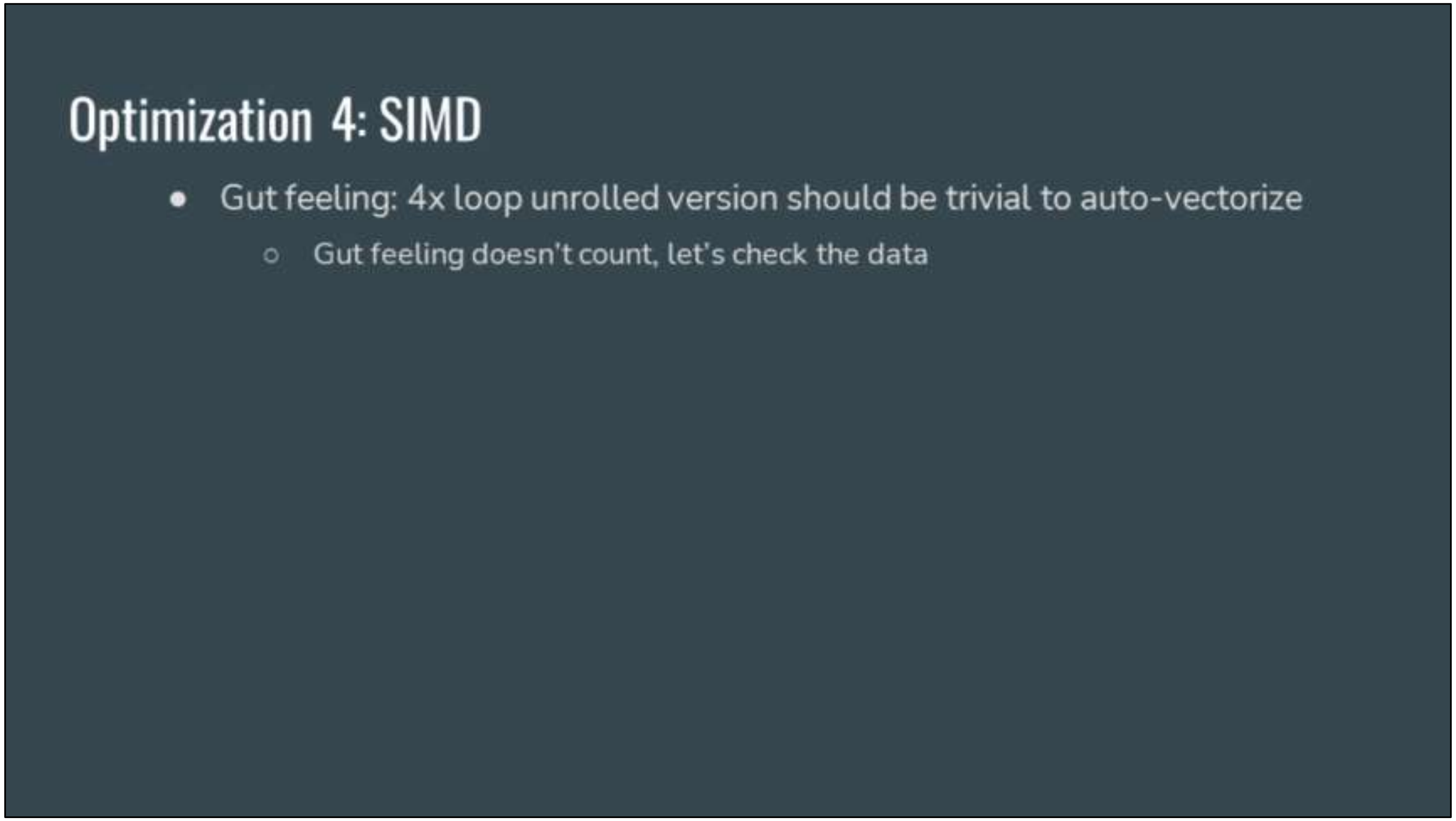


Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize

Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn’t count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there’s zero vectorization in our code. And that’s with optimizations enabled on the latest compiler version.

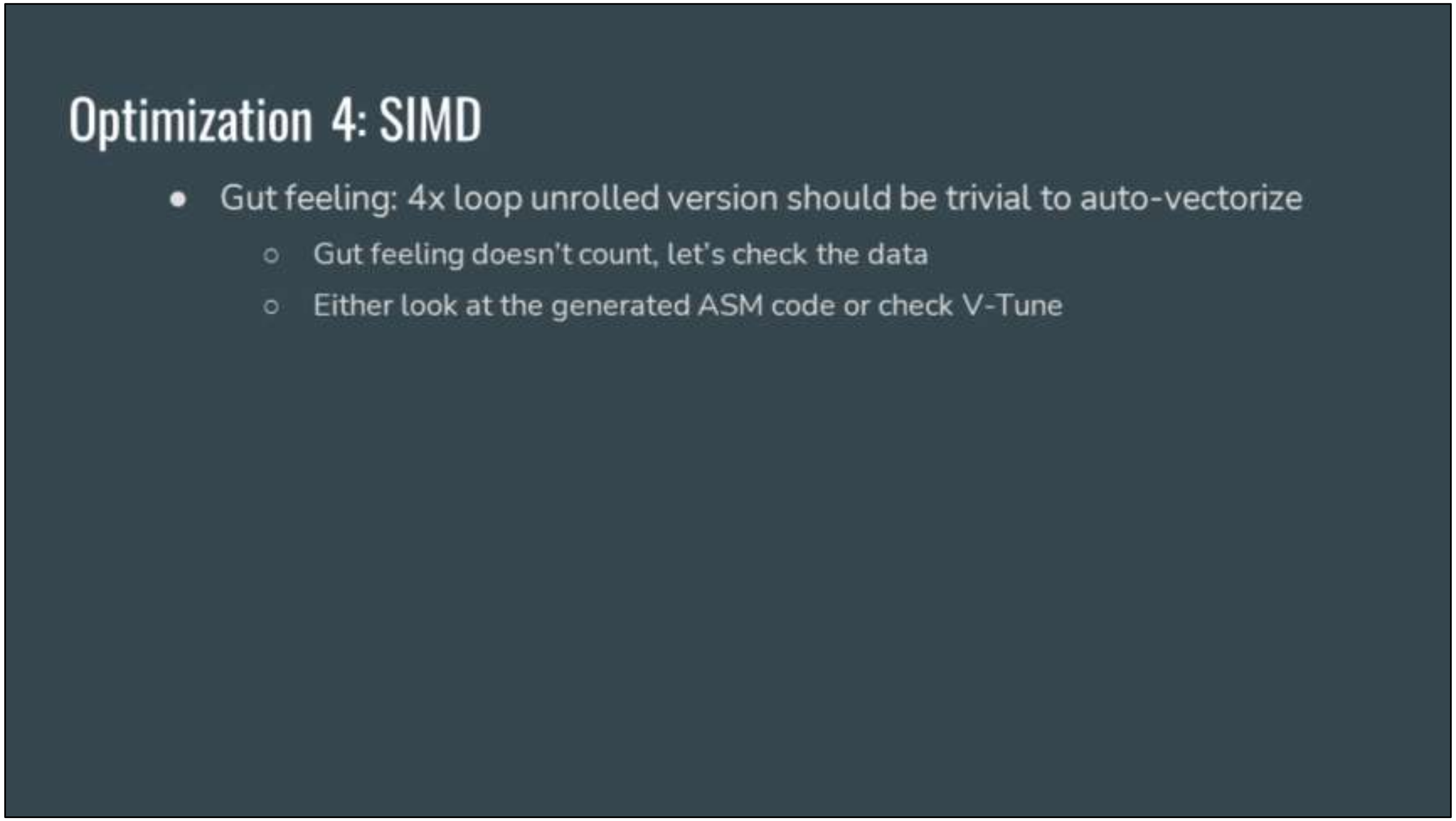


Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize
  - Gut feeling doesn't count, let's check the data

Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn’t count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there’s zero vectorization in our code. And that’s with optimizations enabled on the latest compiler version.

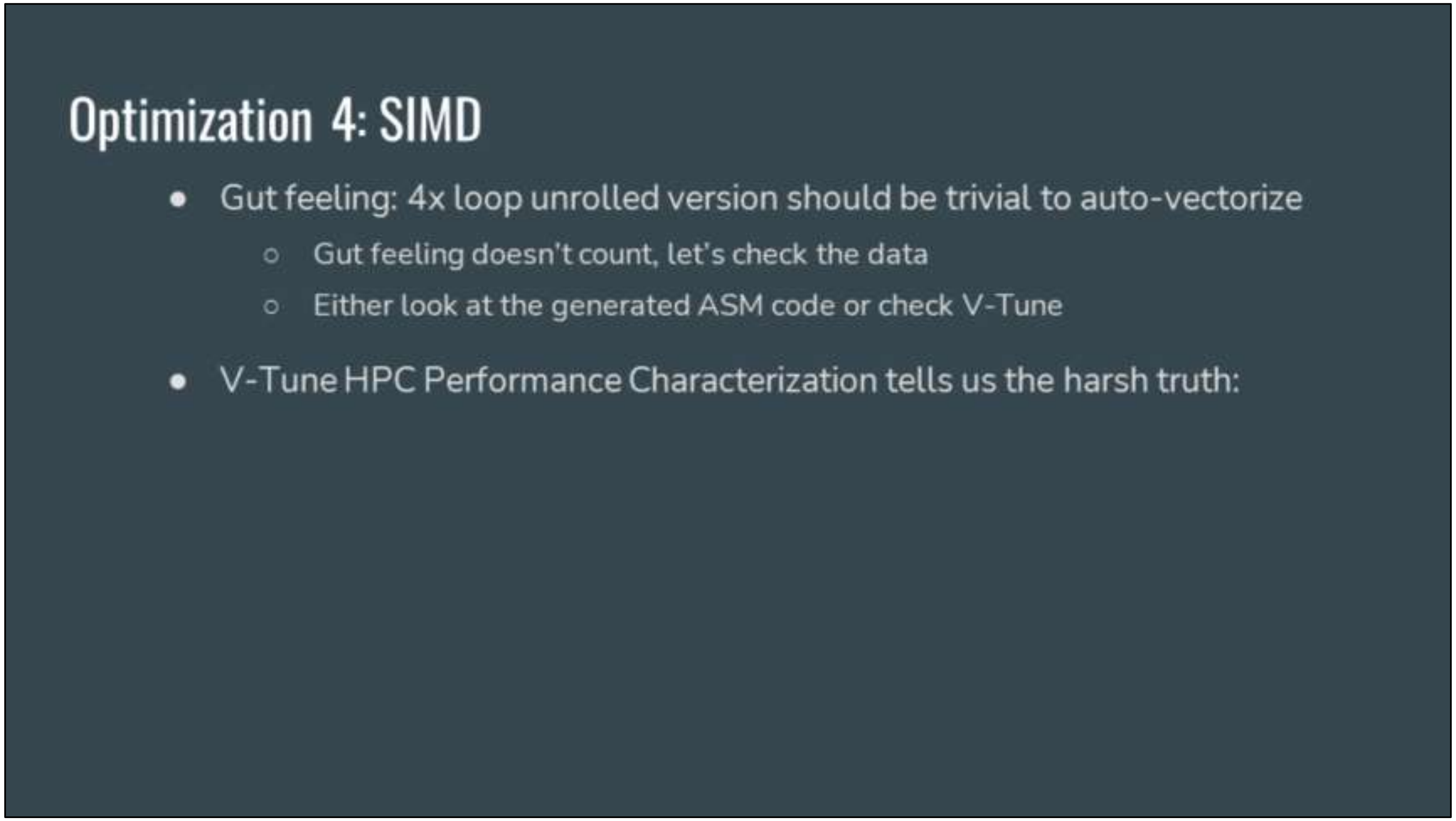


Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize
  - Gut feeling doesn't count, let's check the data
  - Either look at the generated ASM code or check V-Tune

Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn't count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there's zero vectorization in our code. And that's with optimizations enabled on the latest compiler version.

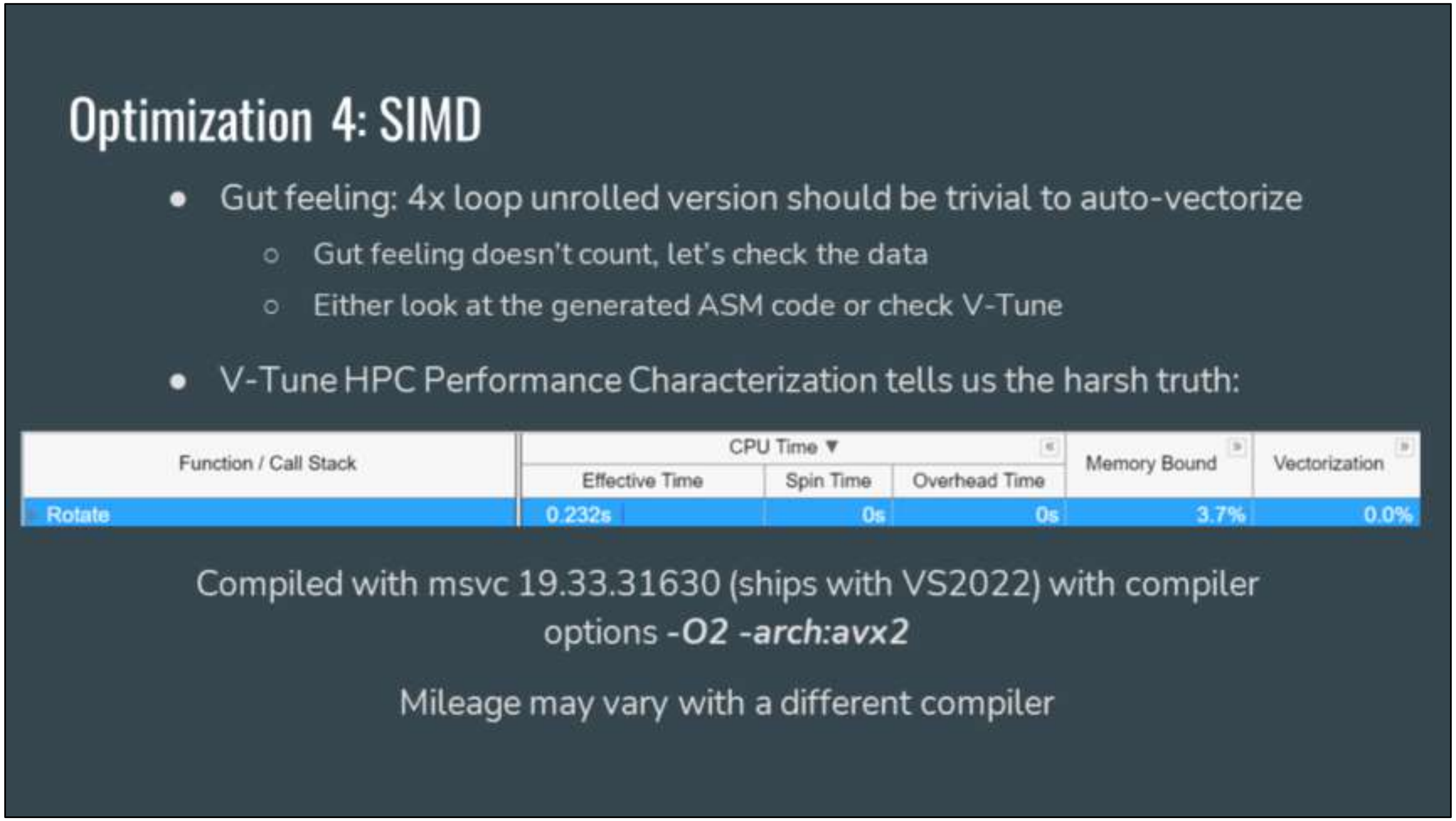


Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize
  - Gut feeling doesn't count, let's check the data
  - Either look at the generated ASM code or check V-Tune
- V-Tune HPC Performance Characterization tells us the harsh truth:

Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn't count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there's zero vectorization in our code. And that's with optimizations enabled on the latest compiler version.



Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize
    - Gut feeling doesn't count, let's check the data
    - Either look at the generated ASM code or check V-Tune
- V-Tune HPC Performance Characterization tells us the harsh truth:

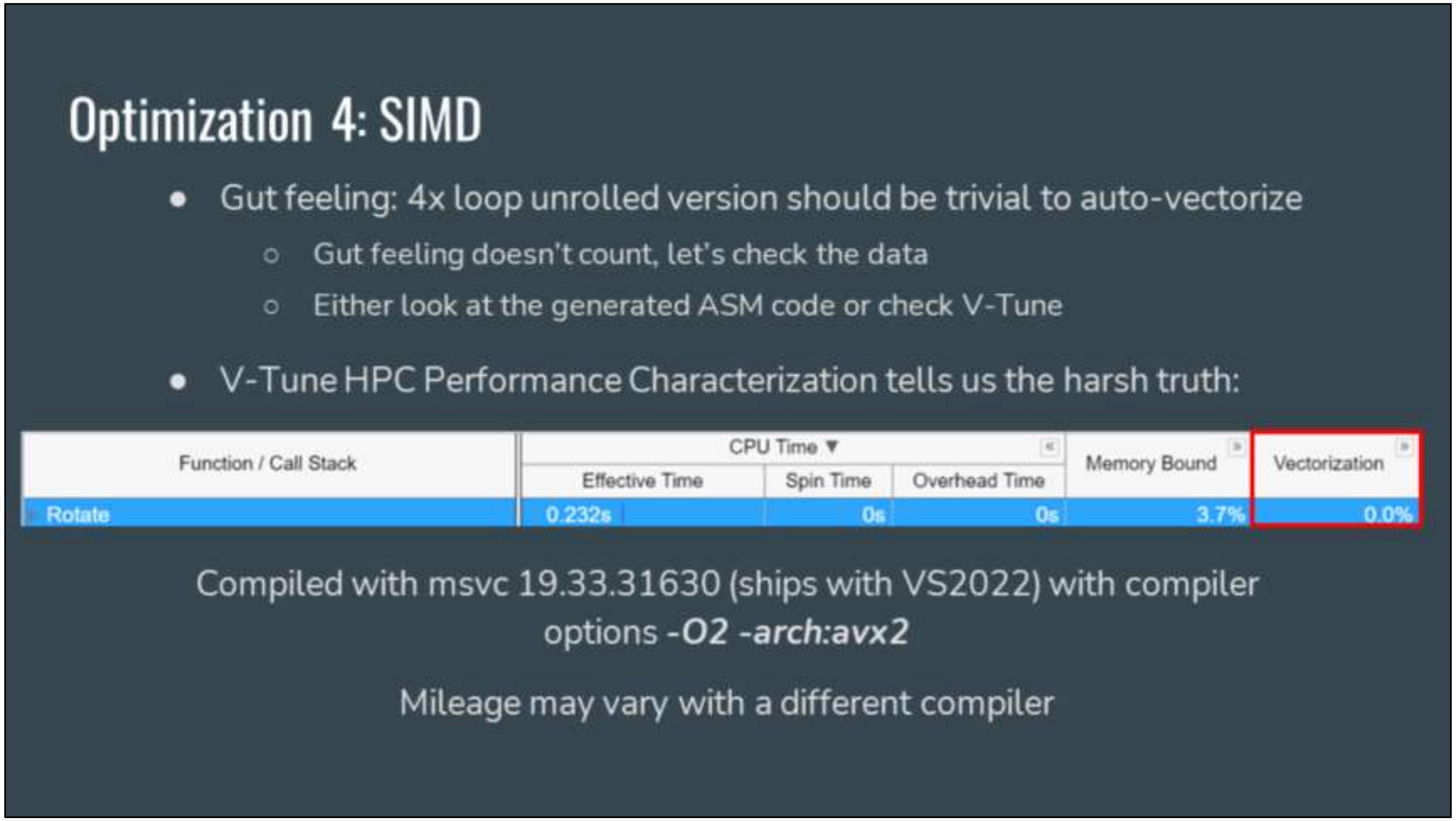
Function / Call Stack	CPU Time ▼			Memory Bound	Vectorization
	Effective Time	Spin Time	Overhead Time		
Rotate	0.232s	0s	0s	3.7%	0.0%

Compiled with msvc 19.33.31630 (ships with VS2022) with compiler options ***-O2 -arch:avx2***

Mileage may vary with a different compiler

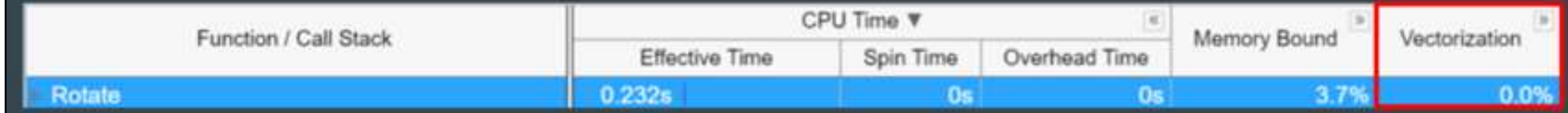
Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn't count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there's zero vectorization in our code. And that's with optimizations enabled on the latest compiler version.



Optimization 4: SIMD

- Gut feeling: 4x loop unrolled version should be trivial to auto-vectorize
  - Gut feeling doesn't count, let's check the data
  - Either look at the generated ASM code or check V-Tune
- V-Tune HPC Performance Characterization tells us the harsh truth:



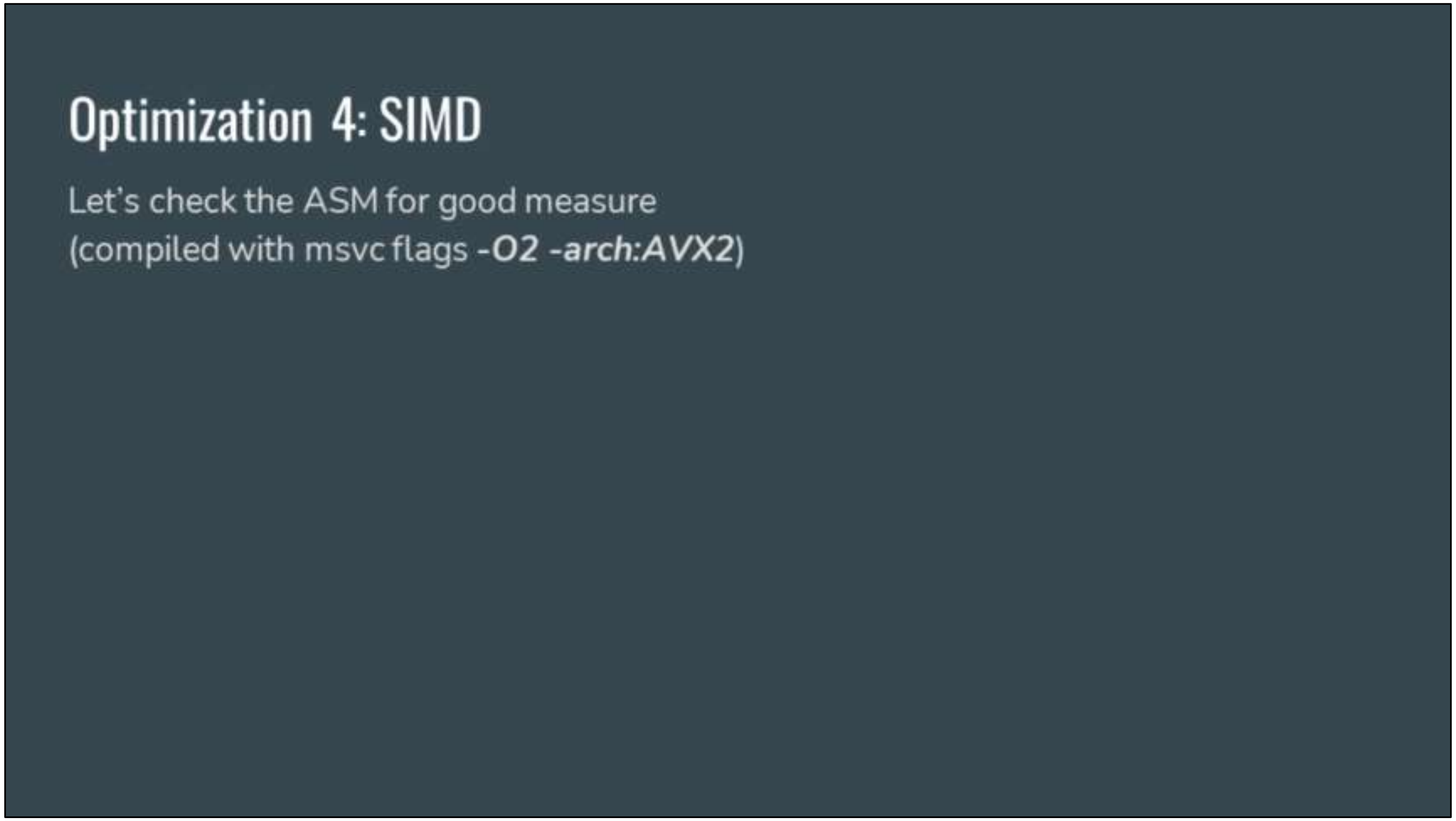
Function / Call Stack	CPU Time ▼			Memory Bound	Vectorization
	Effective Time	Spin Time	Overhead Time		
Rotate	0.232s	0s	0s	3.7%	0.0%

Compiled with msvc 19.33.31630 (ships with VS2022) with compiler options ***-O2 -arch:avx2***

Mileage may vary with a different compiler

Going by my gut feeling, our 4x loop unrolled version of the rotation algorithm should be trivial to auto-vectorize by the compiler. But again, this doesn't count - we need to be sure.
To get verification whether the code got auto vectorize or not, you can either check the generated Assembly code or check the HPC Performance Characterization Analysis in V-Tune.

Checking V-Tune, we see the harsh truth that there's zero vectorization in our code. And that's with optimizations enabled on the latest compiler version.

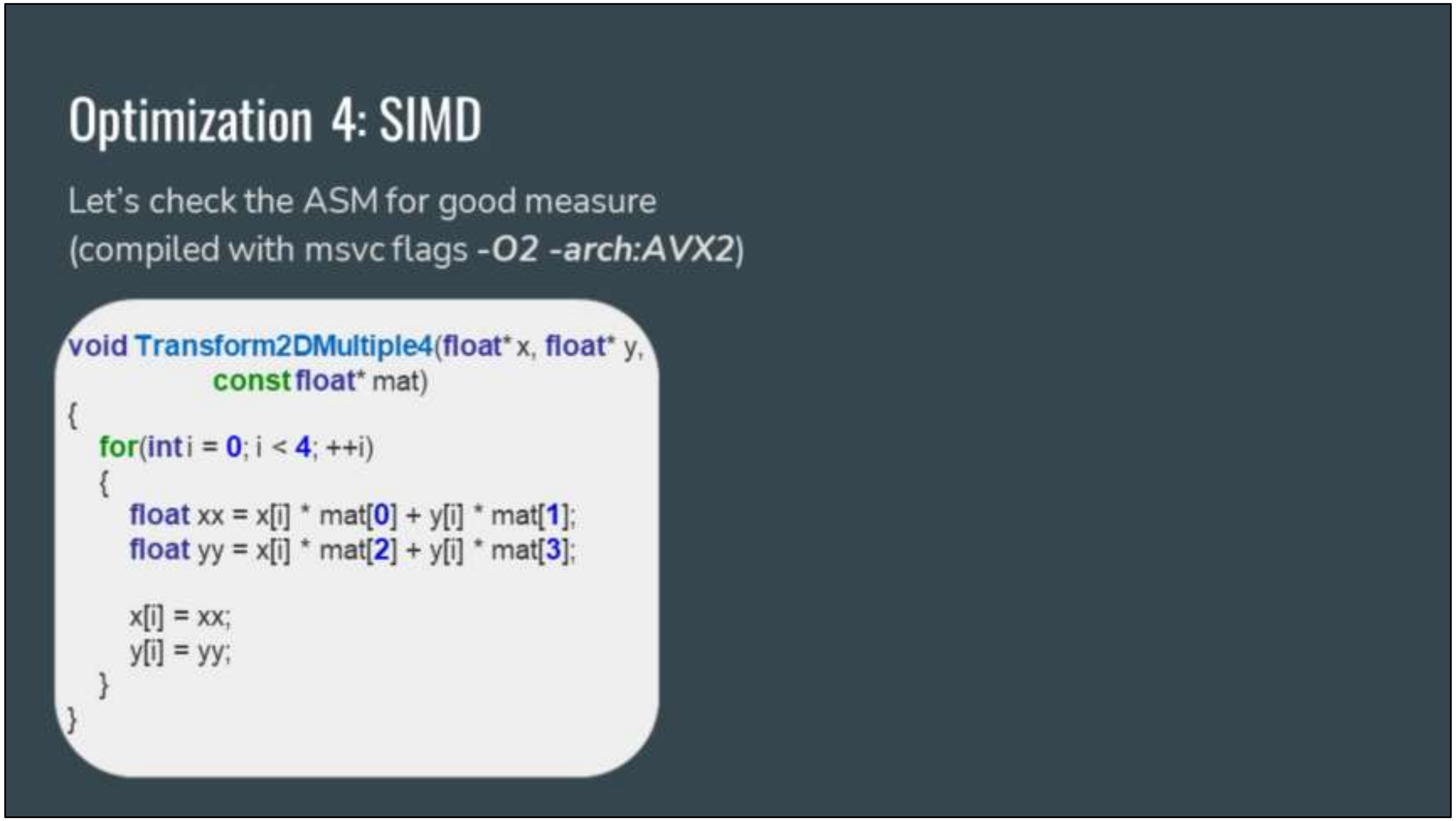


Optimization 4: SIMD

Let’s check the ASM for good measure
(compiled with msvc flags ***-O2 -arch:AVX2***)

But let’s also check the generated assembly code just for good measure.
Here’s another example with 4x loob unrolling.

Using compiler explorer, we can see that the generated instructions are all scalar instructions working on a single element.



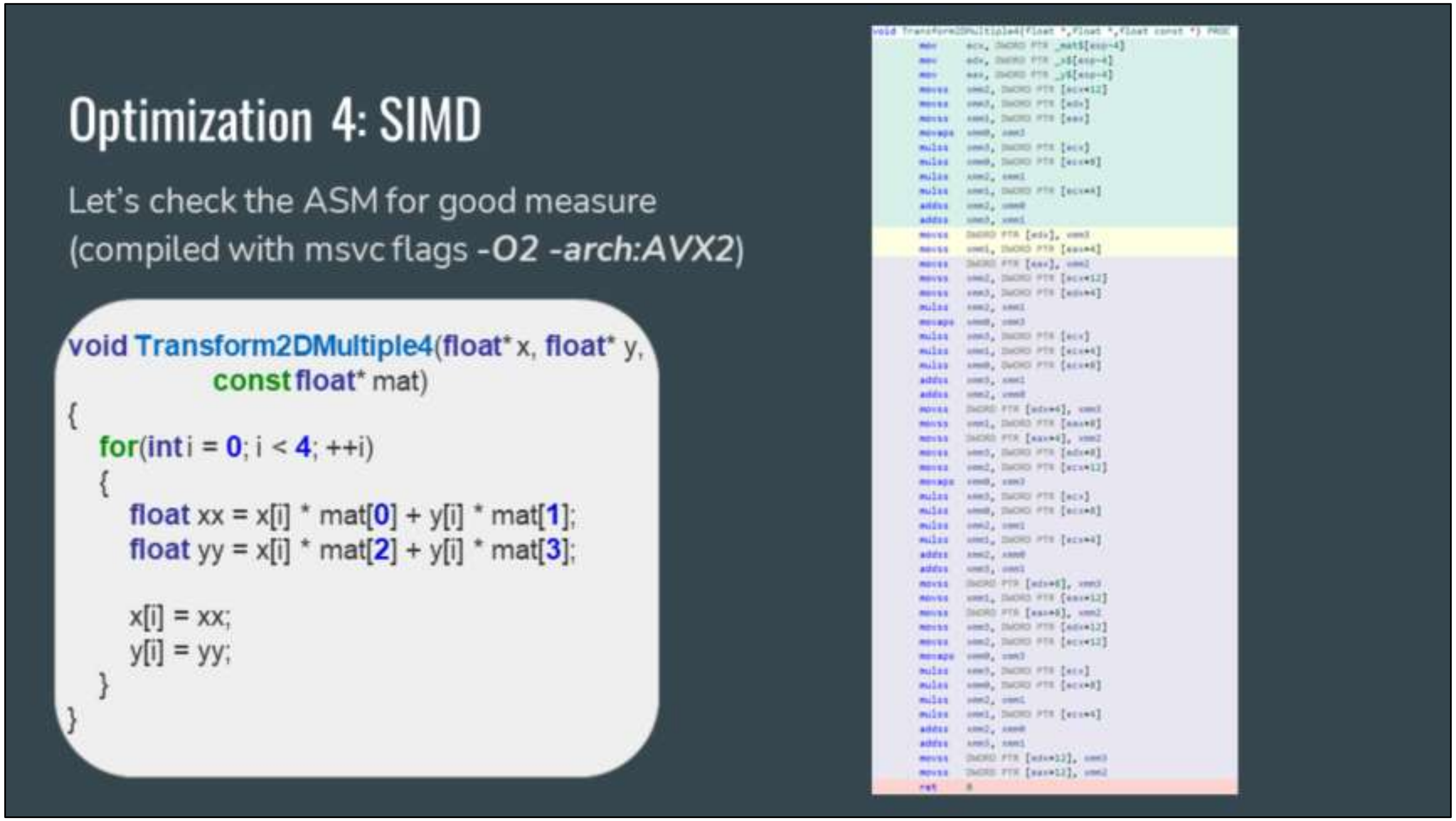
Optimization 4: SIMD

Let's check the ASM for good measure
(compiled with msvc flags *-O2 -arch:AVX2*)

```
void Transform2DMultiple4(float* x, float* y,
              const float* mat)
{
   for(int i = 0; i < 4; ++i)
   {
      float xx = x[i] * mat[0] + y[i] * mat[1];
      float yy = x[i] * mat[2] + y[i] * mat[3];

      x[i] = xx;
      y[i] = yy;
   }
}
```

But let's also check the generated assembly code just for good measure.
Here's another example with 4x loob unrolling.

Using compiler explorer, we can see that the generated instructions are all scalar instructions working on a single element.



Optimization 4: SIMD

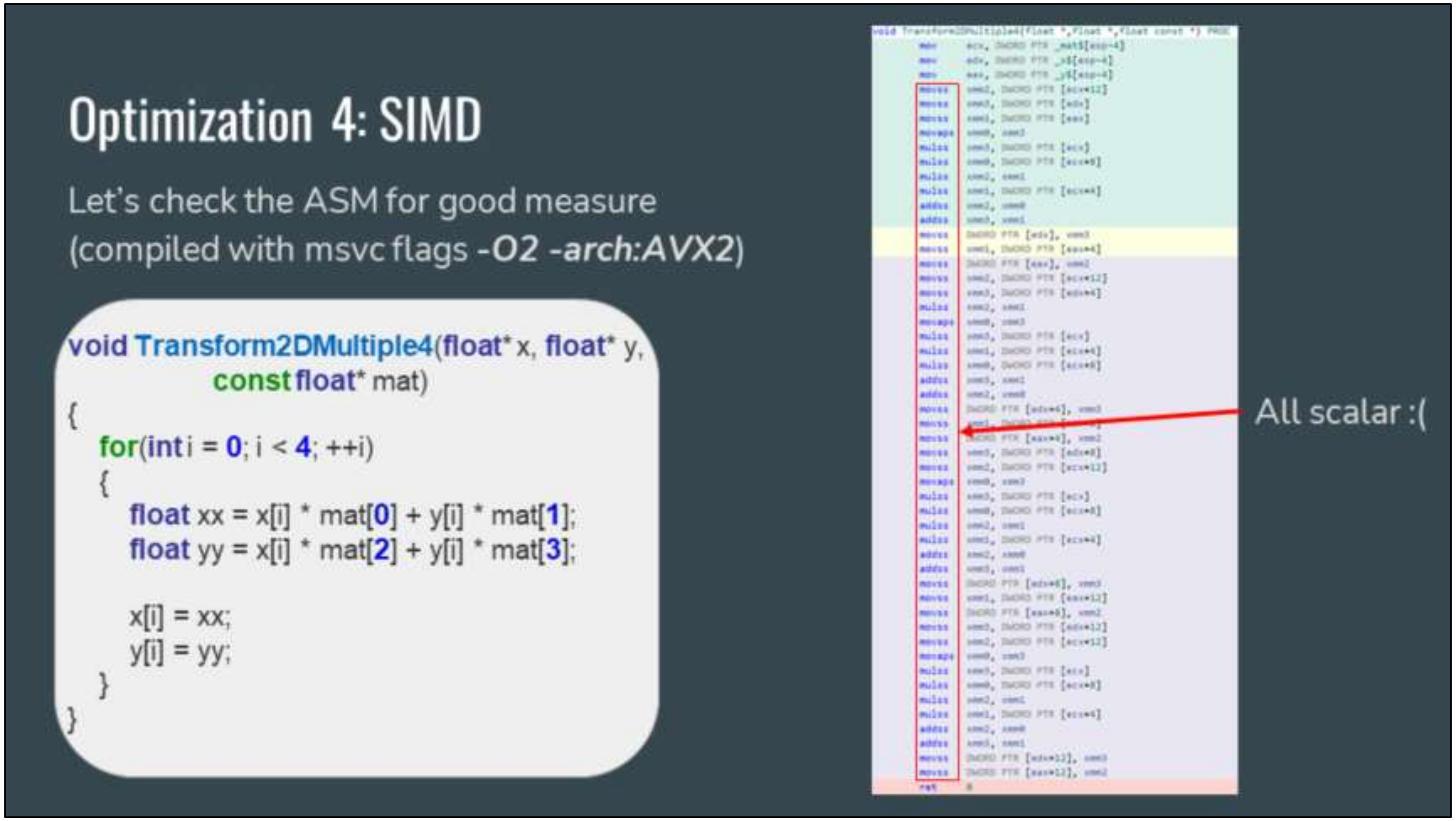
Let's check the ASM for good measure
(compiled with msvc flags *-O2 -arch:AVX2*)

```
void Transform2DMultiple4(float* x, float* y,
                const float* mat)
{
   for(int i = 0; i < 4; ++i)
   {
      float xx = x[i] * mat[0] + y[i] * mat[1];
      float yy = x[i] * mat[2] + y[i] * mat[3];

      x[i] = xx;
      y[i] = yy;
   }
}
```

```
void Transform2DMultiple4(float *,float *,float const *) PROC
        mov     ecx, DWORD PTR _mat$[esp-4]
        mov     edx, DWORD PTR _x$[esp-4]
        mov     eax, DWORD PTR _y$[esp-4]
        movss   xmm2, DWORD PTR [ecx+12]
        movss   xmm3, DWORD PTR [edx]
        movss   xmm1, DWORD PTR [eax]
        movaps  xmm0, xmm3
        mulss   xmm3, DWORD PTR [ecx]
        mulss   xmm0, DWORD PTR [ecx+8]
        mulss   xmm2, xmm1
        mulss   xmm1, DWORD PTR [ecx+4]
        addss   xmm2, xmm0
        addss   xmm3, xmm1
        movss   DWORD PTR [edx], xmm3
        movss   xmm1, DWORD PTR [eax+4]
        movss   DWORD PTR [eax], xmm2
        movss   xmm2, DWORD PTR [ecx+12]
        movss   xmm3, DWORD PTR [edx+4]
        mulss   xmm2, xmm1
        movaps  xmm0, xmm3
        mulss   xmm3, DWORD PTR [ecx]
        mulss   xmm1, DWORD PTR [ecx+4]
        mulss   xmm0, DWORD PTR [ecx+8]
        addss   xmm3, xmm1
        addss   xmm2, xmm0
        movss   DWORD PTR [edx+4], xmm3
        movss   xmm1, DWORD PTR [eax+8]
        movss   DWORD PTR [eax+4], xmm2
        movss   xmm3, DWORD PTR [edx+8]
        movss   xmm2, DWORD PTR [ecx+12]
        movaps  xmm0, xmm3
        mulss   xmm3, DWORD PTR [ecx]
        mulss   xmm0, DWORD PTR [ecx+8]
        mulss   xmm2, xmm1
        mulss   xmm1, DWORD PTR [ecx+4]
        addss   xmm2, xmm0
        addss   xmm3, xmm1
        movss   DWORD PTR [edx+8], xmm3
        movss   xmm1, DWORD PTR [eax+12]
        movss   DWORD PTR [eax+8], xmm2
        movss   xmm3, DWORD PTR [edx+12]
        movss   xmm2, DWORD PTR [ecx+12]
        movaps  xmm0, xmm3
        mulss   xmm3, DWORD PTR [ecx]
        mulss   xmm0, DWORD PTR [ecx+8]
        mulss   xmm2, xmm1
        mulss   xmm1, DWORD PTR [ecx+4]
        addss   xmm2, xmm0
        addss   xmm3, xmm1
        movss   DWORD PTR [edx+12], xmm3
        movss   DWORD PTR [eax+12], xmm2
        ret     0
```

But let's also check the generated assembly code just for good measure.
Here's another example with 4x loob unrolling.

Using compiler explorer, we can see that the generated instructions are all scalar instructions working on a single element.



Optimization 4: SIMD

Let's check the ASM for good measure
(compiled with msvc flags *-O2 -arch:AVX2*)

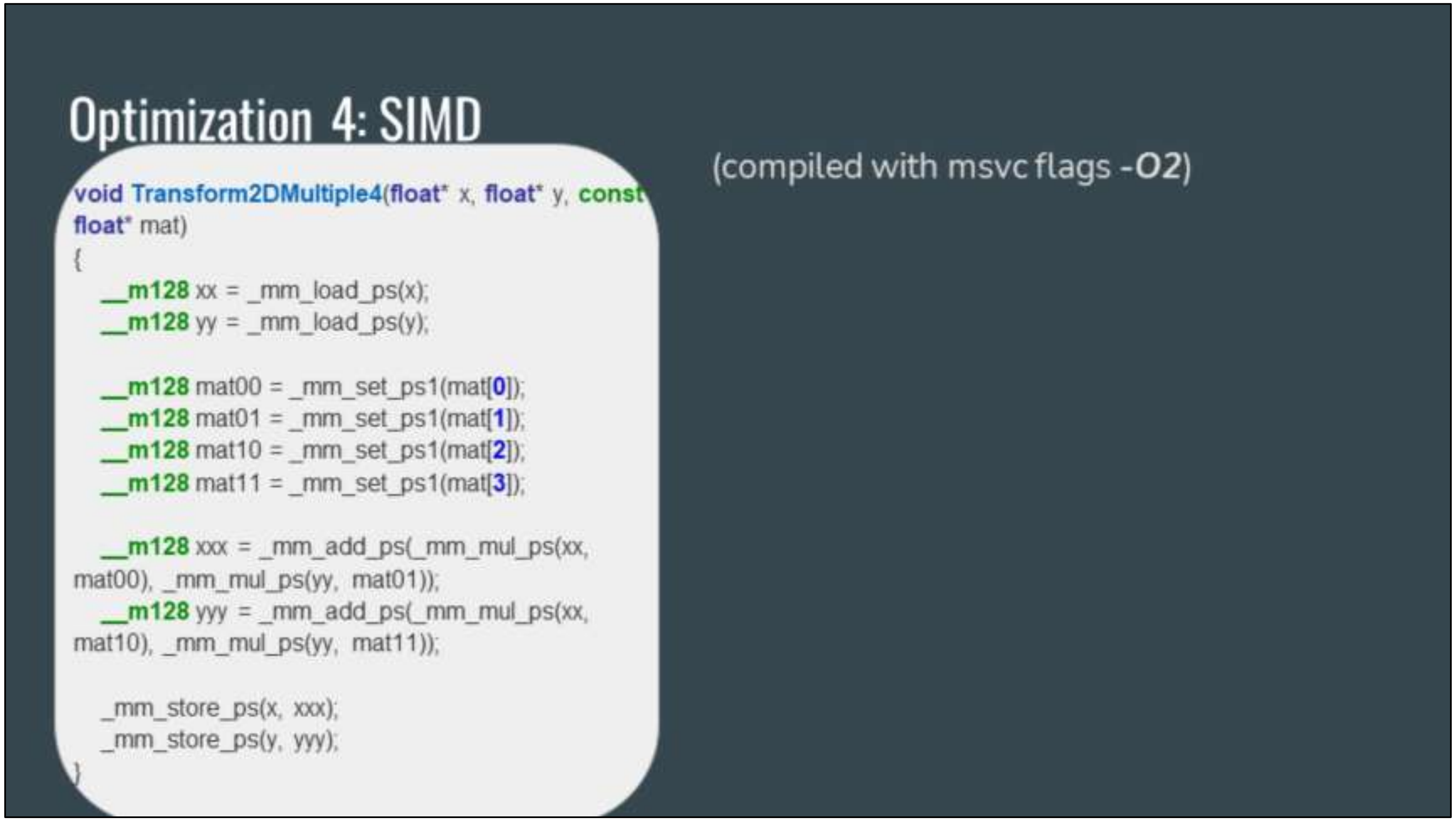
```cpp
void Transform2DMultiple4(float* x, float* y,
                const float* mat)
{
   for(int i = 0; i < 4; ++i)
   {
      float xx = x[i] * mat[0] + y[i] * mat[1];
      float yy = x[i] * mat[2] + y[i] * mat[3];

      x[i] = xx;
      y[i] = yy;
   }
}
```

```
void Transform2DMultiple4(float *,float *,float const *) PROC
        mov      ecx, DWORD PTR _mat$[esp-4]
        mov      edx, DWORD PTR _x$[esp-4]
        mov      eax, DWORD PTR _y$[esp-4]
        movss    xmm2, DWORD PTR [ecx+12]
        movss    xmm3, DWORD PTR [edx]
        movss    xmm1, DWORD PTR [eax]
        movaps   xmm0, xmm3
        mulss    xmm3, DWORD PTR [ecx]
        mulss    xmm0, DWORD PTR [ecx+8]
        mulss    xmm2, xmm1
        mulss    xmm1, DWORD PTR [ecx+4]
        addss    xmm2, xmm0
        addss    xmm3, xmm1
        movss    DWORD PTR [edx], xmm3
        movss    xmm1, DWORD PTR [eax+4]
        movss    DWORD PTR [eax], xmm2
        movss    xmm2, DWORD PTR [ecx+12]
        movss    xmm3, DWORD PTR [edx+4]
        mulss    xmm2, xmm1
        movaps   xmm0, xmm3
        mulss    xmm3, DWORD PTR [ecx]
        mulss    xmm1, DWORD PTR [ecx+4]
        mulss    xmm0, DWORD PTR [ecx+8]
        addss    xmm3, xmm1
        addss    xmm2, xmm0
        movss    DWORD PTR [edx+4], xmm3
        movss    xmm1, DWORD PTR [eax+8]
        movss    DWORD PTR [eax+4], xmm2
        movss    xmm3, DWORD PTR [edx+8]
        movss    xmm2, DWORD PTR [ecx+12]
        movaps   xmm0, xmm3
        mulss    xmm3, DWORD PTR [ecx]
        mulss    xmm0, DWORD PTR [ecx+8]
        mulss    xmm2, xmm1
        mulss    xmm1, DWORD PTR [ecx+4]
        addss    xmm2, xmm0
        addss    xmm3, xmm1
        movss    DWORD PTR [edx+8], xmm3
        movss    xmm1, DWORD PTR [eax+12]
        movss    DWORD PTR [eax+8], xmm2
        movss    xmm3, DWORD PTR [edx+12]
        movss    xmm2, DWORD PTR [ecx+12]
        movaps   xmm0, xmm3
        mulss    xmm3, DWORD PTR [ecx]
        mulss    xmm0, DWORD PTR [ecx+8]
        mulss    xmm2, xmm1
        mulss    xmm1, DWORD PTR [ecx+4]
        addss    xmm2, xmm0
        addss    xmm3, xmm1
        movss    DWORD PTR [edx+12], xmm3
        movss    DWORD PTR [eax+12], xmm2
        ret      0
```

All scalar :(

But let's also check the generated assembly code just for good measure.
Here's another example with 4x loob unrolling.

Using compiler explorer, we can see that the generated instructions are all scalar instructions working on a single element.

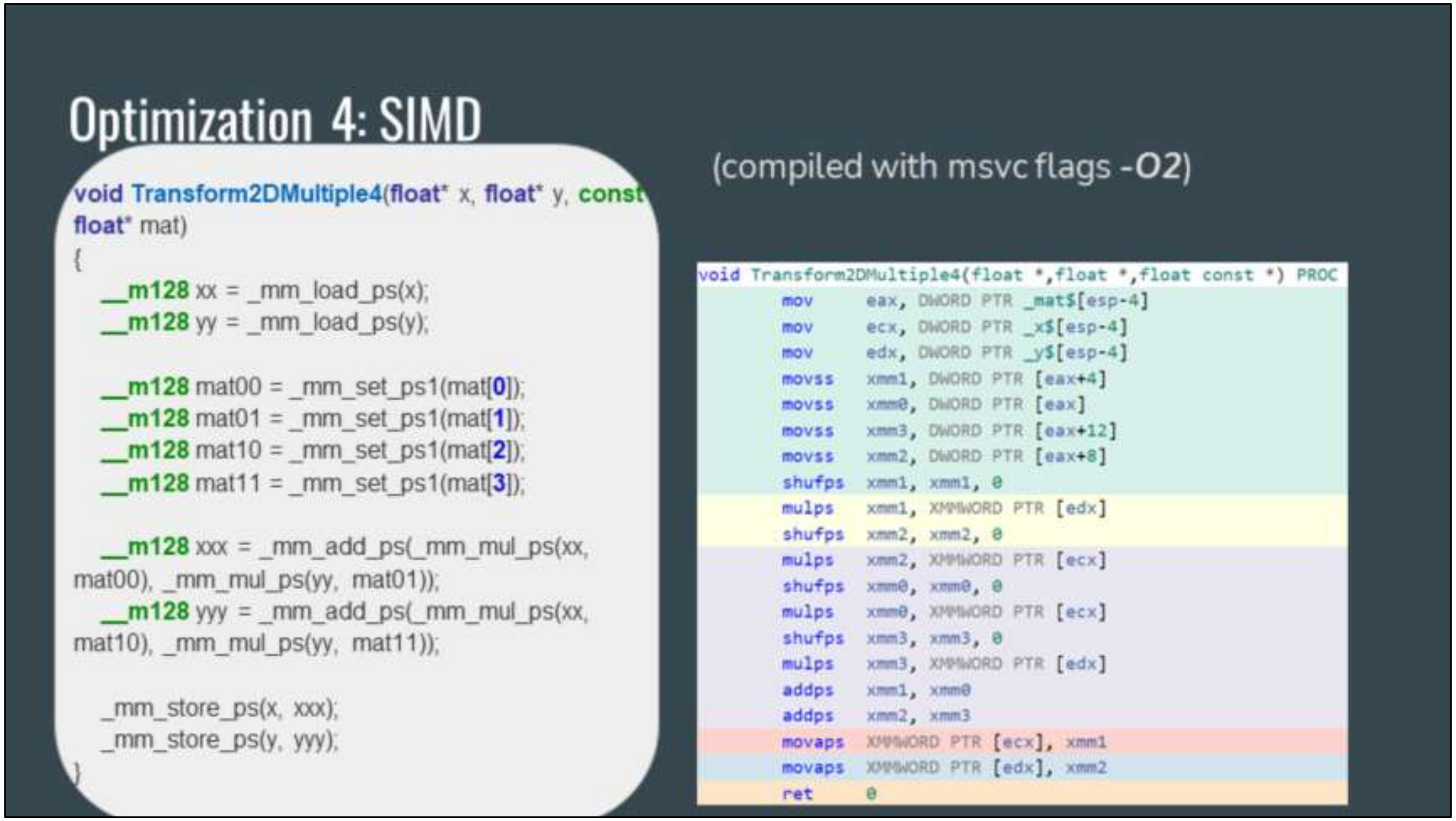


Optimization 4: SIMD

(compiled with msvc flags **-O2**)

```
void Transform2DMultiple4(float* x, float* y, const
float* mat)
{
   __m128 xx = _mm_load_ps(x);
   __m128 yy = _mm_load_ps(y);

   __m128 mat00 = _mm_set_ps1(mat[0]);
   __m128 mat01 = _mm_set_ps1(mat[1]);
   __m128 mat10 = _mm_set_ps1(mat[2]);
   __m128 mat11 = _mm_set_ps1(mat[3]);

   __m128 xxx = _mm_add_ps(_mm_mul_ps(xx,
mat00), _mm_mul_ps(yy, mat01));
   __m128 yyy = _mm_add_ps(_mm_mul_ps(xx,
mat10), _mm_mul_ps(yy, mat11));

   _mm_store_ps(x, xxx);
   _mm_store_ps(y, yyy);
}
```

If we hand-roll the SIMD version using SSE2 intrinsics we see that this generated the vectorized code.



Optimization 4: SIMD

(compiled with msvc flags -O2)

```
void Transform2DMultiple4(float* x, float* y, const
float* mat)
{
    __m128 xx = _mm_load_ps(x);
    __m128 yy = _mm_load_ps(y);

    __m128 mat00 = _mm_set_ps1(mat[0]);
    __m128 mat01 = _mm_set_ps1(mat[1]);
    __m128 mat10 = _mm_set_ps1(mat[2]);
    __m128 mat11 = _mm_set_ps1(mat[3]);

    __m128 xxx = _mm_add_ps(_mm_mul_ps(xx,
mat00), _mm_mul_ps(yy, mat01));
    __m128 yyy = _mm_add_ps(_mm_mul_ps(xx,
mat10), _mm_mul_ps(yy, mat11));

    _mm_store_ps(x, xxx);
    _mm_store_ps(y, yyy);
}
```

```
void Transform2DMultiple4(float *,float *,float const *) PROC
        mov     eax, DWORD PTR _mat$[esp-4]
        mov     ecx, DWORD PTR _x$[esp-4]
        mov     edx, DWORD PTR _y$[esp-4]
        movss   xmm1, DWORD PTR [eax+4]
        movss   xmm0, DWORD PTR [eax]
        movss   xmm3, DWORD PTR [eax+12]
        movss   xmm2, DWORD PTR [eax+8]
        shufps  xmm1, xmm1, 0
        mulps   xmm1, XMMWORD PTR [edx]
        shufps  xmm2, xmm2, 0
        mulps   xmm2, XMMWORD PTR [ecx]
        shufps  xmm0, xmm0, 0
        mulps   xmm0, XMMWORD PTR [ecx]
        shufps  xmm3, xmm3, 0
        mulps   xmm3, XMMWORD PTR [edx]
        addps   xmm1, xmm0
        addps   xmm2, xmm3
        movaps  XMMWORD PTR [ecx], xmm1
        movaps  XMMWORD PTR [edx], xmm2
        ret     0
```

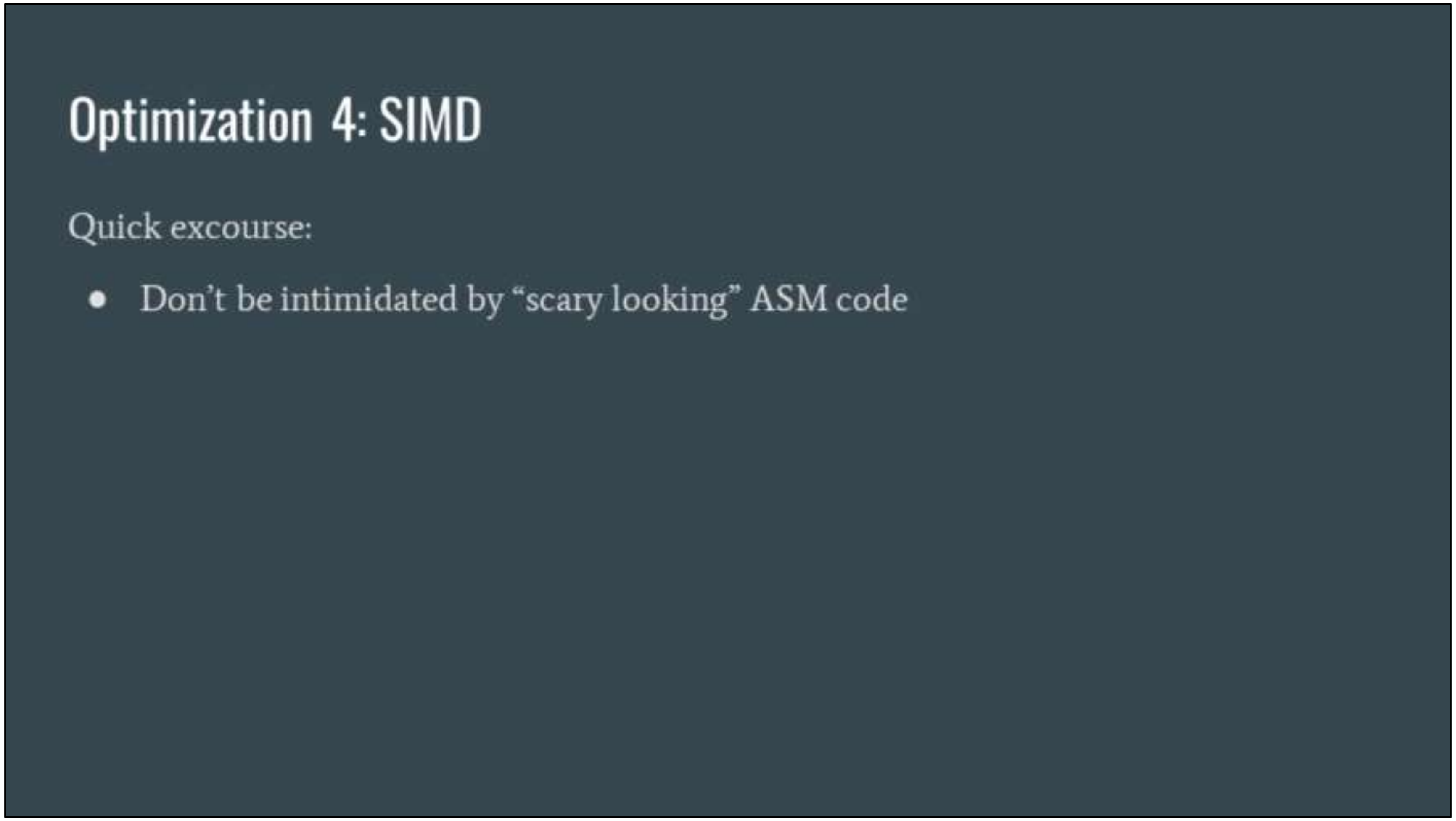
If we hand-roll the SIMD version using SSE2 intrinsics we see that this generated the vectorized code.

Optimization 4: SIMD

Quick excourse:

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don’t make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



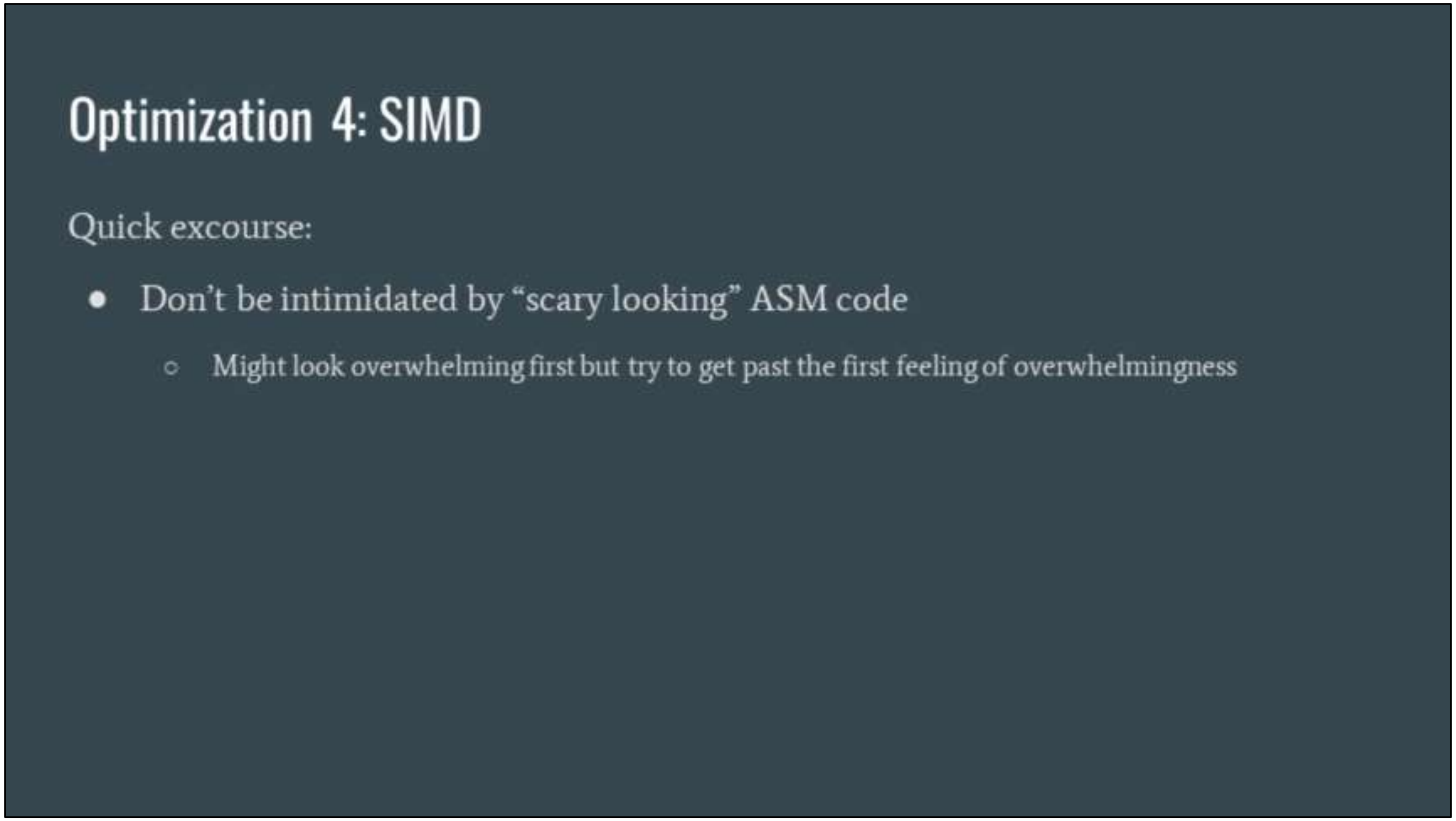
Optimization 4: SIMD

Quick excourse:

- Don’t be intimidated by “scary looking” ASM code

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don’t make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



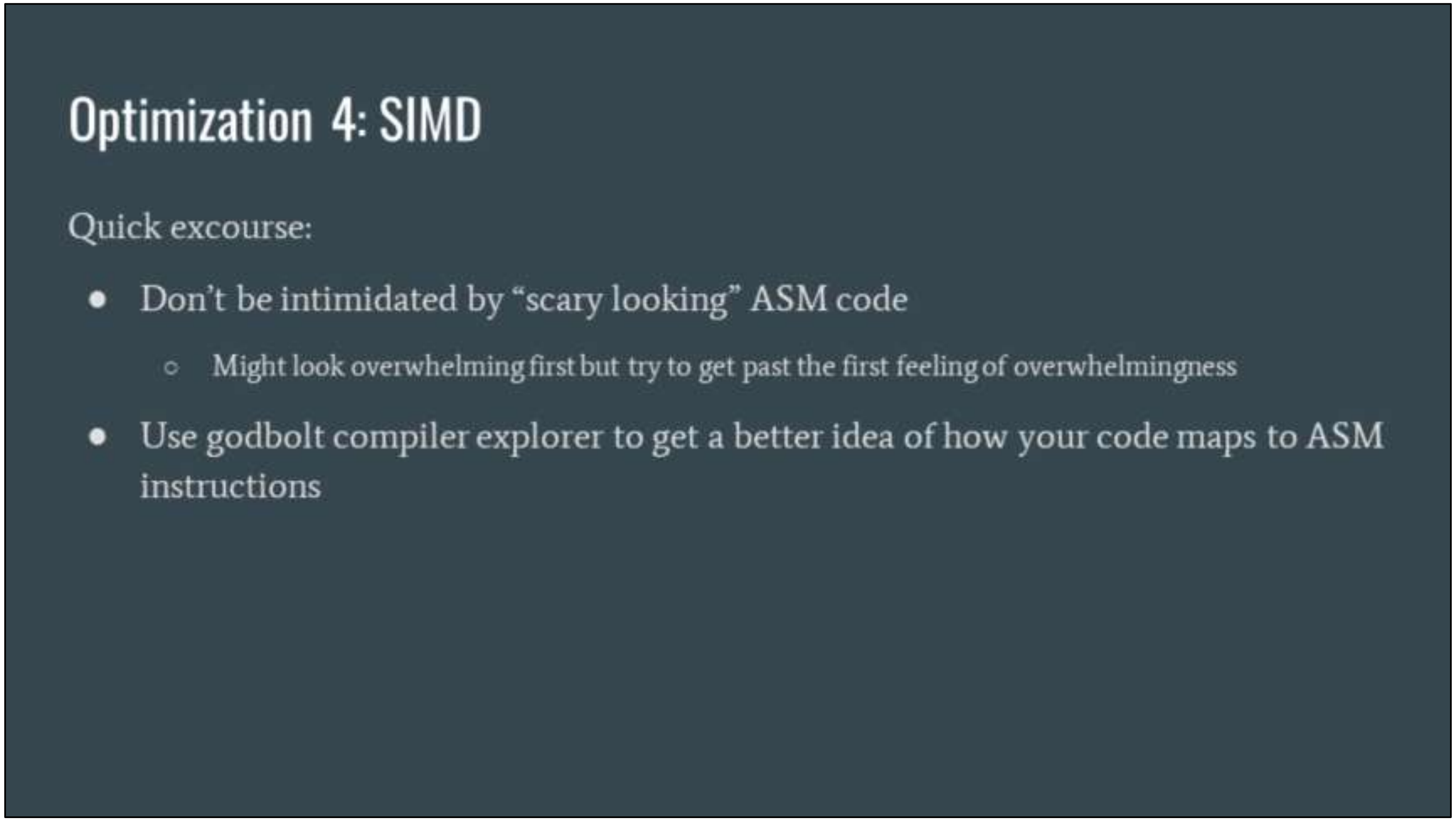
Optimization 4: SIMD

Quick excourse:

- Don’t be intimidated by “scary looking” ASM code
  - Might look overwhelming first but try to get past the first feeling of overwhelmingness

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don’t make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



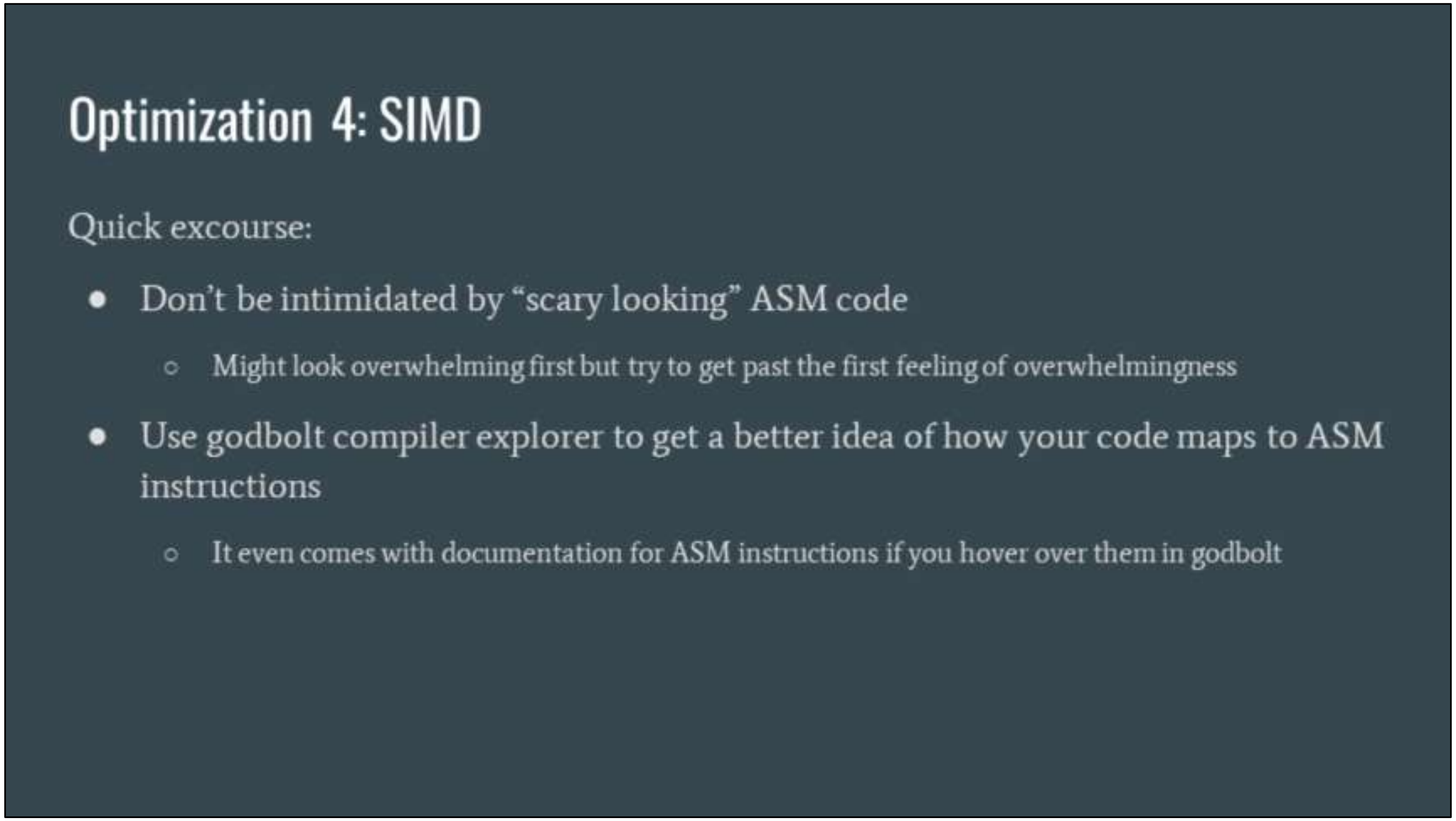
Optimization 4: SIMD

Quick excourse:

- Don't be intimidated by "scary looking" ASM code
  - Might look overwhelming first but try to get past the first feeling of overwhelmingness
- Use godbolt compiler explorer to get a better idea of how your code maps to ASM instructions

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don't make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



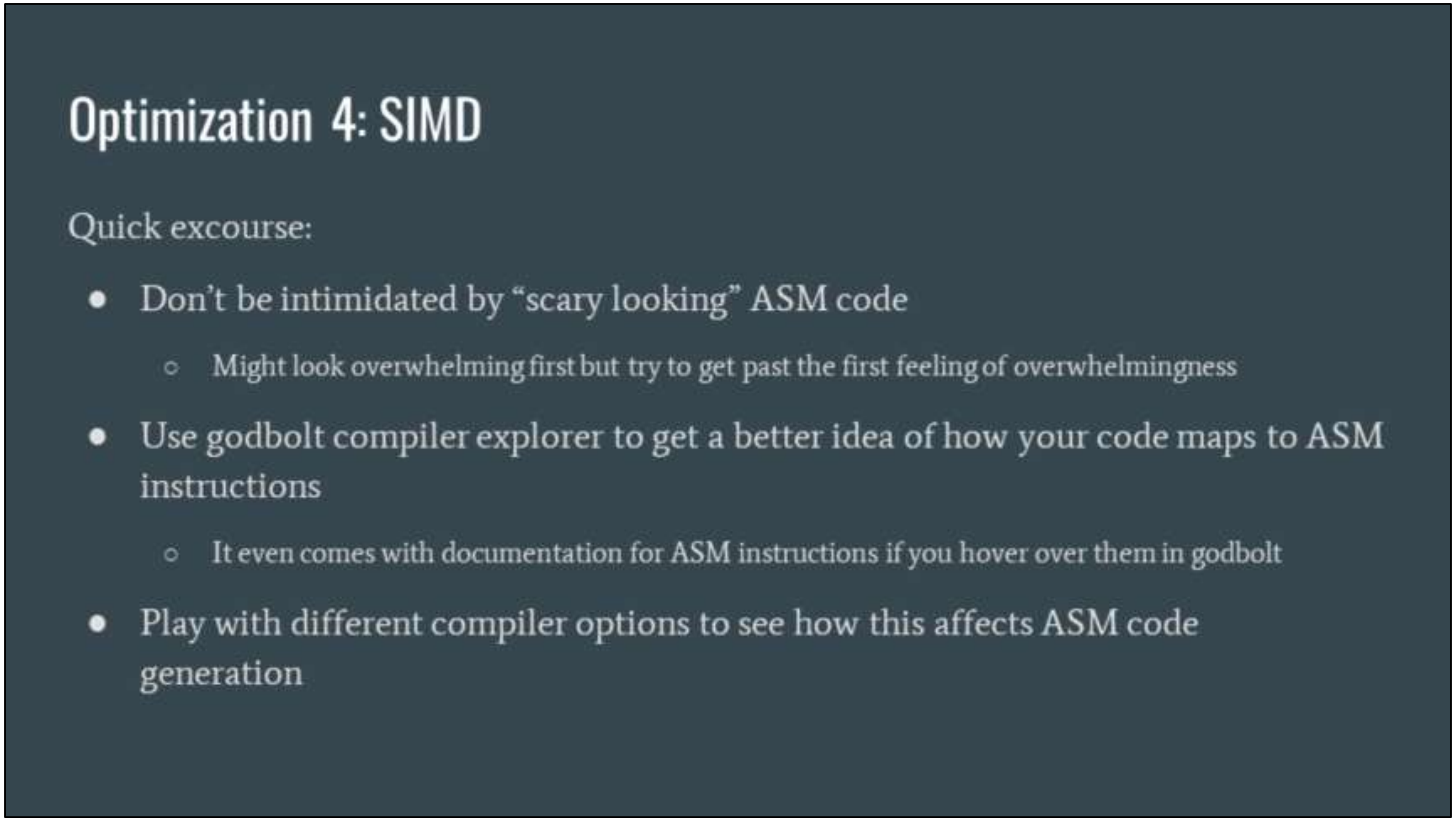
Optimization 4: SIMD

Quick excourse:

- Don’t be intimidated by “scary looking” ASM code
  - Might look overwhelming first but try to get past the first feeling of overwhelmingness
- Use godbolt compiler explorer to get a better idea of how your code maps to ASM instructions
  - It even comes with documentation for ASM instructions if you hover over them in godbolt

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don’t make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



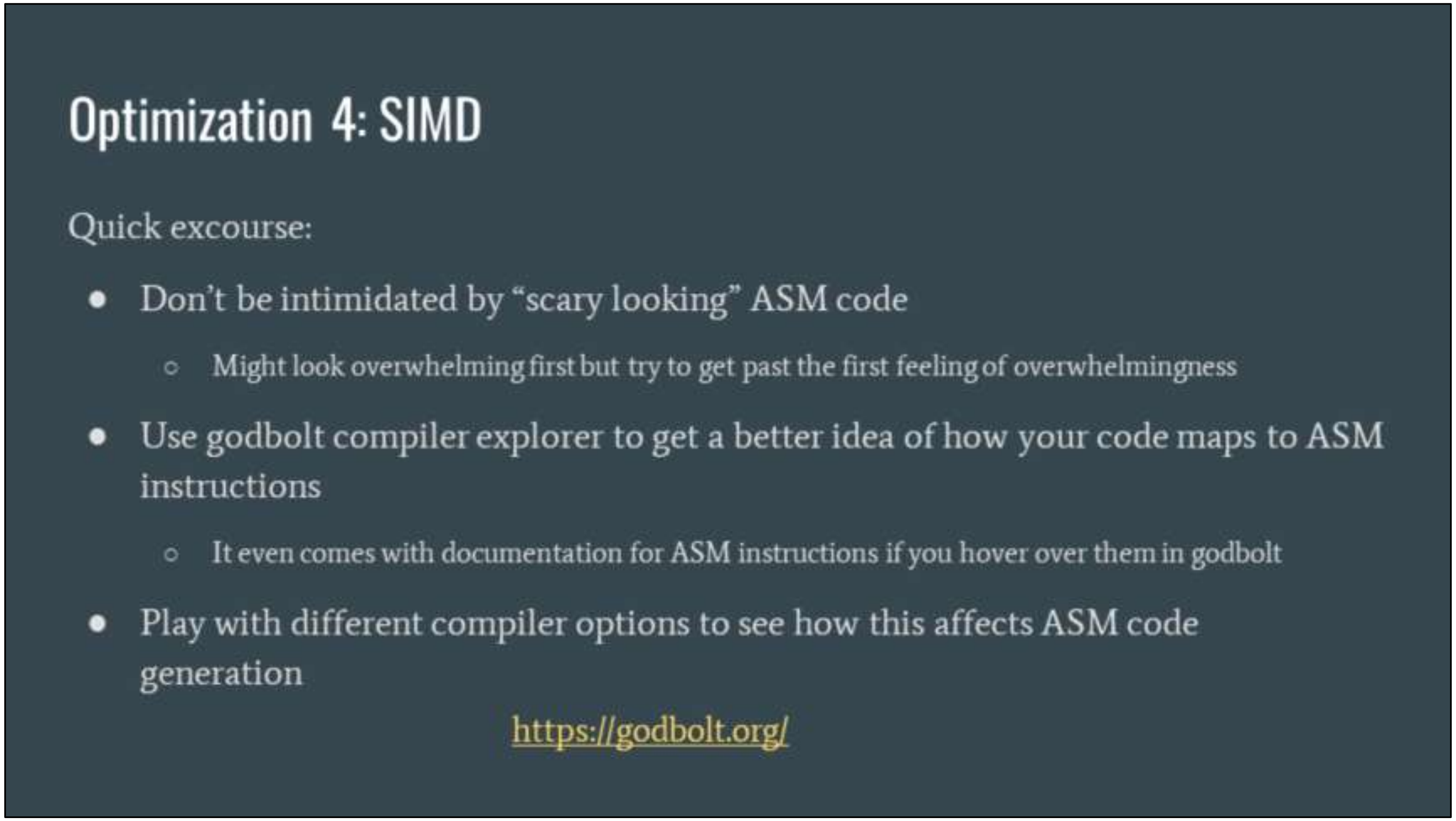
Optimization 4: SIMD

Quick excourse:

- Don't be intimidated by "scary looking" ASM code
  - Might look overwhelming first but try to get past the first feeling of overwhelmingness
- Use godbolt compiler explorer to get a better idea of how your code maps to ASM instructions
  - It even comes with documentation for ASM instructions if you hover over them in godbolt
- Play with different compiler options to see how this affects ASM code generation

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don't make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



Optimization 4: SIMD

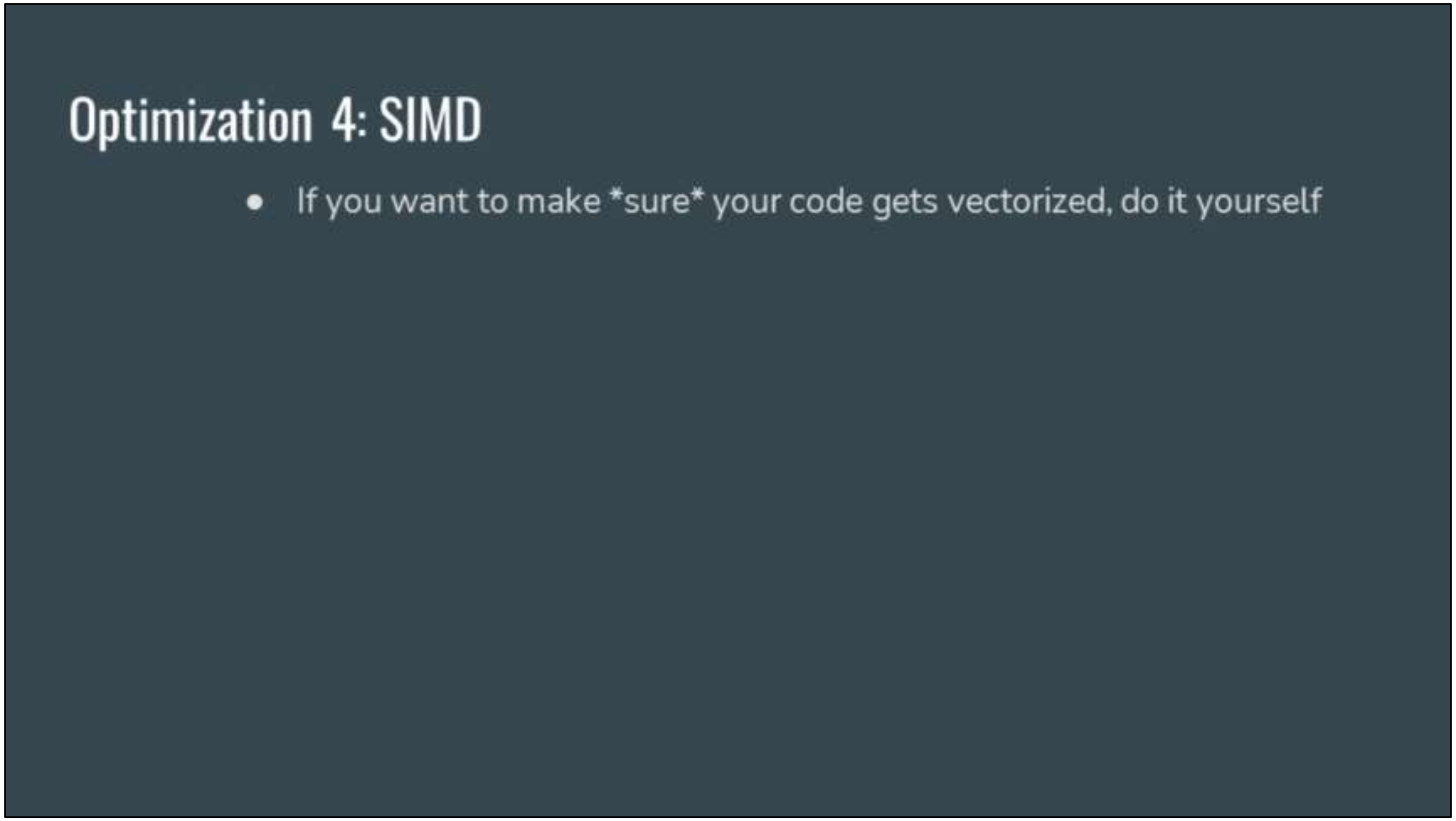
Quick excourse:

- Don’t be intimidated by “scary looking” ASM code
  - Might look overwhelming first but try to get past the first feeling of overwhelmingness
- Use godbolt compiler explorer to get a better idea of how your code maps to ASM instructions
  - It even comes with documentation for ASM instructions if you hover over them in godbolt
- Play with different compiler options to see how this affects ASM code generation

https://godbolt.org/

Just a quick cool-down: Yes, we just saw assembly code and yes, I know that is can be intimidating if you tried looking at assembly before and were overwhelmed.
It can be overwhelming at first because you see lots of things that don’t make sense yet. Try to get past this first feeling of overwhelmingness and try to understand what is happening on the assembly level. This is invaluable and the tools we have today make this not as painful as it used to be.

Especially compiler explorer is a tool I want to highlight here. This is a tool were you can compile soure code using different compilers and examine the generated assembly code.



Optimization 4: SIMD

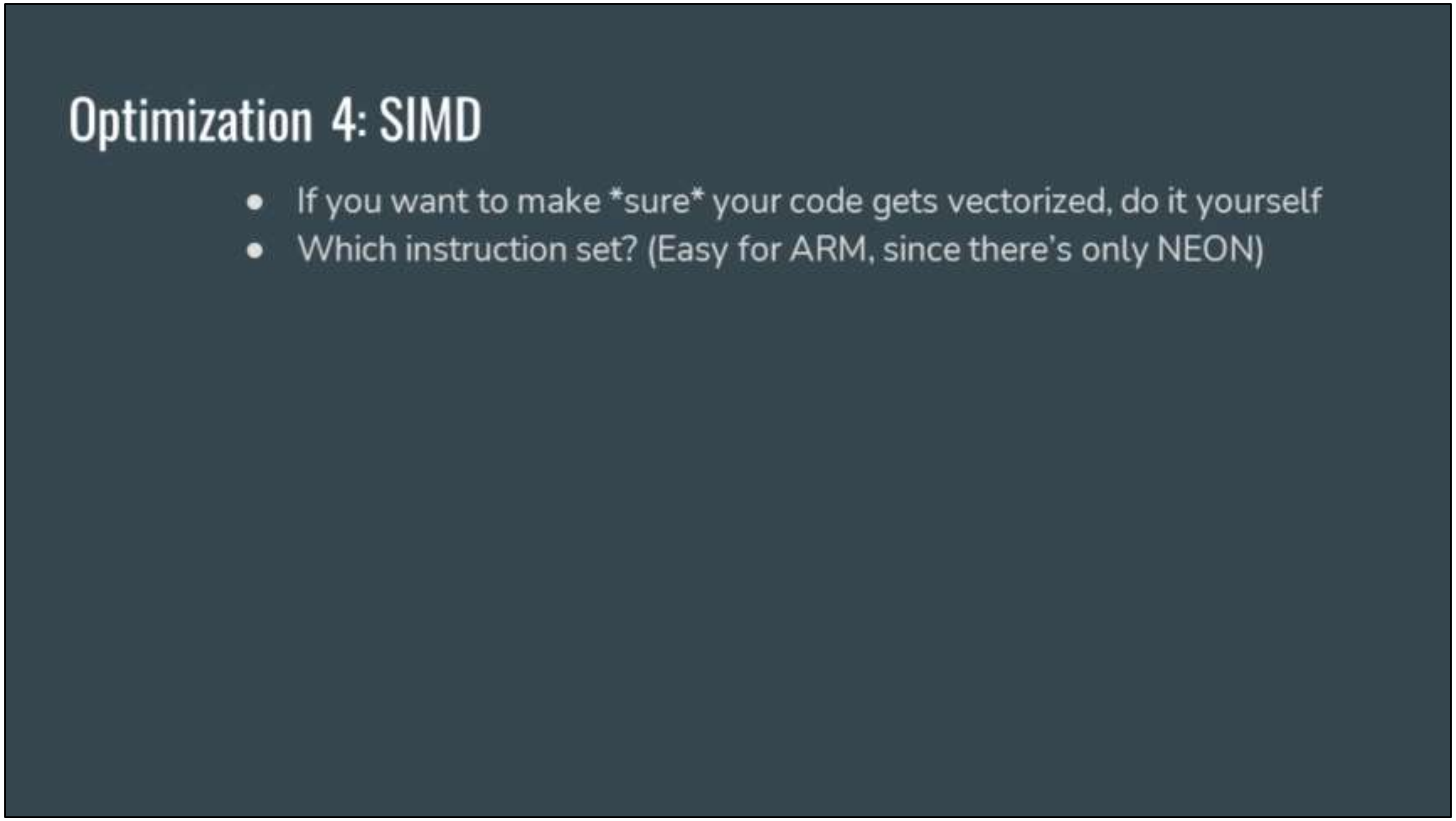
- If you want to make *sure* your code gets vectorized, do it yourself

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there’s only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you’re targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

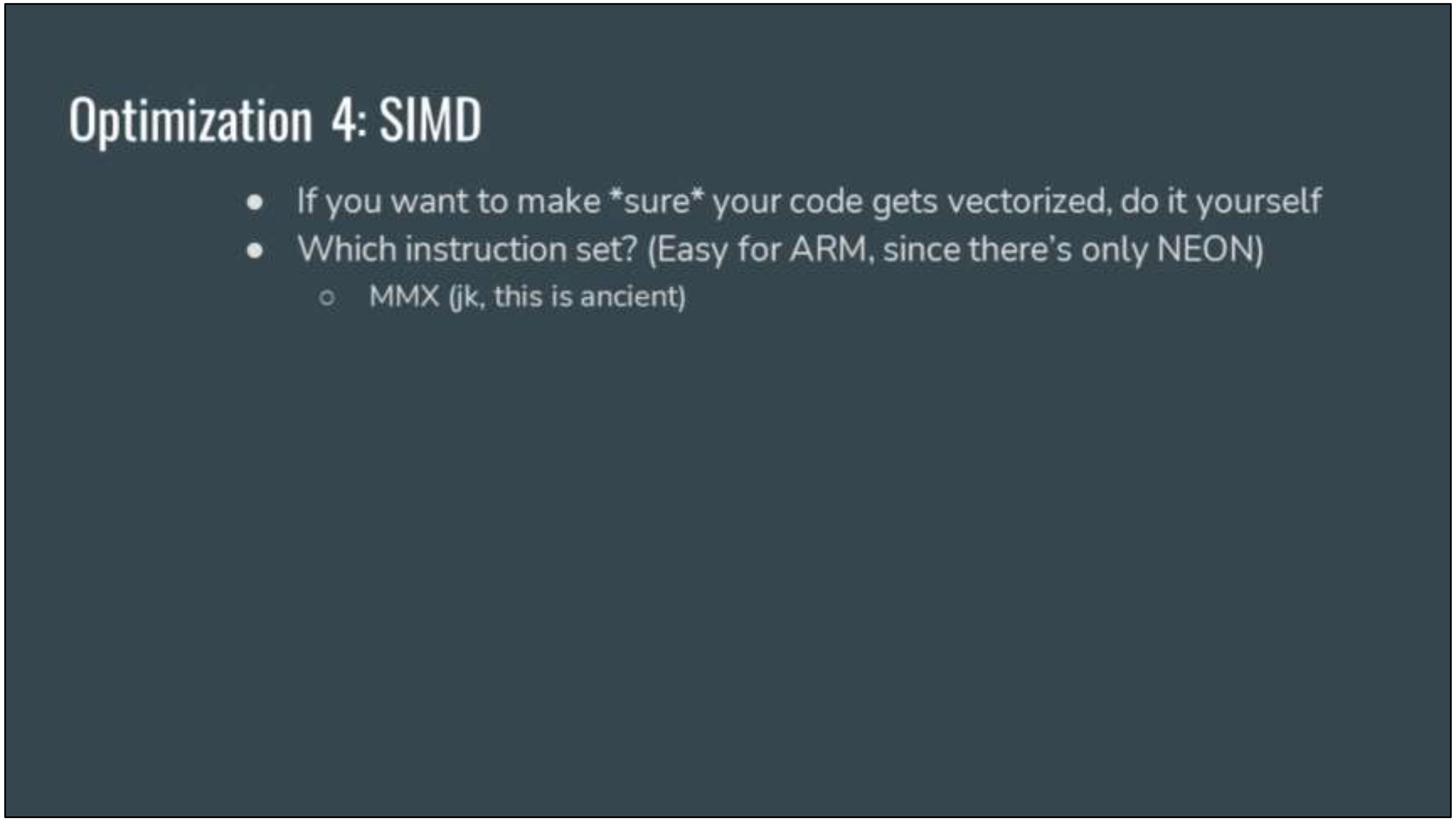
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

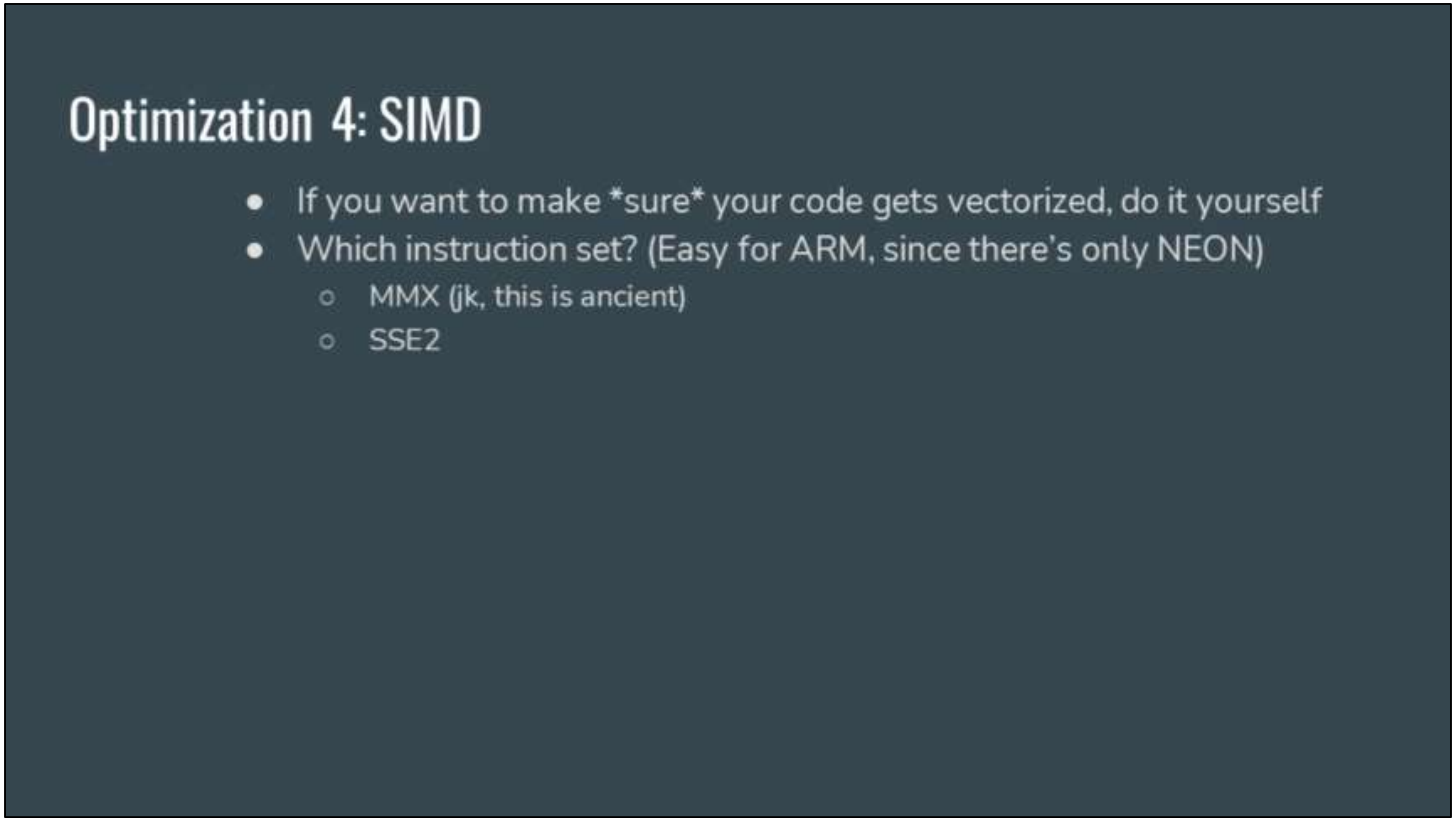
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

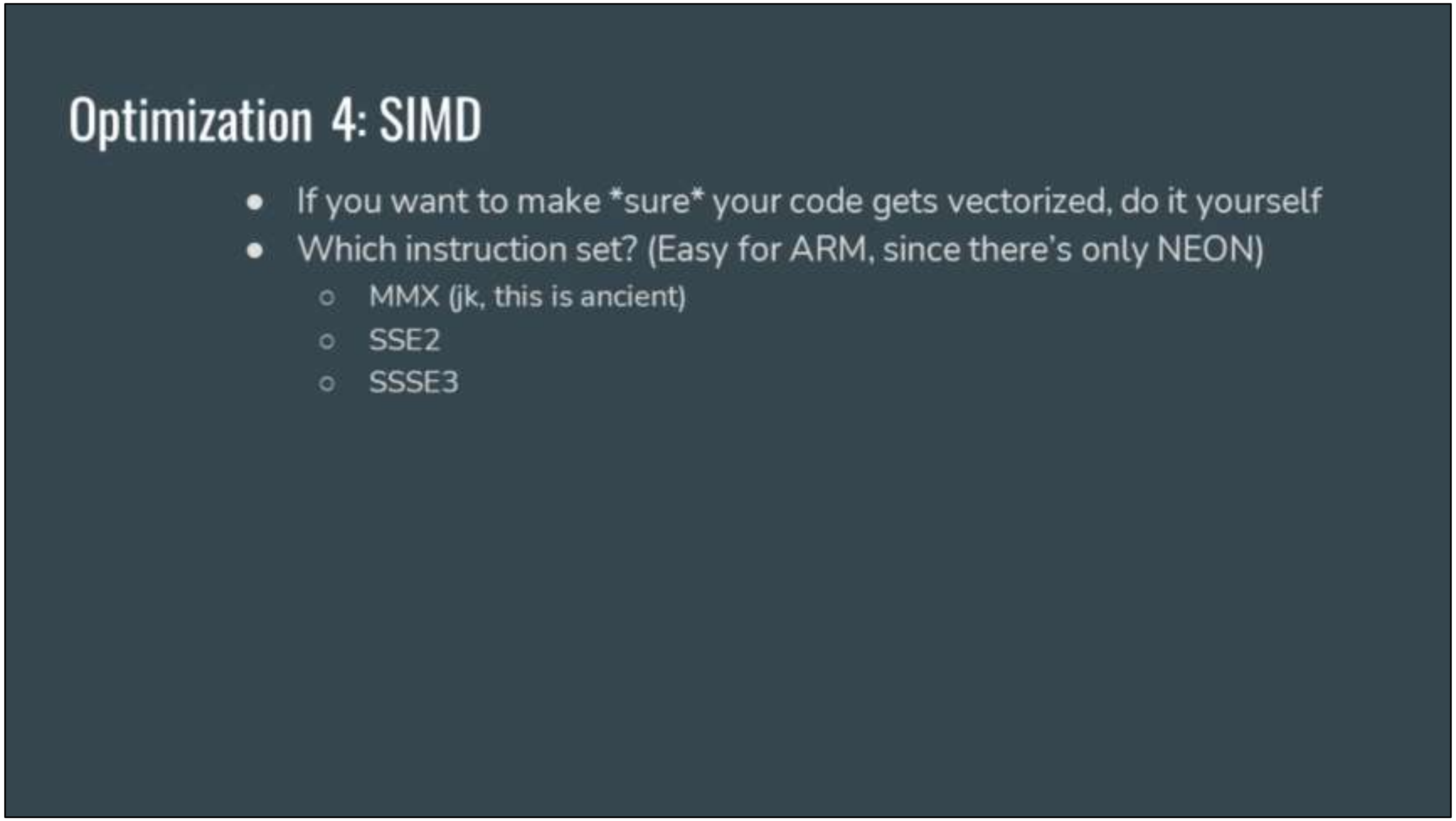
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.


Optimization 4: SIMD

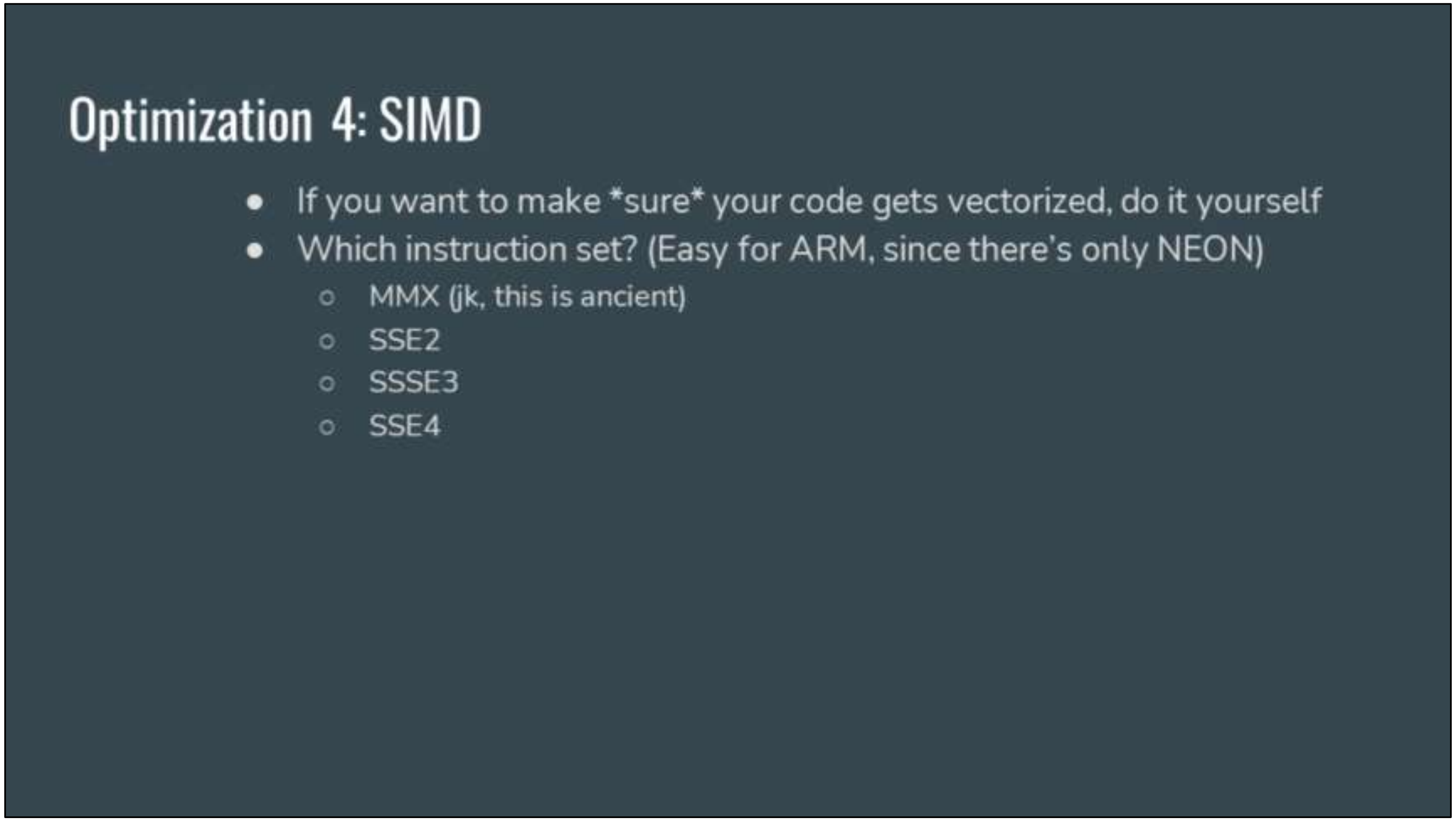
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
    - MMX (jk, this is ancient)
    - SSE2
    - SSSE3

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

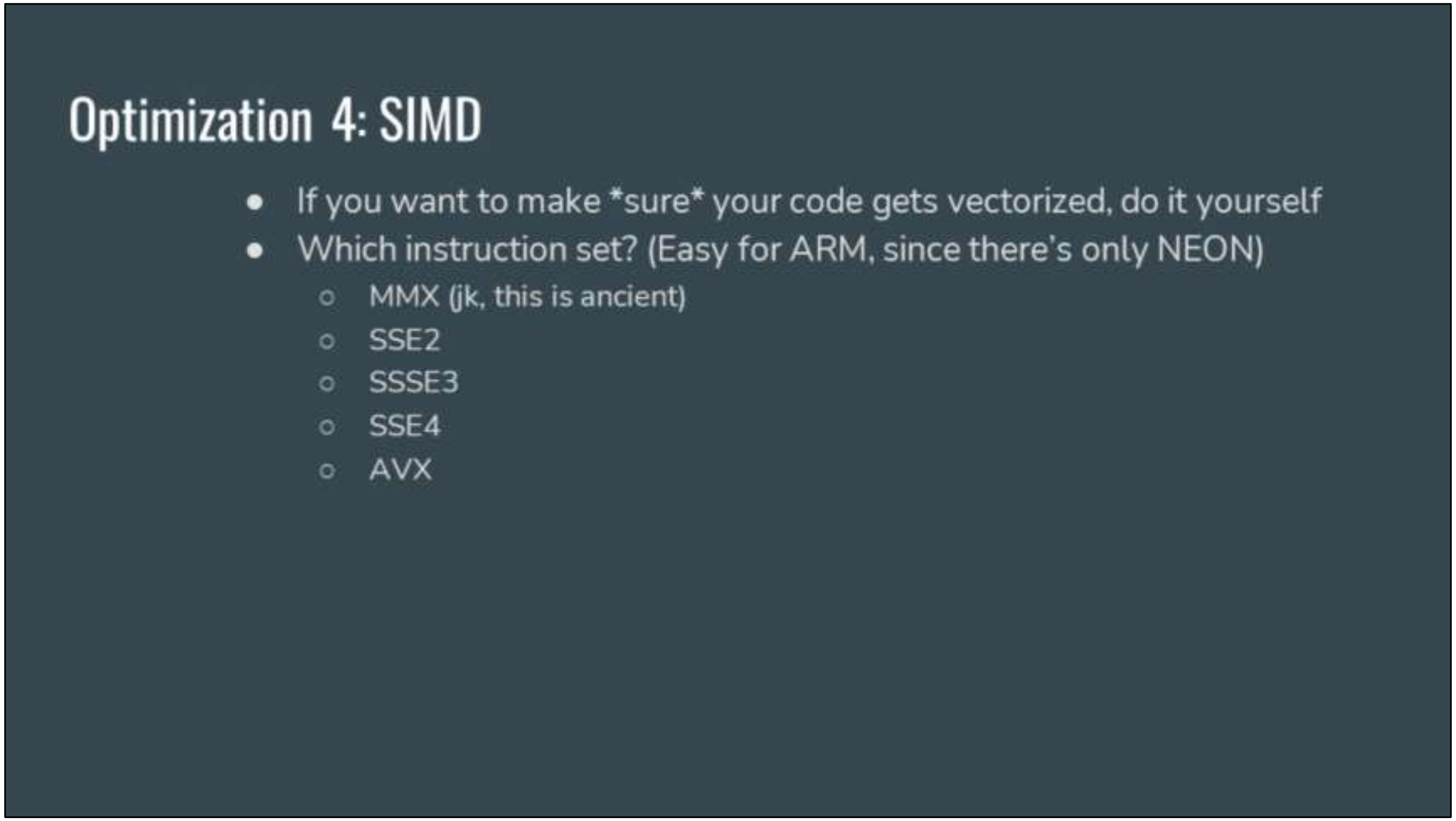
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2
  - SSSE3
  - SSE4

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

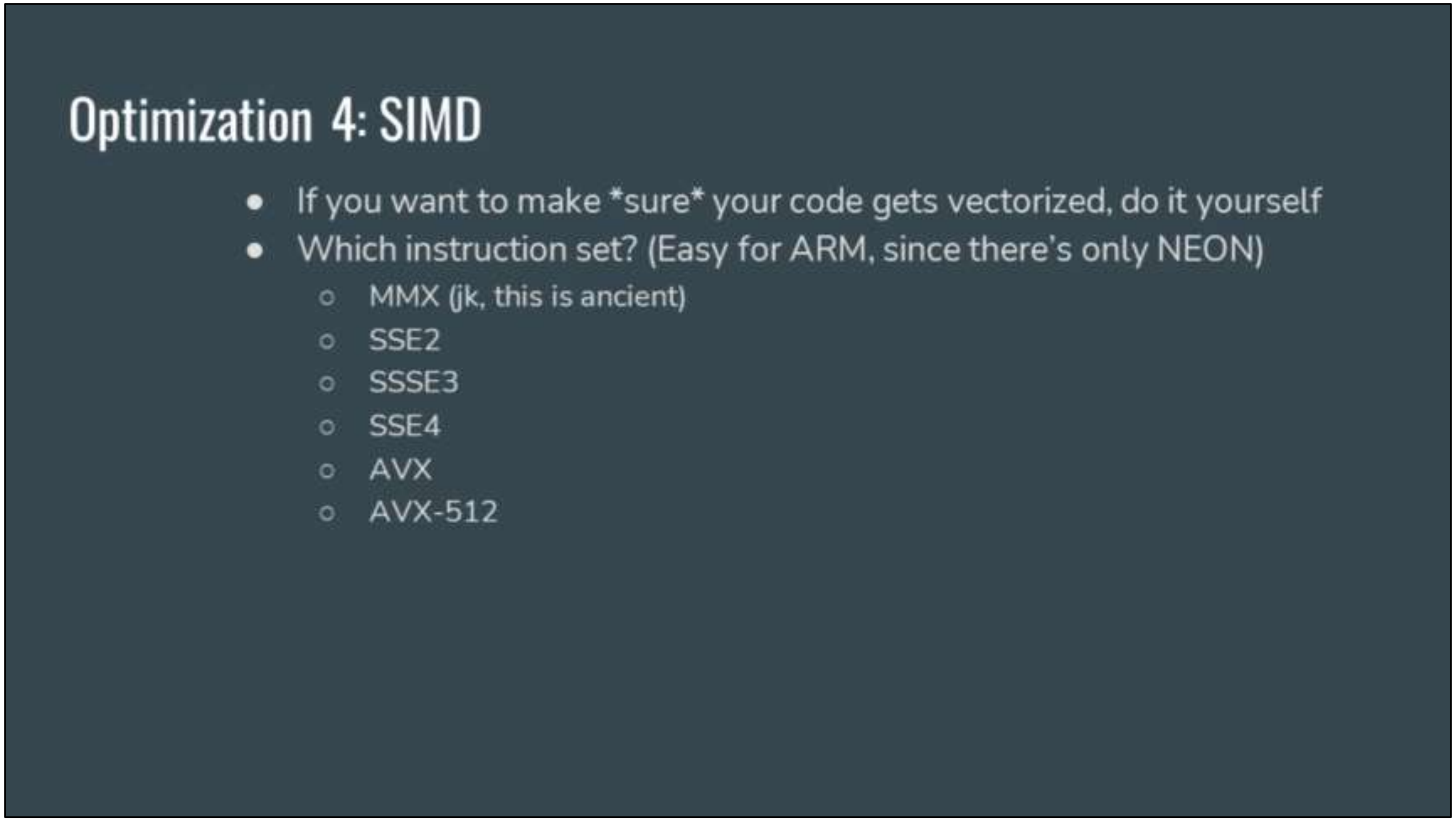
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2
  - SSSE3
  - SSE4
  - AVX

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

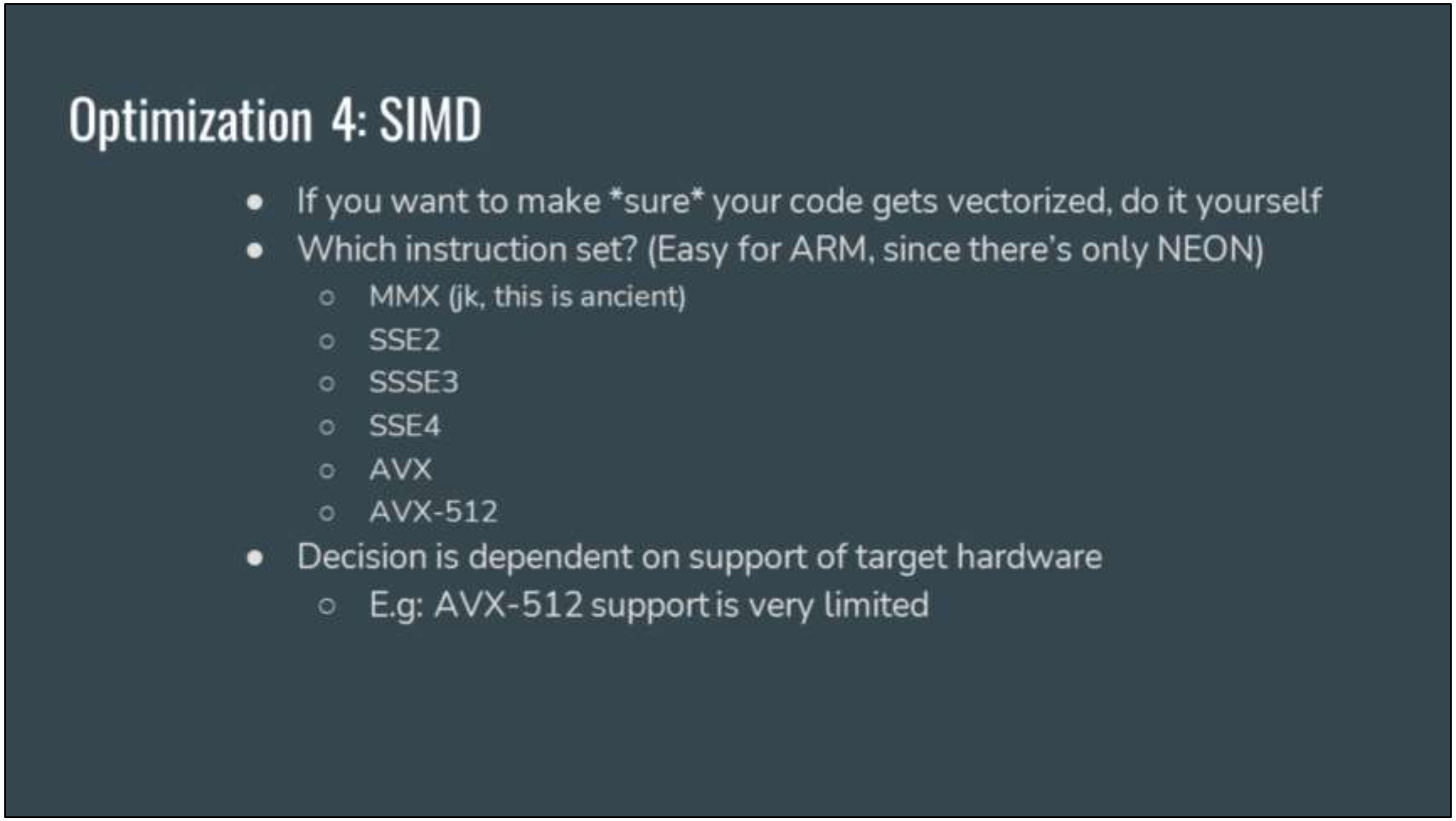
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2
  - SSSE3
  - SSE4
  - AVX
  - AVX-512

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

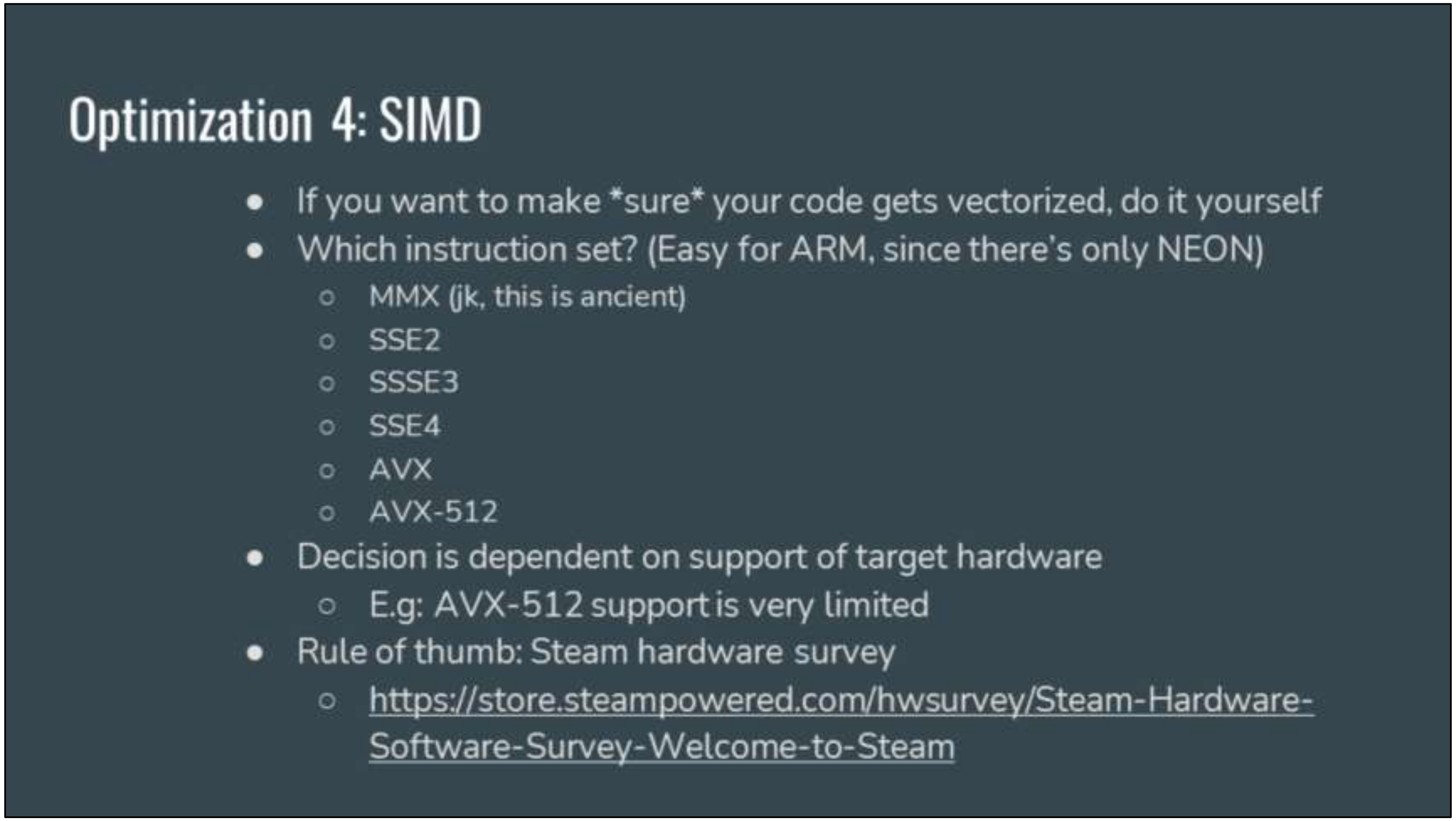
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2
  - SSSE3
  - SSE4
  - AVX
  - AVX-512
- Decision is dependent on support of target hardware
  - E.g: AVX-512 support is very limited

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

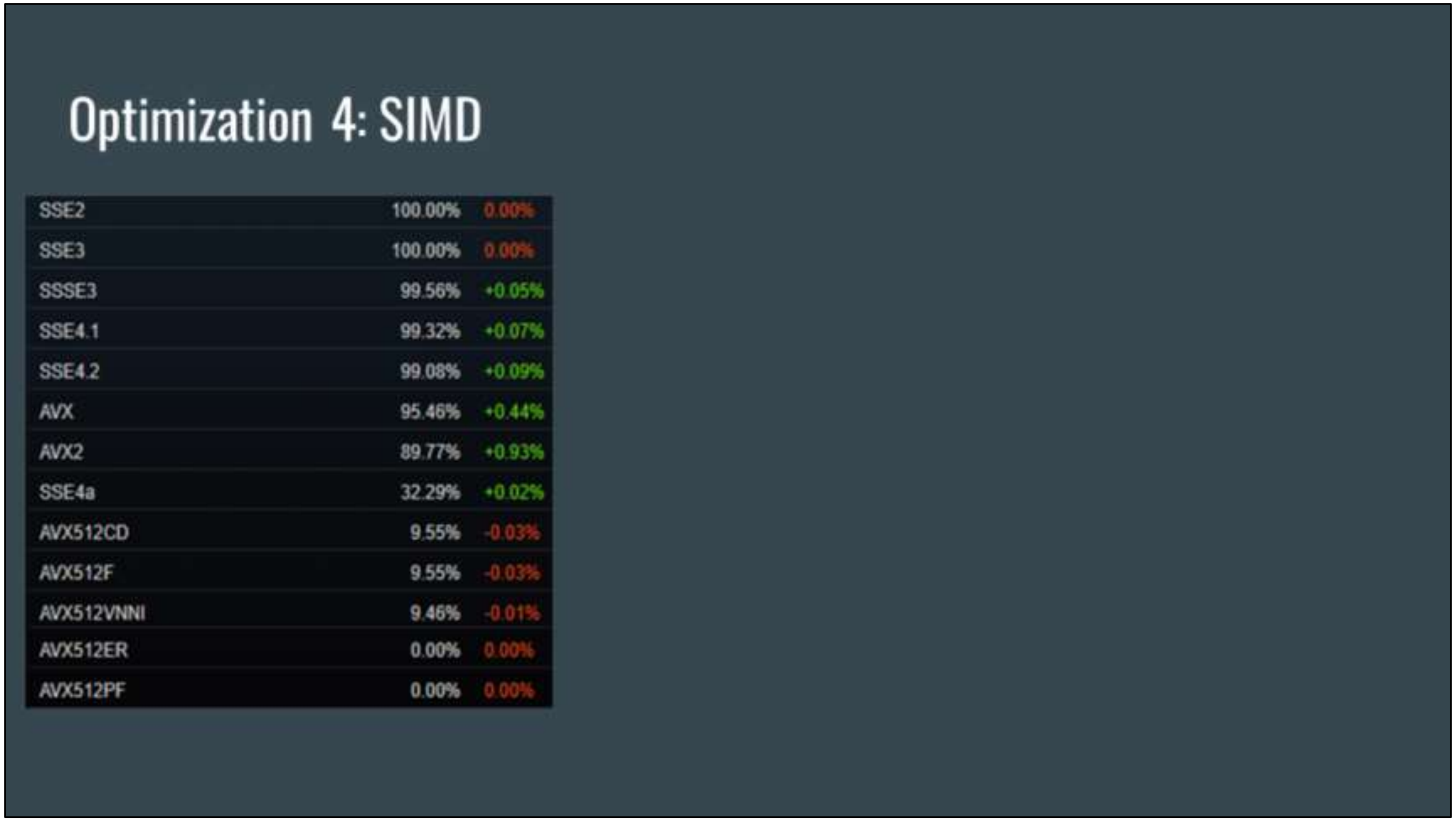
- If you want to make *sure* your code gets vectorized, do it yourself
- Which instruction set? (Easy for ARM, since there's only NEON)
  - MMX (jk, this is ancient)
  - SSE2
  - SSSE3
  - SSE4
  - AVX
  - AVX-512
- Decision is dependent on support of target hardware
  - E.g: AVX-512 support is very limited
- Rule of thumb: Steam hardware survey
  - https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam

But back to our optimization: If you want to be 100% certain that a certain piece of code gets vectorized across compilers then you have to do it yourself or play with the source code to help the compiler with auto-vectorization.

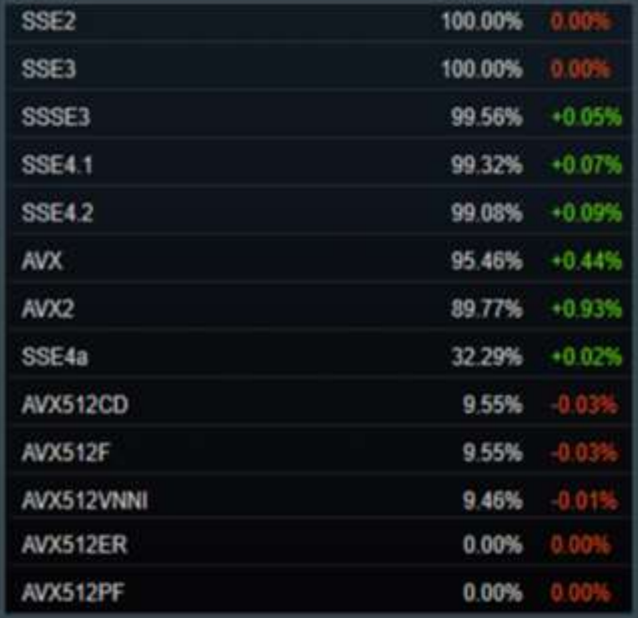
But if you want to roll your own SIMD implementation, you have to decide what instruction set(s) you want to target.
For ARM this choice is easy since there's only NEON but for x86_64 you have choice between multiple instruction sets.

The decision what instruction set to use depends on the hardware that you're targeting. Eg: if you work on software for a fixed platform like a game console then this becomes easy, just check what CPU the console uses and then you know what instruction sets will be available. For software targeting PCs this is a little bit more complicated.

To get a good feeling of what kind of hardware is out there, the Steam hardware survey might be good source of information.



Optimization 4: SIMD

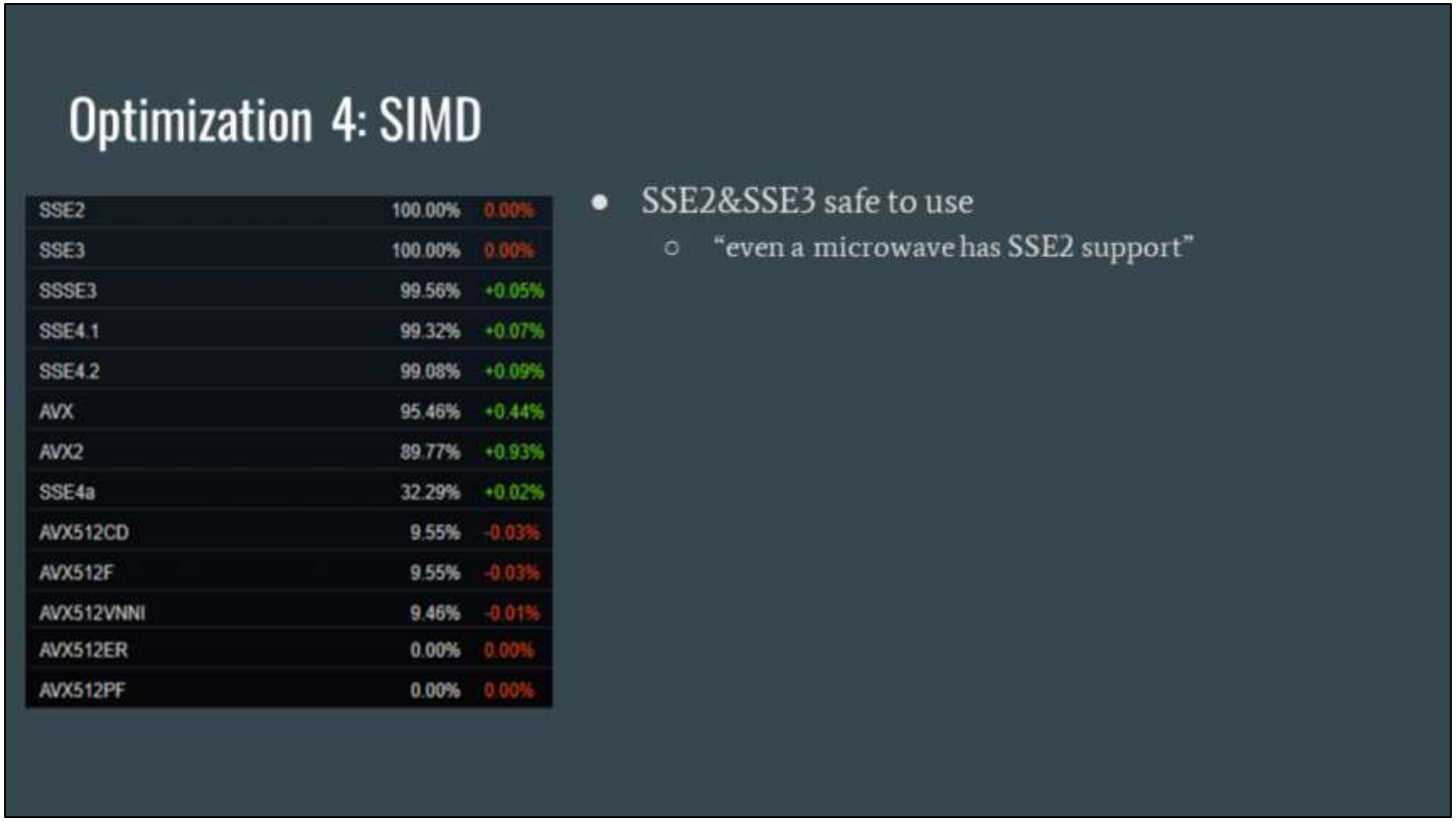


SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

Like, looking at this survey from a few days ago we can already see that's it's safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don't have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

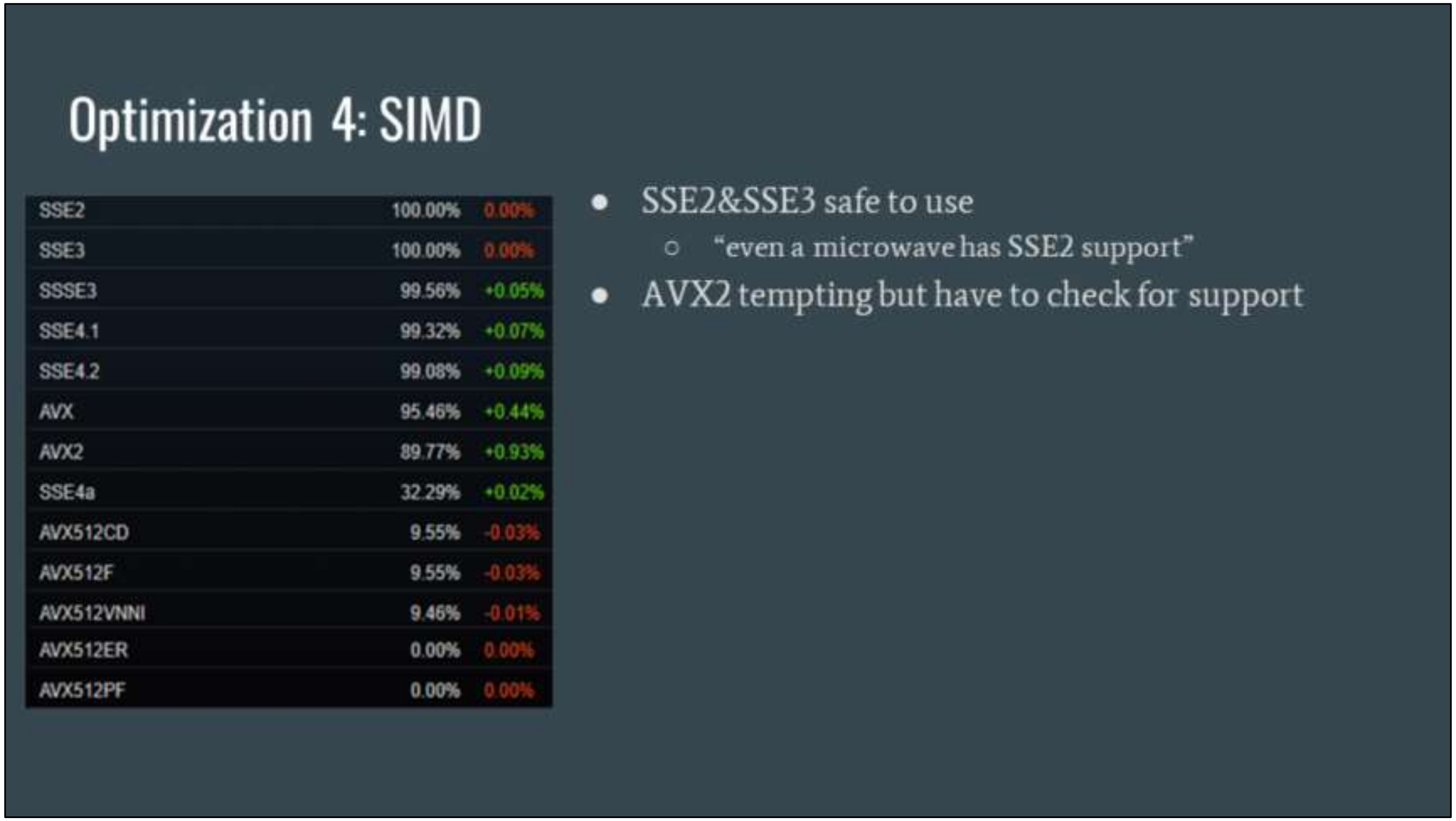
SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

- SSE2&SSE3 safe to use
  - "even a microwave has SSE2 support"

Like, looking at this survey from a few days ago we can already see that's it's safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don't have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

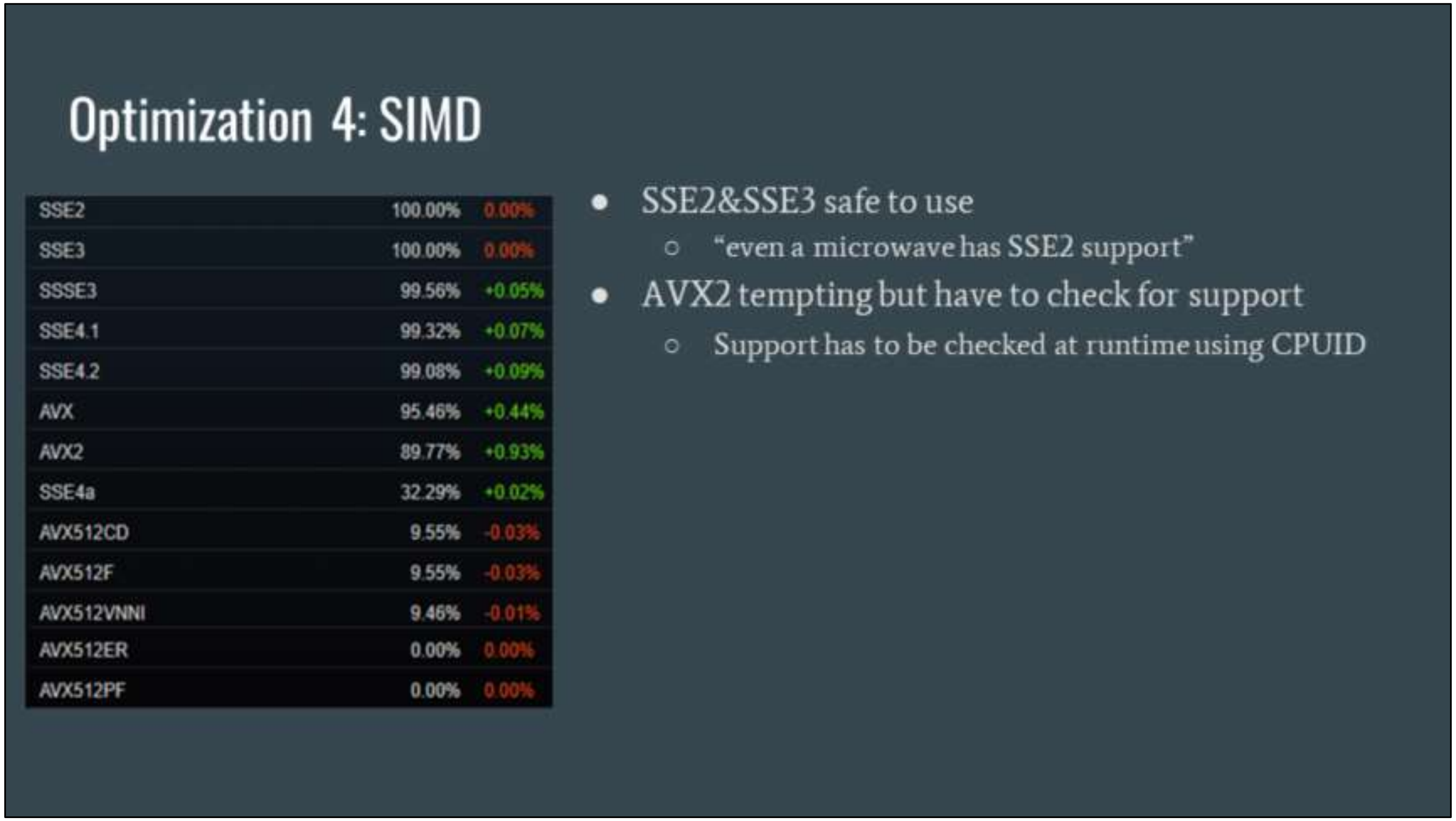
SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

- SSE2&SSE3 safe to use
  - “even a microwave has SSE2 support”
- AVX2 tempting but have to check for support

Like, looking at this survey from a few days ago we can already see that’s it’s safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don’t have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

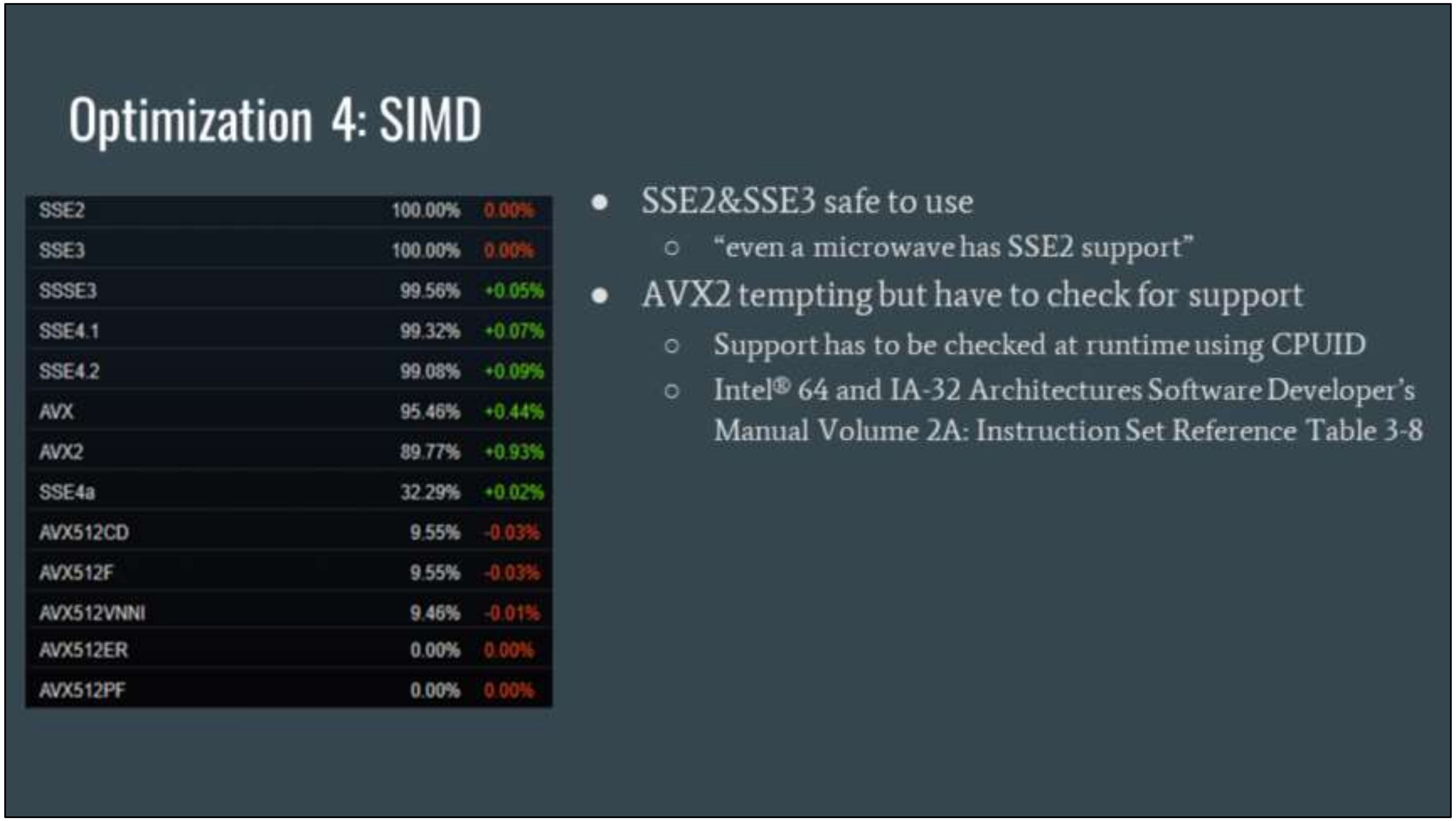
SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

- SSE2&SSE3 safe to use
  - "even a microwave has SSE2 support"
- AVX2 tempting but have to check for support
  - Support has to be checked at runtime using CPUID

Like, looking at this survey from a few days ago we can already see that's it's safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don't have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

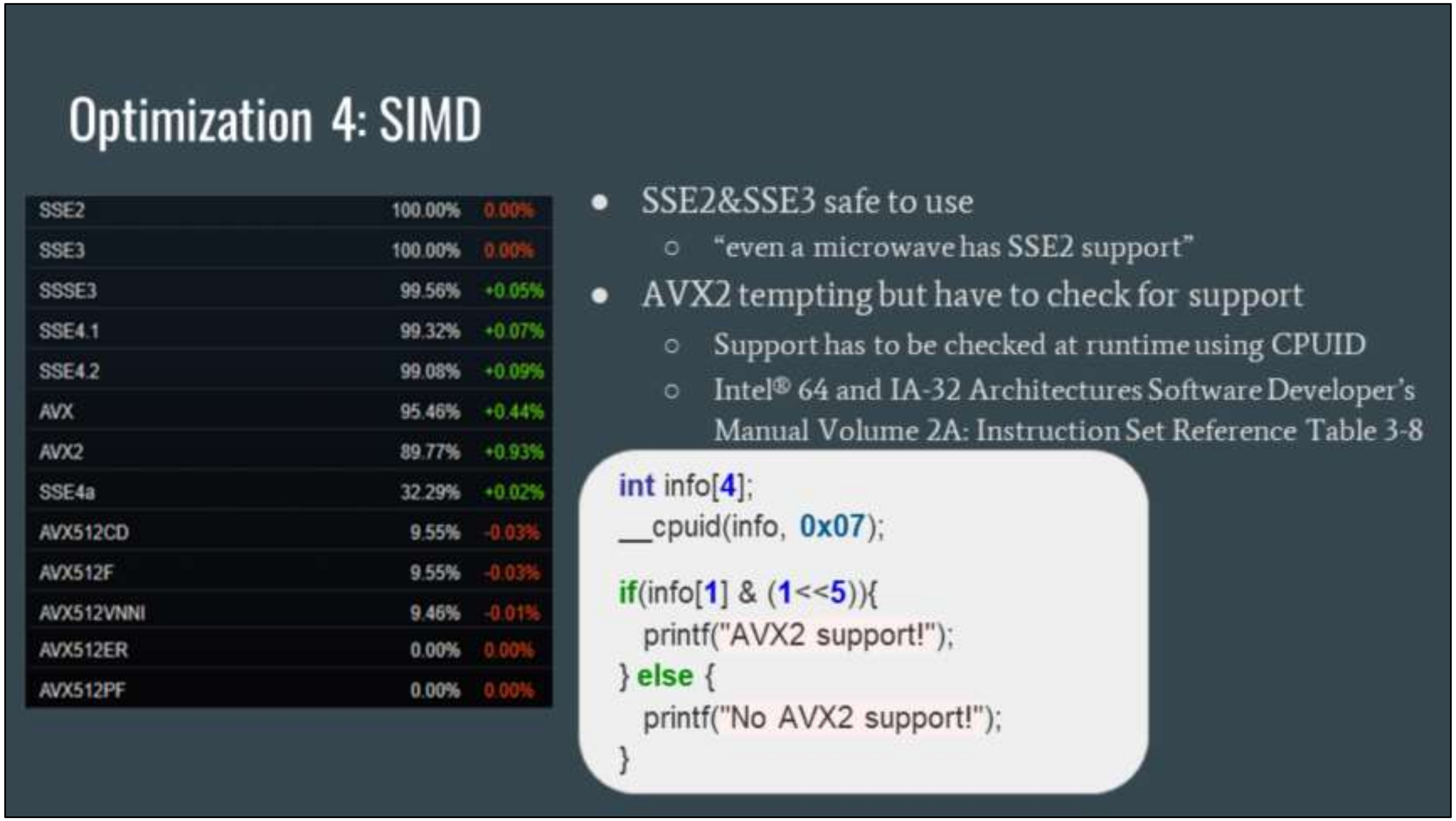
SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

- SSE2&SSE3 safe to use
  - “even a microwave has SSE2 support”
- AVX2 tempting but have to check for support
  - Support has to be checked at runtime using CPUID
  - Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference Table 3-8

Like, looking at this survey from a few days ago we can already see that’s it’s safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don’t have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

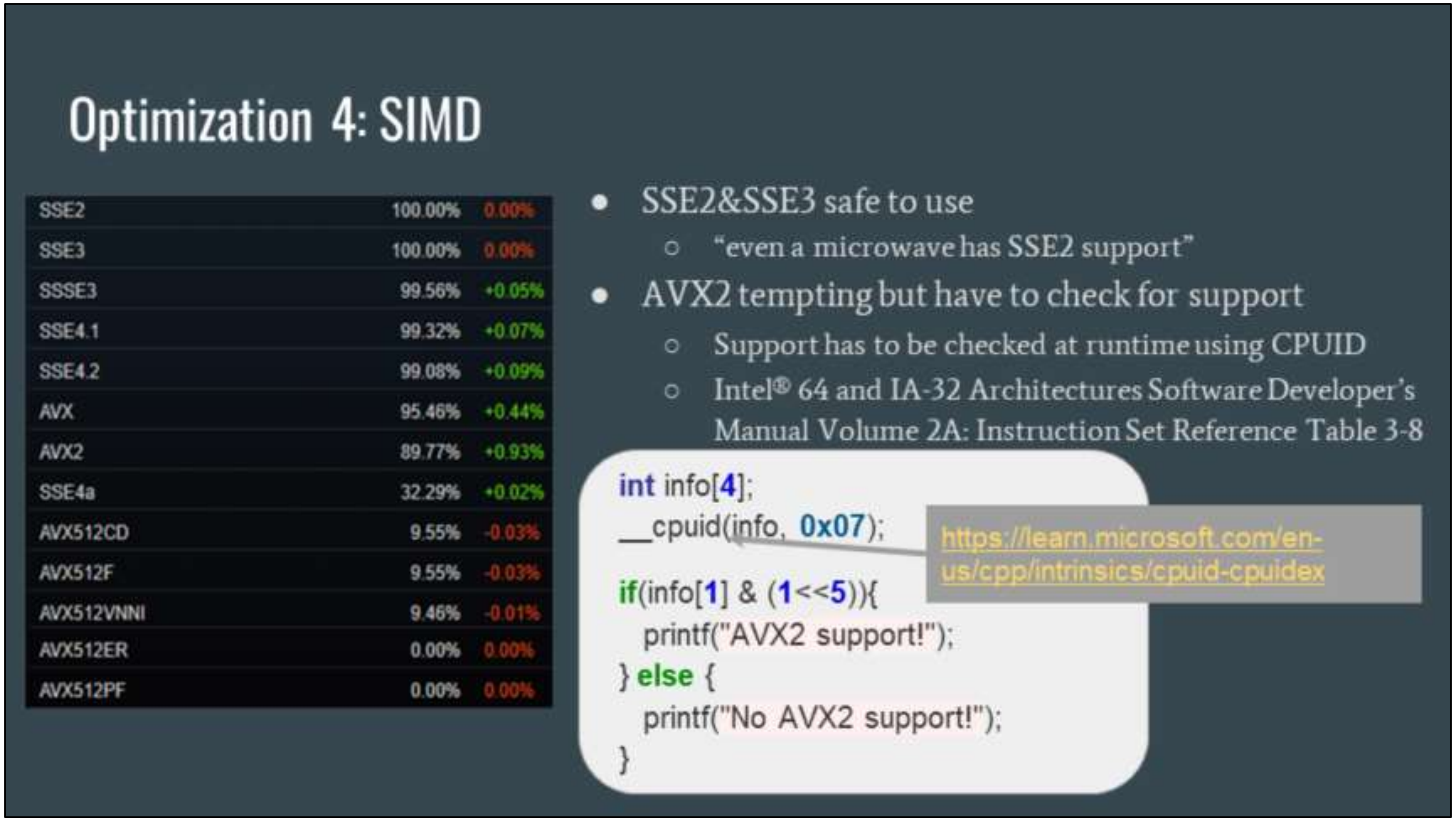
- SSE2&SSE3 safe to use
  - "even a microwave has SSE2 support"
- AVX2 tempting but have to check for support
  - Support has to be checked at runtime using CPUID
  - Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference Table 3-8

```
int info[4];
__cpuid(info, 0x07);

if(info[1] & (1<<5)){
  printf("AVX2 support!");
} else {
  printf("No AVX2 support!");
}
```

Like, looking at this survey from a few days ago we can already see that's it's safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don't have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.



Optimization 4: SIMD

SSE2	100.00%	0.00%
SSE3	100.00%	0.00%
SSSE3	99.56%	+0.05%
SSE4.1	99.32%	+0.07%
SSE4.2	99.08%	+0.09%
AVX	95.46%	+0.44%
AVX2	89.77%	+0.93%
SSE4a	32.29%	+0.02%
AVX512CD	9.55%	-0.03%
AVX512F	9.55%	-0.03%
AVX512VNNI	9.46%	-0.01%
AVX512ER	0.00%	0.00%
AVX512PF	0.00%	0.00%

- SSE2&SSE3 safe to use
  - "even a microwave has SSE2 support"
- AVX2 tempting but have to check for support
  - Support has to be checked at runtime using CPUID
  - Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference Table 3-8

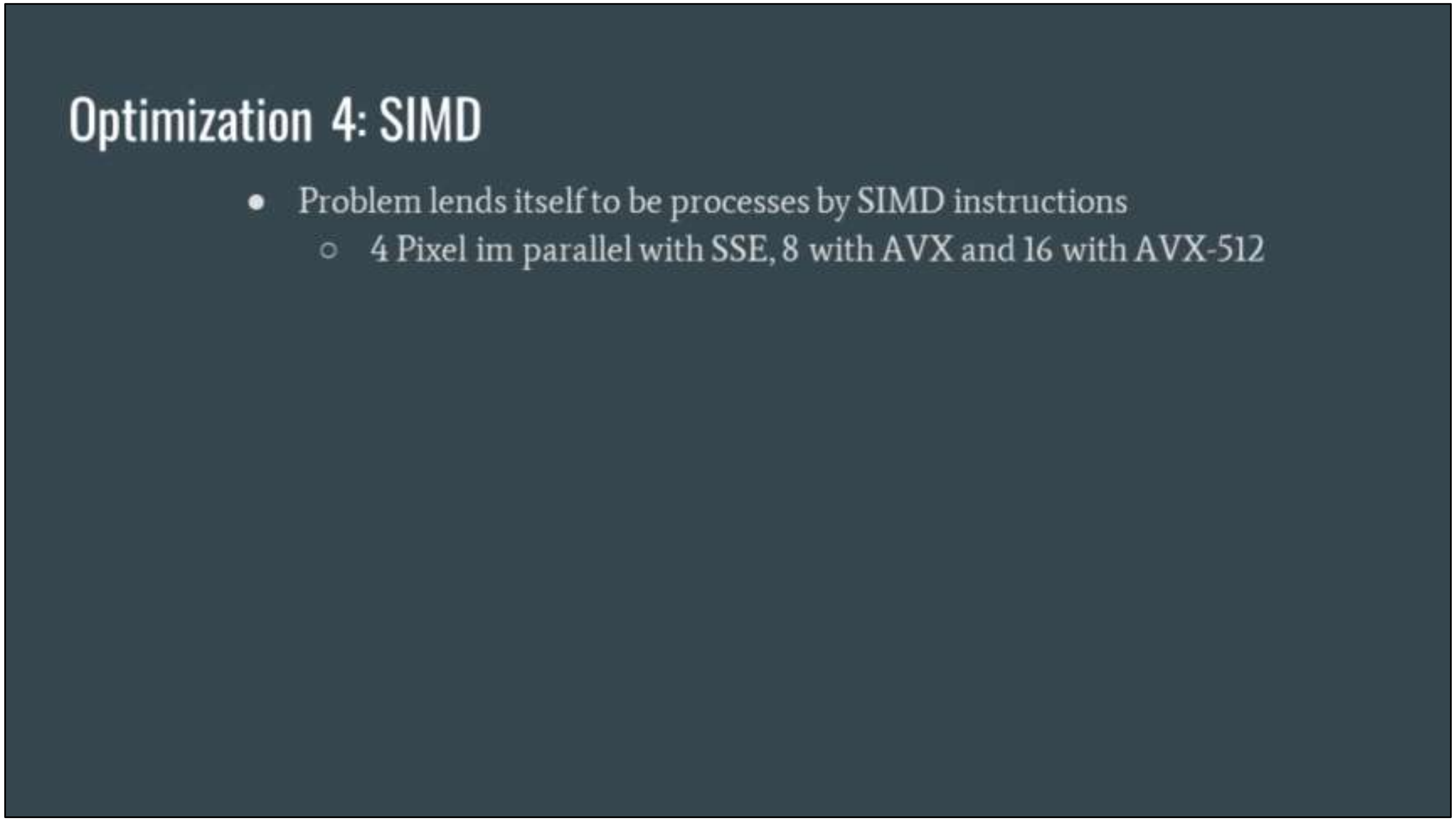
```
int info[4];
__cpuid(info, 0x07);

if(info[1] & (1<<5)){
  printf("AVX2 support!");
} else {
  printf("No AVX2 support!");
}
```

https://learn.microsoft.com/en-us/cpp/intrinsics/cpuid-cpuidex

Like, looking at this survey from a few days ago we can already see that's it's safe to use SSE2 and SSE3. I would argue that even SSE4.x is safe to use.
For our problem AVX2 seems tempting. We have wider registers and pretty wide support with nearly 90% of steam users having a CPU that supports AVX2.

To not instantly crash for users that don't have AVX2 support, code has to be added to check for support as run-time. For this the CPUID instruction can be used. This is an instruction that can be used to query information about the CPU. And we can use it to check for AVX2 support.

If the CPU does not support AVX2 then we can either inform the user and terminate the program or we can provide the scalar version so that the program still works.

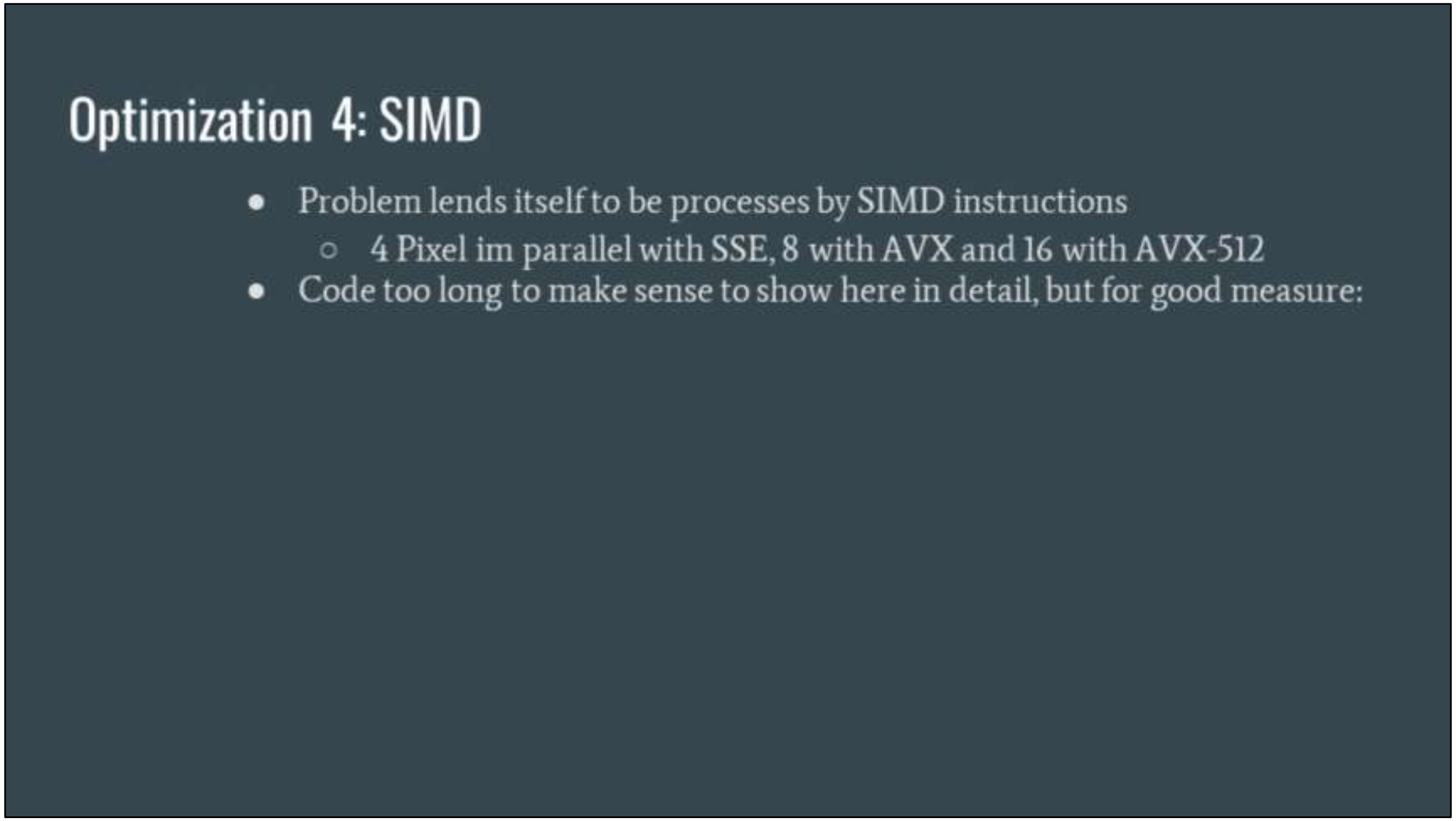


Optimization 4: SIMD

- Problem lends itself to be processes by SIMD instructions
  - 4 Pixel im parallel with SSE, 8 with AVX and 16 with AVX-512

Using SIMD for our problem is a good fit. Depending on what instruction set we use, we can either operate at 4, 8 or 16 pixel in parallel. Since we choose to use AVX2, we can work on 8 pixels at a time.

You already saw in earlier examples that moving an algorithm to make use of SIMD intrinsics will greatly increase the amount of code that is written - so it makes no sense here to show you all the SIMD code in detail but for good measure here are 2 screenshots of the code. This is the complete thing with transformation, bilinear sampling and mirror addressing mode.

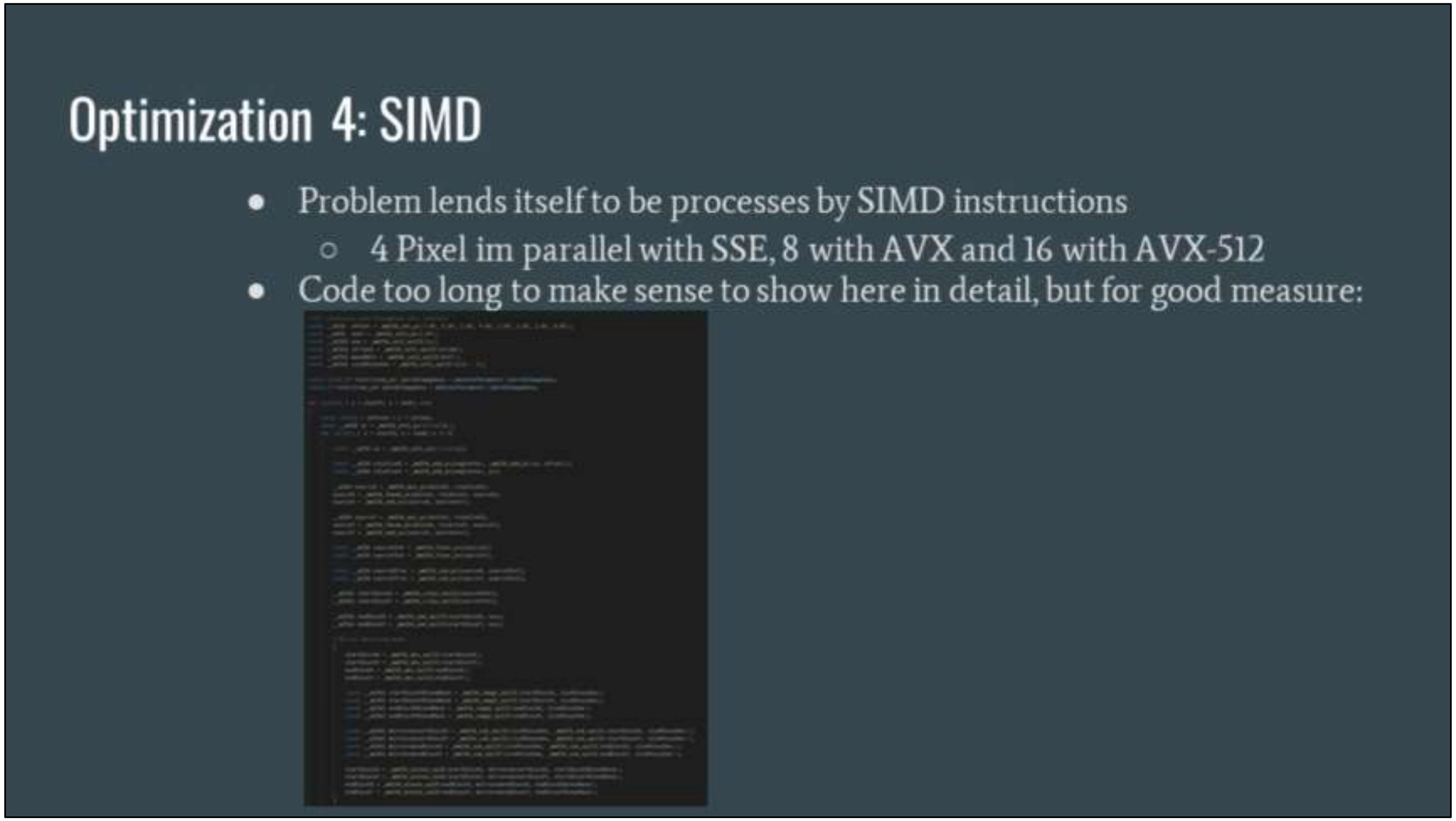


Optimization 4: SIMD

- Problem lends itself to be processes by SIMD instructions
  - 4 Pixel im parallel with SSE, 8 with AVX and 16 with AVX-512
- Code too long to make sense to show here in detail, but for good measure:

Using SIMD for our problem is a good fit. Depending on what instruction set we use, we can either operate at 4, 8 or 16 pixel in parallel. Since we choose to use AVX2, we can work on 8 pixels at a time.

You already saw in earlier examples that moving an algorithm to make use of SIMD intrinsics will greatly increase the amount of code that is written - so it makes no sense here to show you all the SIMD code in detail but for good measure here are 2 screenshots of the code. This is the complete thing with transformation, bilinear sampling and mirror addressing mode.



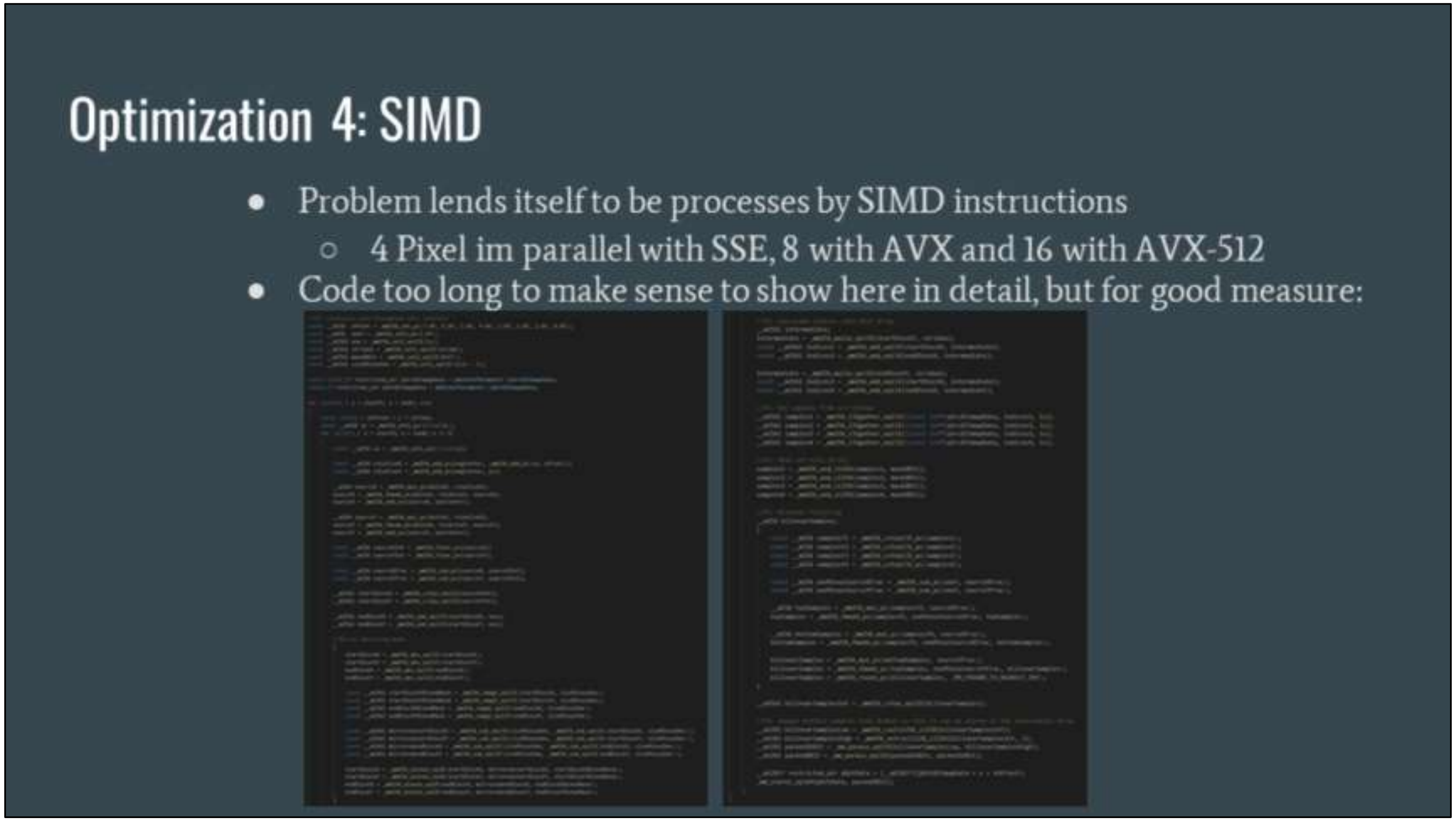
Optimization 4: SIMD

- Problem lends itself to be processes by SIMD instructions
  - 4 Pixel im parallel with SSE, 8 with AVX and 16 with AVX-512
- Code too long to make sense to show here in detail, but for good measure:

```
[illegible]
```

Using SIMD for our problem is a good fit. Depending on what instruction set we use, we can either operate at 4, 8 or 16 pixel in parallel. Since we choose to use AVX2, we can work on 8 pixels at a time.

You already saw in earlier examples that moving an algorithm to make use of SIMD intrinsics will greatly increase the amount of code that is written - so it makes no sense here to show you all the SIMD code in detail but for good measure here are 2 screenshots of the code. This is the complete thing with transformation, bilinear sampling and mirror addressing mode.

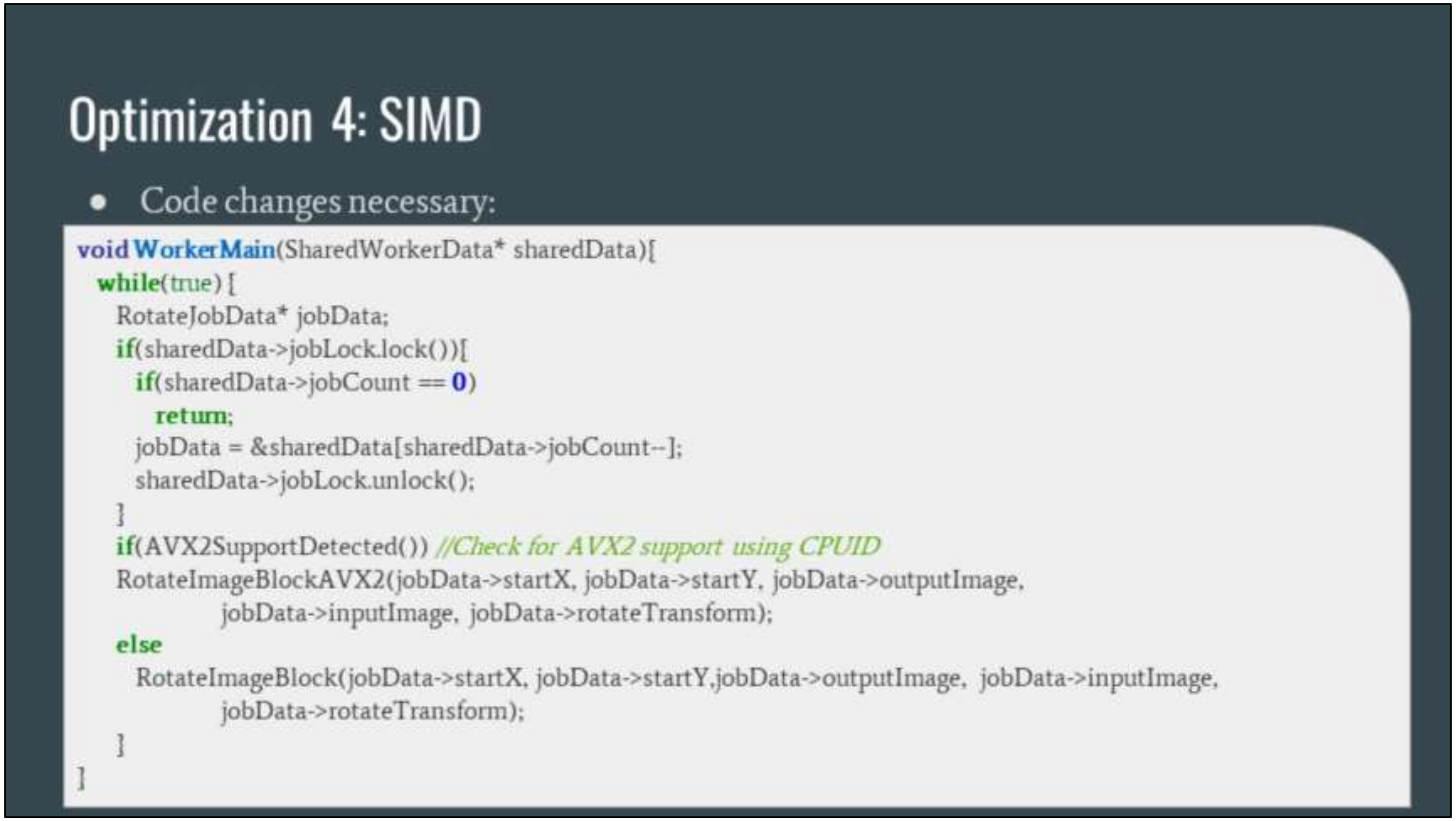


Optimization 4: SIMD

- Problem lends itself to be processes by SIMD instructions
  - 4 Pixel im parallel with SSE, 8 with AVX and 16 with AVX-512
- Code too long to make sense to show here in detail, but for good measure:

Using SIMD for our problem is a good fit. Depending on what instruction set we use, we can either operate at 4, 8 or 16 pixel in parallel. Since we choose to use AVX2, we can work on 8 pixels at a time.

You already saw in earlier examples that moving an algorithm to make use of SIMD intrinsics will greatly increase the amount of code that is written - so it makes no sense here to show you all the SIMD code in detail but for good measure here are 2 screenshots of the code. This is the complete thing with transformation, bilinear sampling and mirror addressing mode.

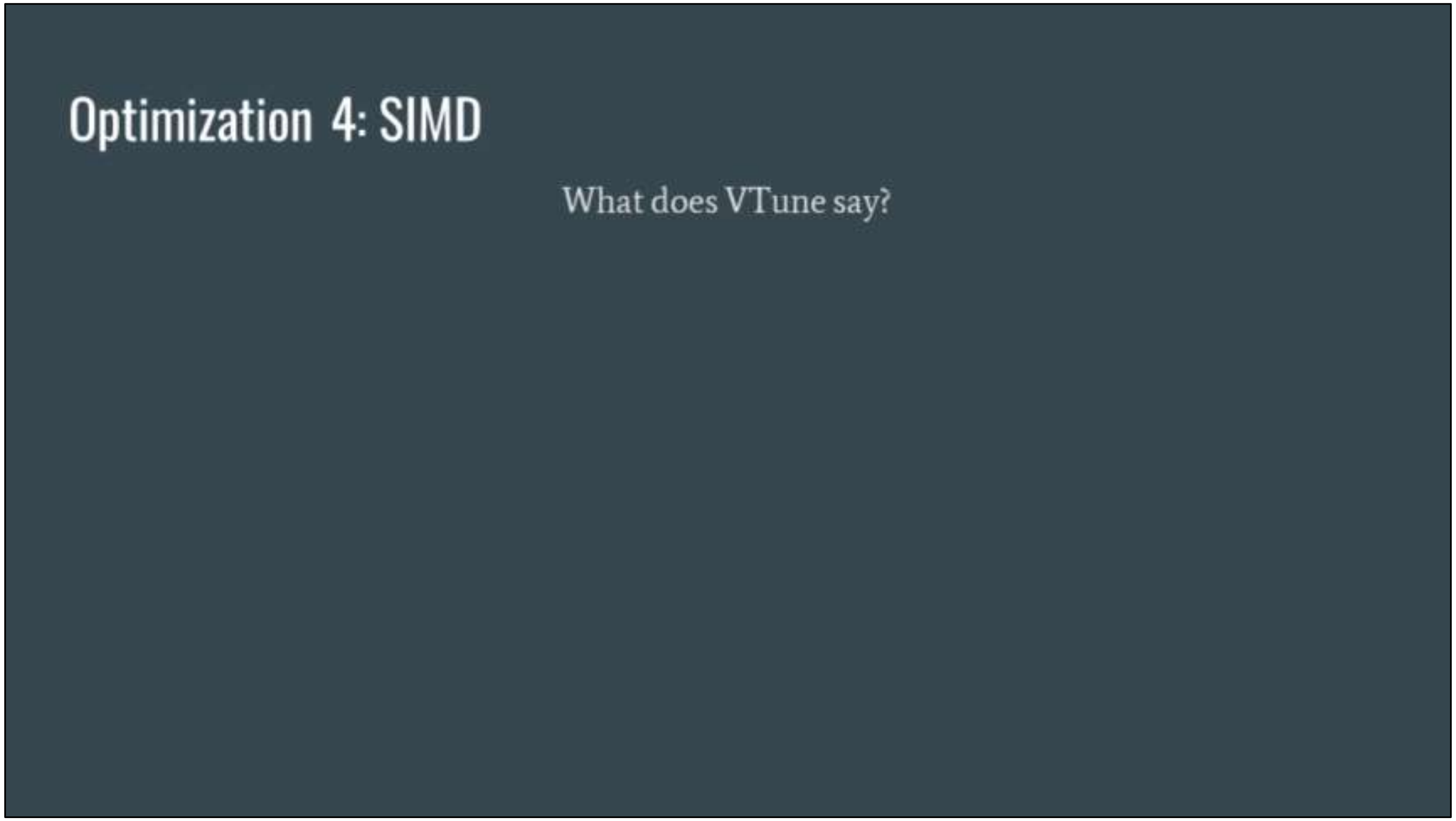


Optimization 4: SIMD

- Code changes necessary:

```
void WorkerMain(SharedWorkerData* sharedData){
  while(true) {
    RotateJobData* jobData;
    if(sharedData->jobLock.lock()){
      if(sharedData->jobCount == 0)
        return;
      jobData = &sharedData[sharedData->jobCount--];
      sharedData->jobLock.unlock();
    }
    if(AVX2SupportDetected()) //Check for AVX2 support using CPUID
    RotateImageBlockAVX2(jobData->startX, jobData->startY, jobData->outputImage,
             jobData->inputImage, jobData->rotateTransform);
    else
      RotateImageBlock(jobData->startX, jobData->startY,jobData->outputImage,  jobData->inputImage,
             jobData->rotateTransform);
    }
}
```

To use the new code in our existing codebase, we have to add this code to the WorkerMain function.
We check for AVX2 support and run the AVX2 code if support has been detected and if not, we fall back to the scalar version.


Optimization 4: SIMD

What does VTune say?

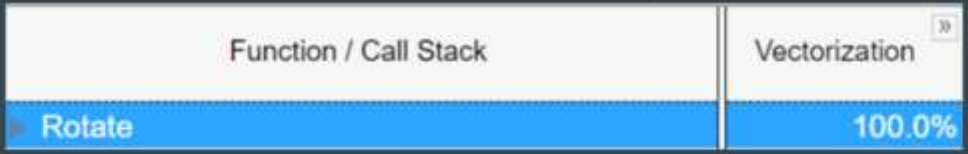
Let’s check VTune again to see what it has to say and tada, it now correctly detects 100% vectorization in our function.

If even shows us what instruction sets have been used and all this work results in a nice performance improvement.

Looking at the performance, we see another nice boost.

Optimization 4: SIMD

What does VTune say?



Function / Call Stack	Vectorization
Rotate	100.0%

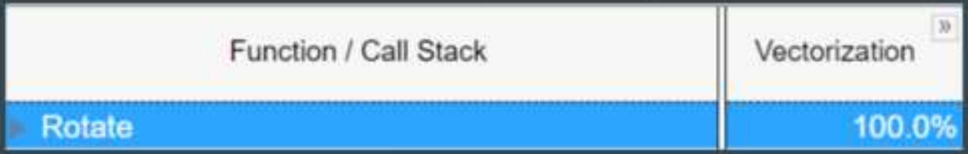
Let’s check VTune again to see what it has to say and tada, it now correctly detects 100% vectorization in our function.

If even shows us what instruction sets have been used and all this work results in a nice performance improvement.

Looking at the performance, we see another nice boost.

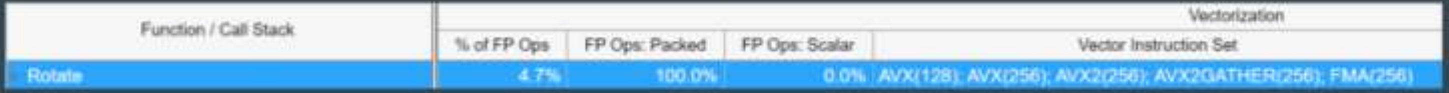
Optimization 4: SIMD

What does VTune say?



Function / Call Stack	Vectorization
Rotate	100.0%

Even tells us what instruction sets have been used:

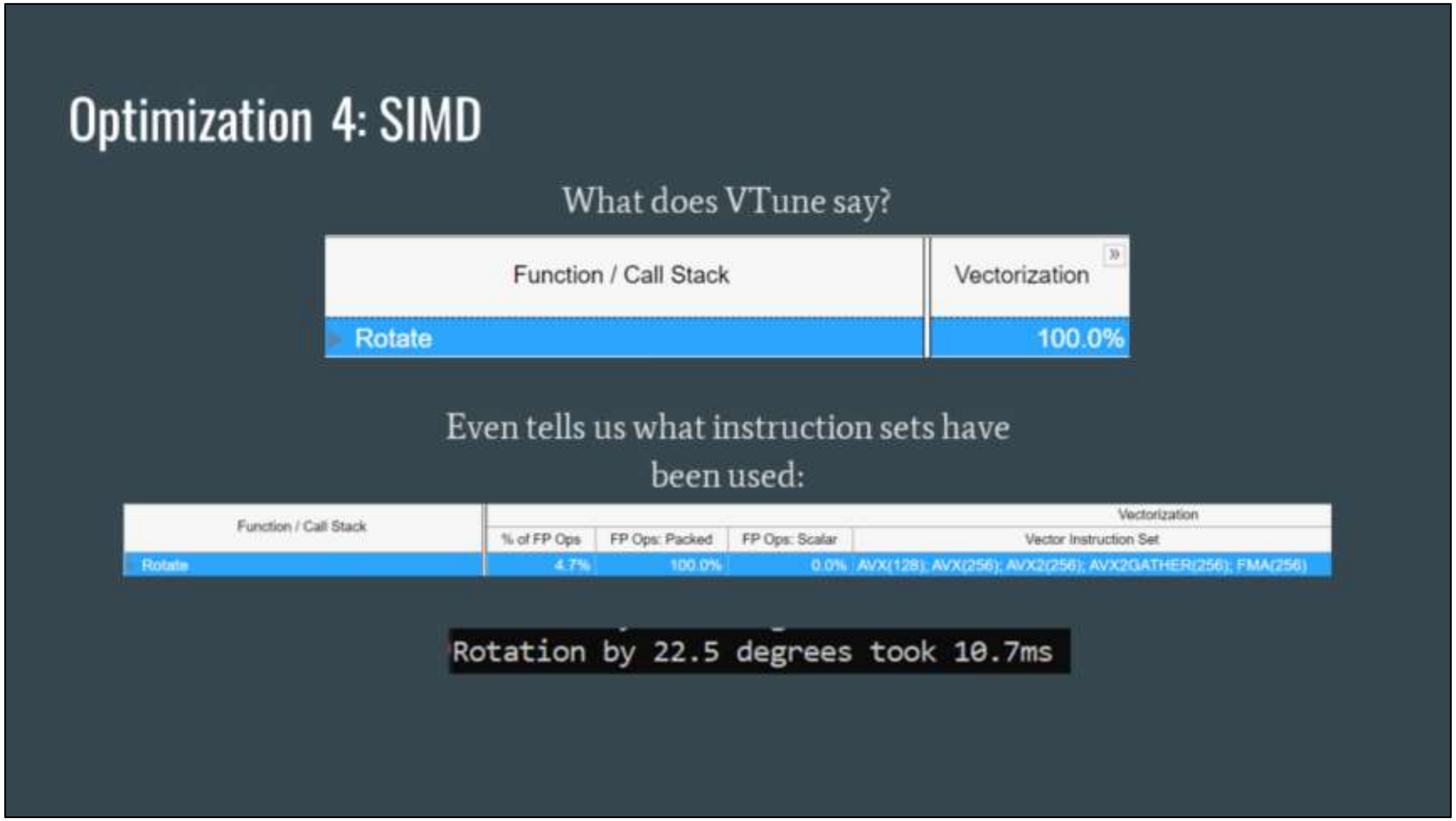


Function / Call Stack	Vectorization			
	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
Rotate	4.7%	100.0%	0.0%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(256); FMA(256)

Let’s check VTune again to see what it has to say and tada, it now correctly detects 100% vectorization in our function.

If even shows us what instruction sets have been used and all this work results in a nice performance improvement.

Looking at the performance, we see another nice boost.



Optimization 4: SIMD

What does VTune say?

Function / Call Stack	Vectorization
Rotate	100.0%

Even tells us what instruction sets have been used:

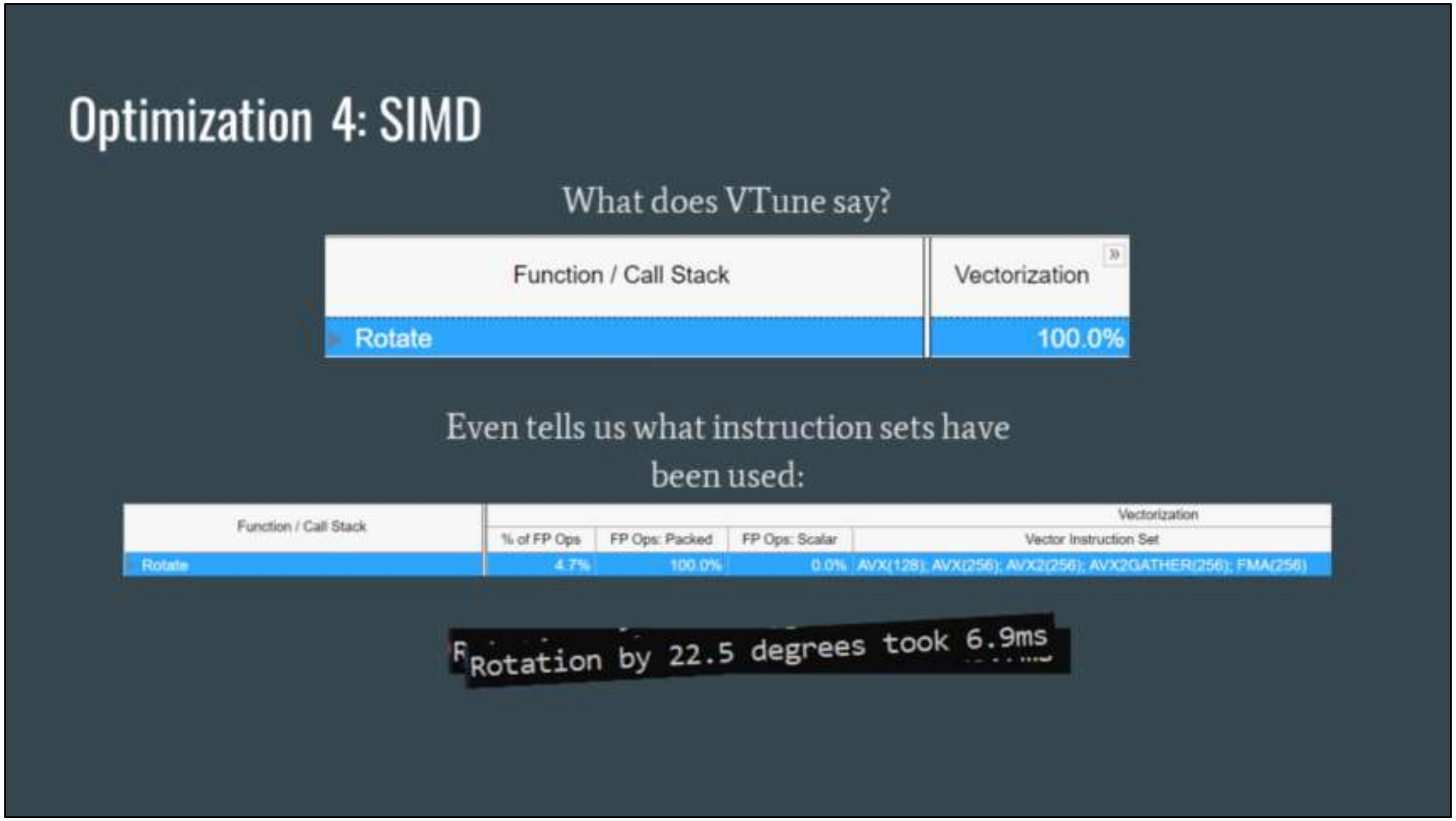
Function / Call Stack	Vectorization			
	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
Rotate	4.7%	100.0%	0.0%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(256); FMA(256)

```
Rotation by 22.5 degrees took 10.7ms
```

Let’s check VTune again to see what it has to say and tada, it now correctly detects 100% vectorization in our function.

If even shows us what instruction sets have been used and all this work results in a nice performance improvement.

Looking at the performance, we see another nice boost.



Optimization 4: SIMD

What does VTune say?

Function / Call Stack	Vectorization
Rotate	100.0%

Even tells us what instruction sets have been used:

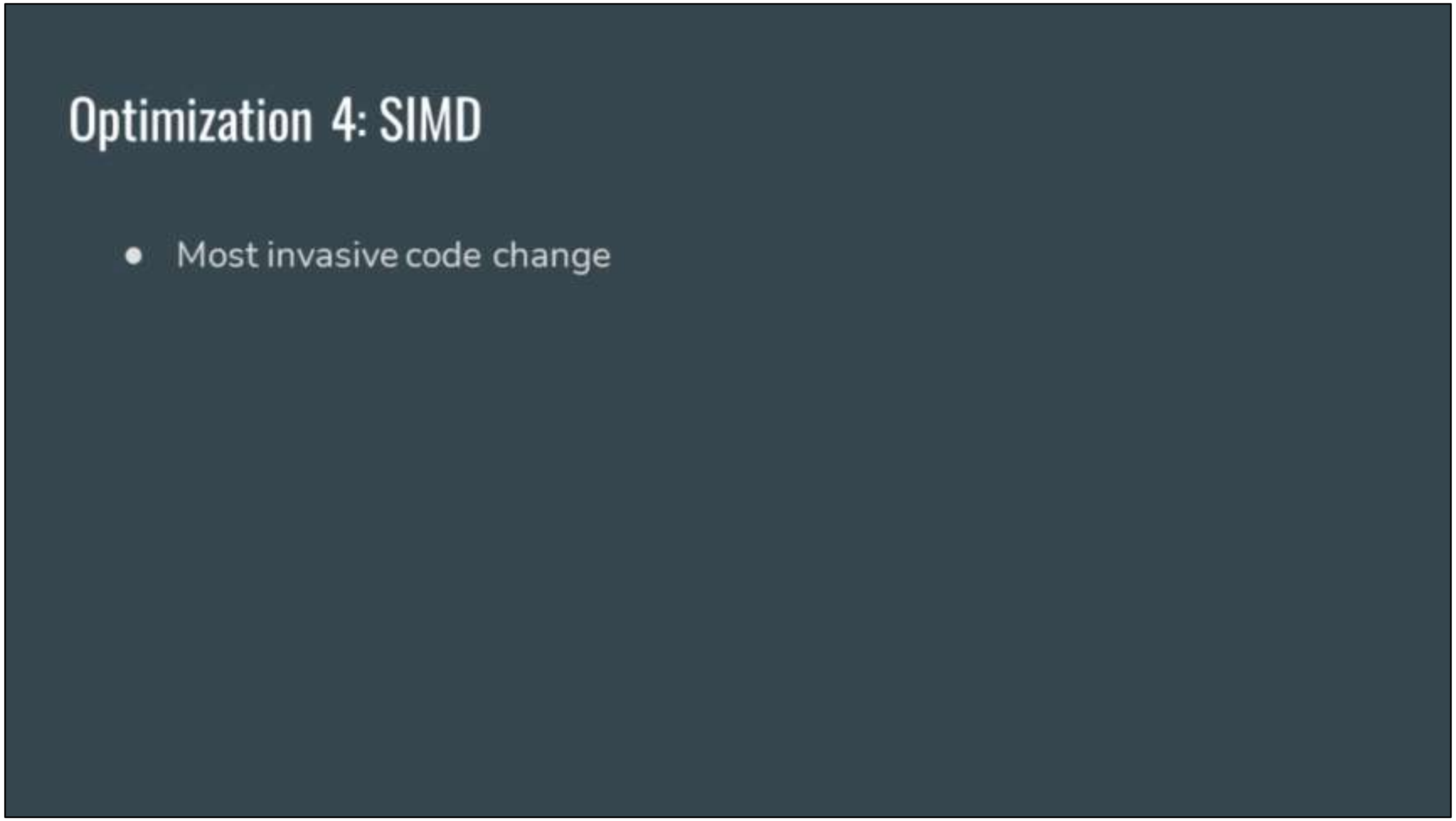
Function / Call Stack	Vectorization			
	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
Rotate	4.7%	100.0%	0.0%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(256); FMA(256)

```
Rotation by 22.5 degrees took 6.9ms
```

Let’s check VTune again to see what it has to say and tada, it now correctly detects 100% vectorization in our function.

If even shows us what instruction sets have been used and all this work results in a nice performance improvement.

Looking at the performance, we see another nice boost.



Optimization 4: SIMD

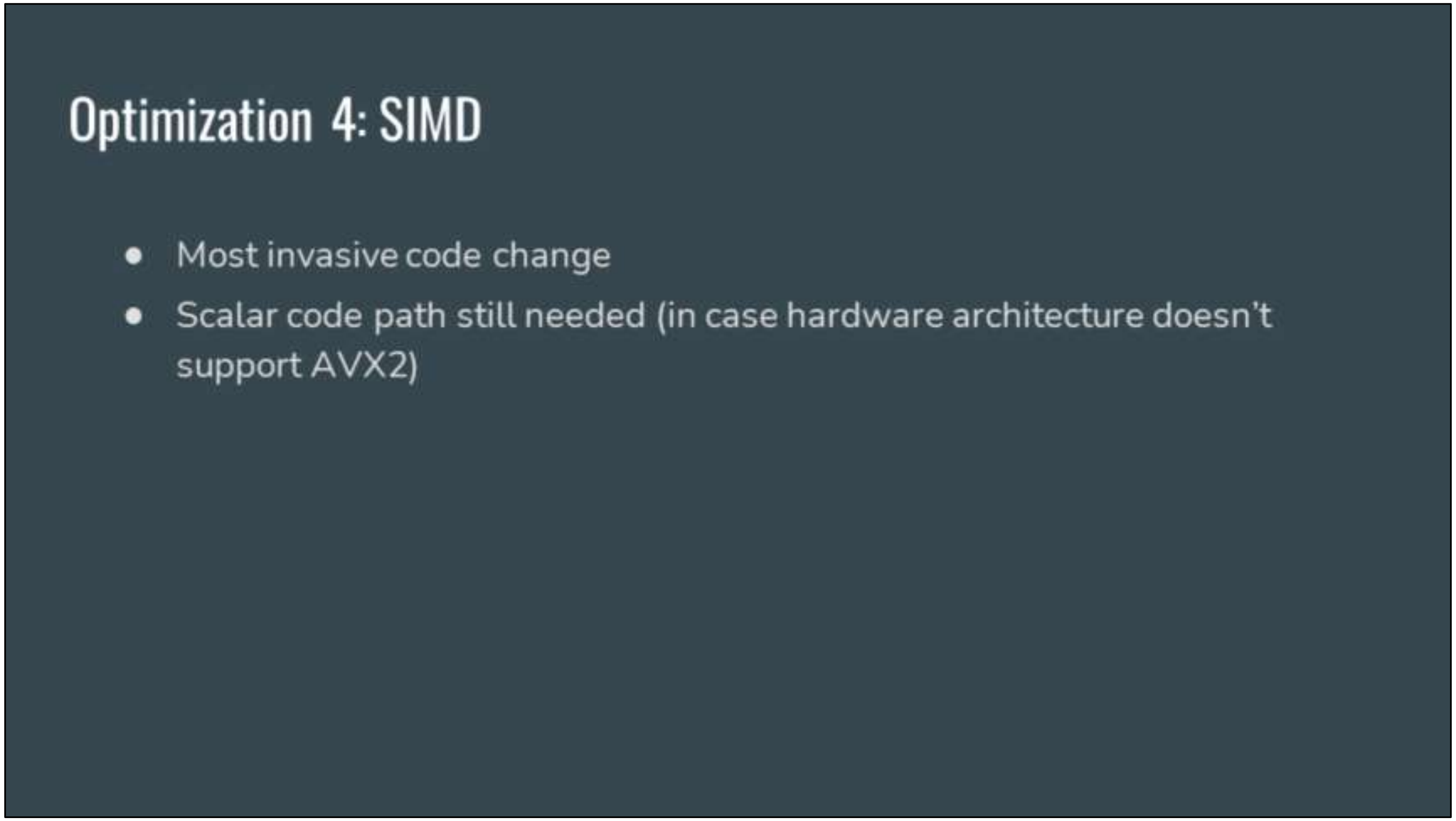
- Most invasive code change

While this is nice and we’re now in the realm of real-time processing this is by far the most invasive code change since we had to completely rewrite the rotation function to make use of the AVX2 instruction set. Additionally we can’t even delete the old code path or else users without AVX2 won’t be able to run our program (we might AVX2 a hard requirement though).

On top of that I’d argue that the set of people how can read and debug this code has been reduced, but I might be a little bit pessimistic in this regard.

If you feel more comfortable writing “scalar-looking” C code then you might want to check out ISCP. ISCP is a compiler from Intel that targets a C language with additional extensions for better auto-vectorization.

Additionally there’s the Intel Intrinsics Guide, which will show you all the SIMD intrinsics that come with the different instruction sets.



Optimization 4: SIMD

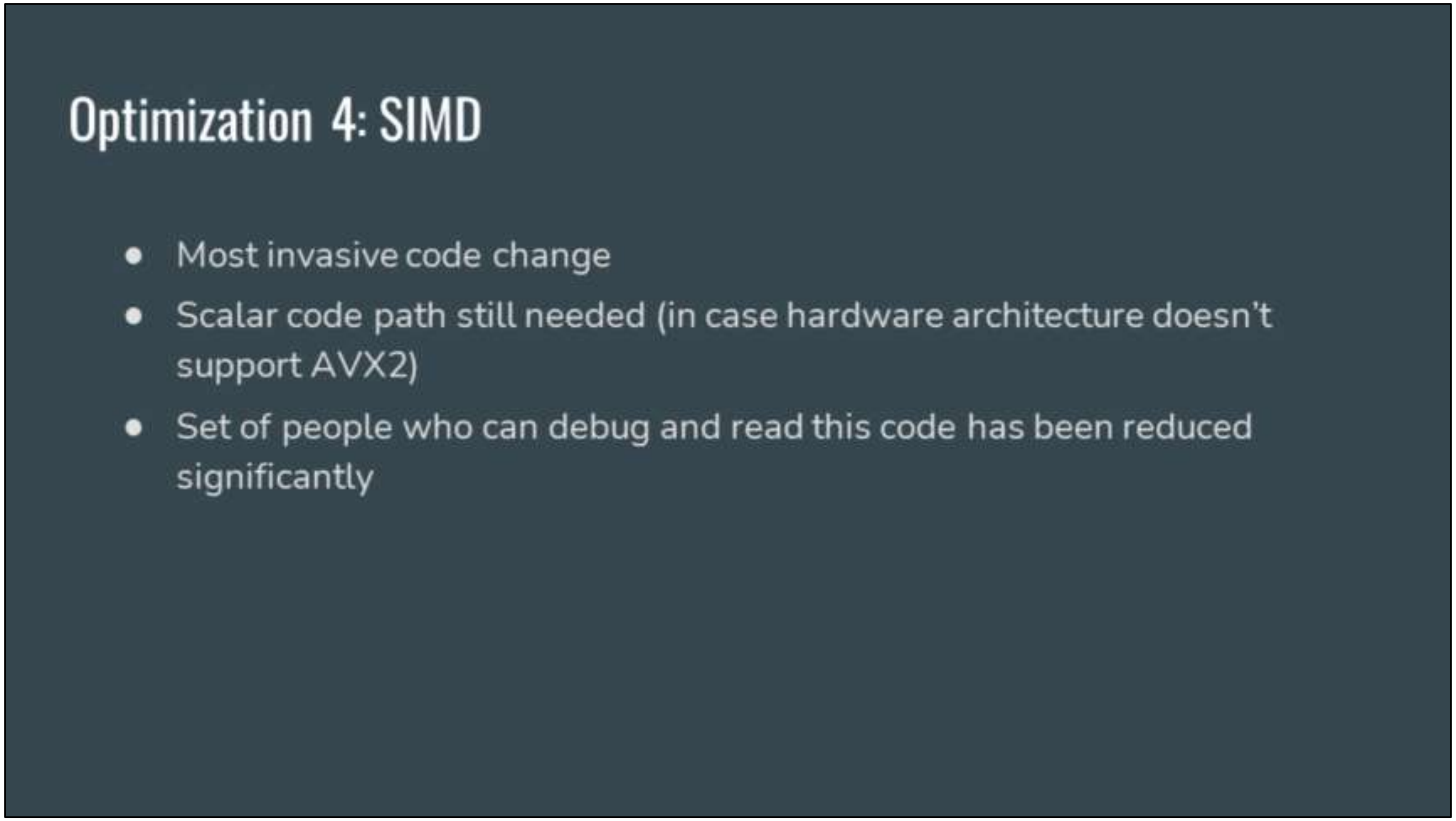
- Most invasive code change
- Scalar code path still needed (in case hardware architecture doesn't support AVX2)

While this is nice and we're now in the realm of real-time processing this is by far the most invasive code change since we had to completely rewrite the rotation function to make use of the AVX2 instruction set. Additionally we can't even delete the old code path or else users without AVX2 won't be able to run our program (we might AVX2 a hard requirement though).

On top of that I'd argue that the set of people how can read and debug this code has been reduced, but I might be a little bit pessimistic in this regard.

If you feel more comfortable writing "scalar-looking" C code then you might want to check out ISCP. ISCP is a compiler from Intel that targets a C language with additional extensions for better auto-vectorization.

Additionally there's the Intel Intrinsics Guide, which will show you all the SIMD intrinsics that come with the different instruction sets.



Optimization 4: SIMD

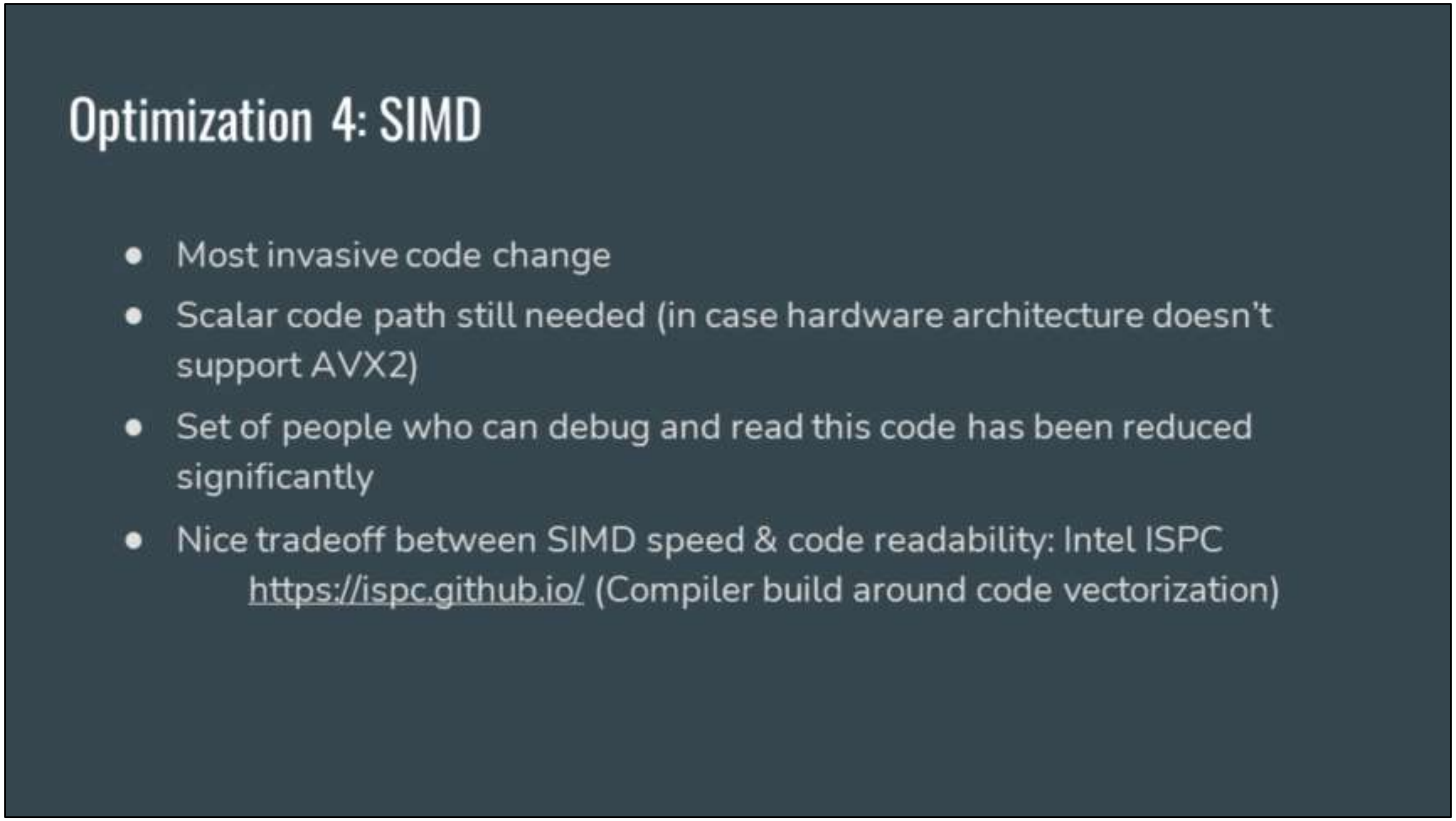
- Most invasive code change
- Scalar code path still needed (in case hardware architecture doesn't support AVX2)
- Set of people who can debug and read this code has been reduced significantly

While this is nice and we're now in the realm of real-time processing this is by far the most invasive code change since we had to completely rewrite the rotation function to make use of the AVX2 instruction set. Additionally we can't even delete the old code path or else users without AVX2 won't be able to run our program (we might AVX2 a hard requirement though).

On top of that I'd argue that the set of people how can read and debug this code has been reduced, but I might be a little bit pessimistic in this regard.

If you feel more comfortable writing "scalar-looking" C code then you might want to check out ISCP. ISCP is a compiler from Intel that targets a C language with additional extensions for better auto-vectorization.

Additionally there's the Intel Intrinsics Guide, which will show you all the SIMD intrinsics that come with the different instruction sets.



Optimization 4: SIMD

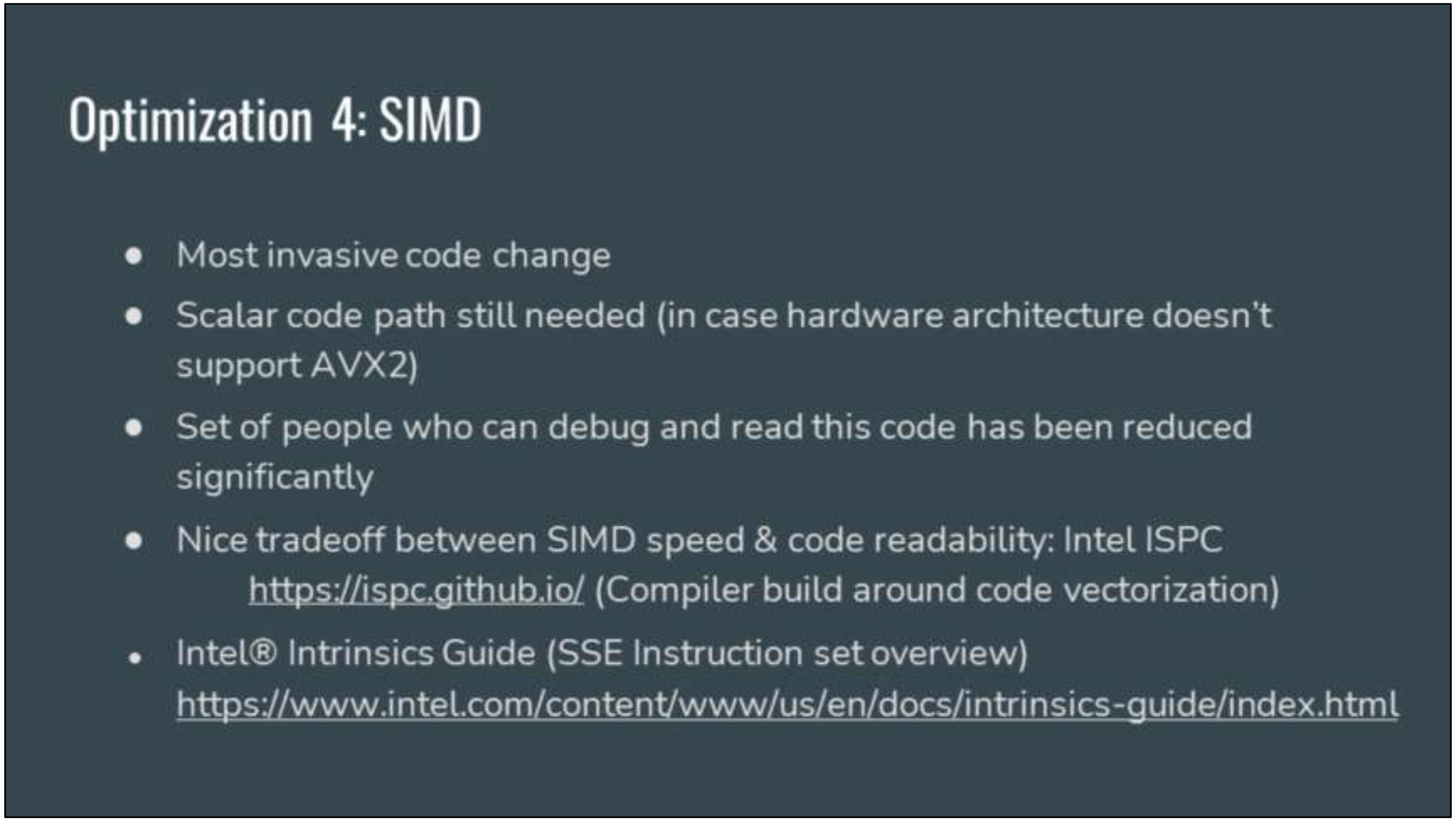
- Most invasive code change
- Scalar code path still needed (in case hardware architecture doesn't support AVX2)
- Set of people who can debug and read this code has been reduced significantly
- Nice tradeoff between SIMD speed & code readability: Intel ISPC
  https://ispc.github.io/ (Compiler build around code vectorization)

While this is nice and we're now in the realm of real-time processing this is by far the most invasive code change since we had to completely rewrite the rotation function to make use of the AVX2 instruction set. Additionally we can't even delete the old code path or else users without AVX2 won't be able to run our program (we might AVX2 a hard requirement though).

On top of that I'd argue that the set of people how can read and debug this code has been reduced, but I might be a little bit pessimistic in this regard.

If you feel more comfortable writing "scalar-looking" C code then you might want to check out ISCP. ISCP is a compiler from Intel that targets a C language with additional extensions for better auto-vectorization.

Additionally there's the Intel Intrinsics Guide, which will show you all the SIMD intrinsics that come with the different instruction sets.



Optimization 4: SIMD

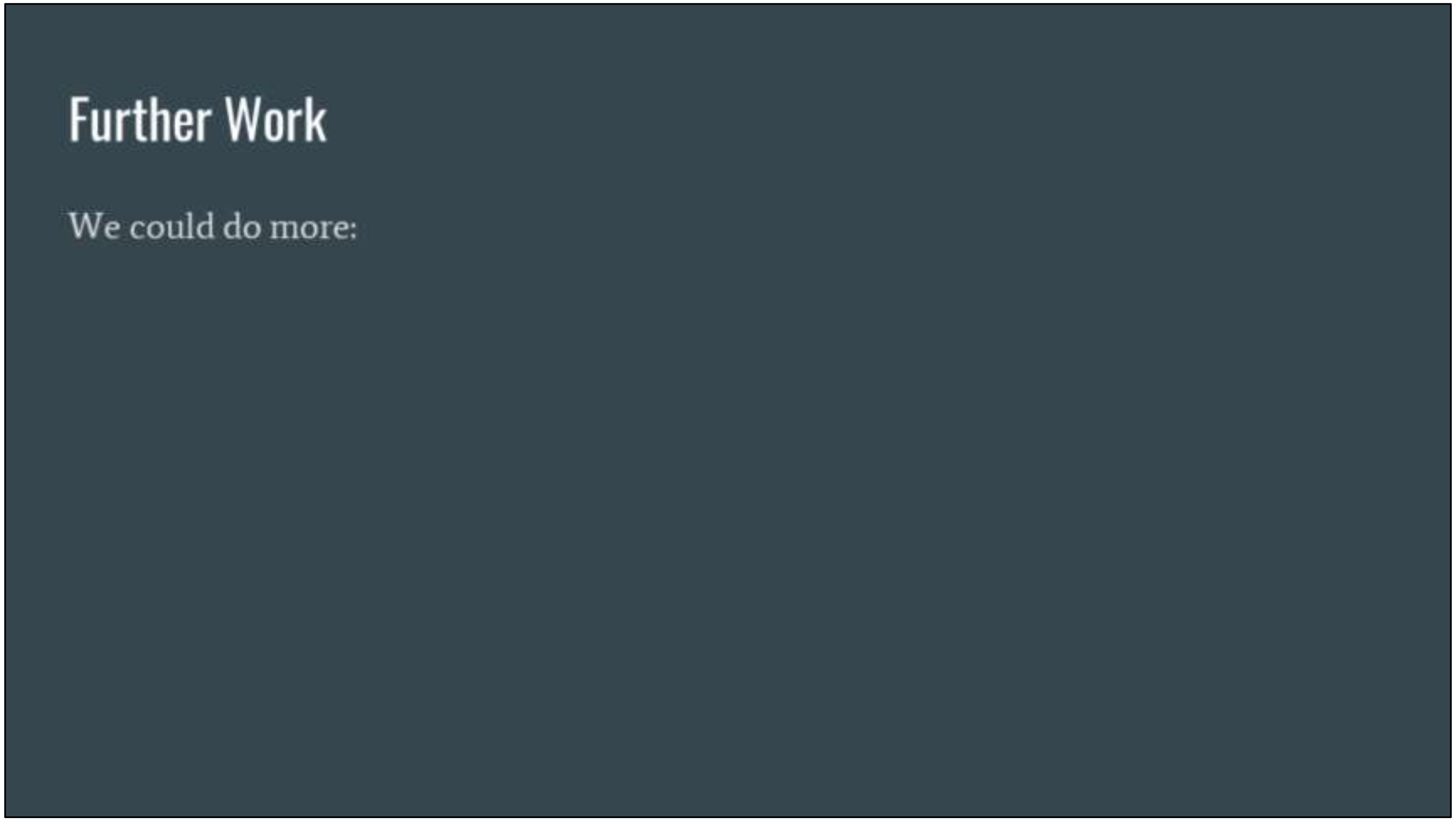
- Most invasive code change
- Scalar code path still needed (in case hardware architecture doesn't support AVX2)
- Set of people who can debug and read this code has been reduced significantly
- Nice tradeoff between SIMD speed & code readability: Intel ISPC
  https://ispc.github.io/ (Compiler build around code vectorization)
- Intel® Intrinsics Guide (SSE Instruction set overview)
  https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

While this is nice and we're now in the realm of real-time processing this is by far the most invasive code change since we had to completely rewrite the rotation function to make use of the AVX2 instruction set. Additionally we can't even delete the old code path or else users without AVX2 won't be able to run our program (we might AVX2 a hard requirement though).

On top of that I'd argue that the set of people how can read and debug this code has been reduced, but I might be a little bit pessimistic in this regard.

If you feel more comfortable writing "scalar-looking" C code then you might want to check out ISCP. ISCP is a compiler from Intel that targets a C language with additional extensions for better auto-vectorization.

Additionally there's the Intel Intrinsics Guide, which will show you all the SIMD intrinsics that come with the different instruction sets.

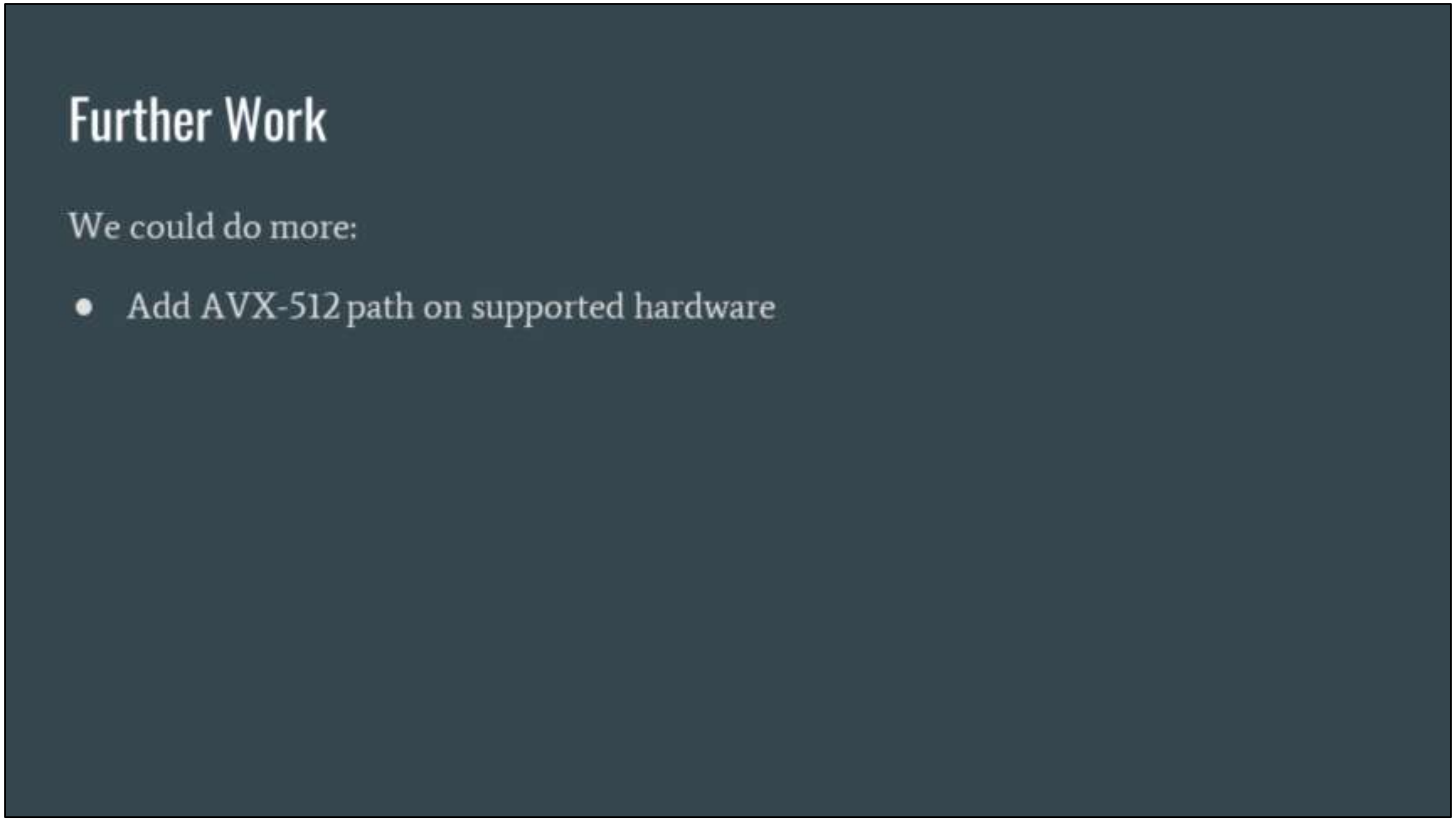


Further Work

We could do more:

After doing the AVX2 implementation we could still do more.
We could add an AVX-512 code path for hardware with AVX-512 support.
Additionally we could let each worker work on it’s own bitmap and merge them later for potentially better cache performance
We could also manually prefetch the pixel data for the next loop iteration by precalculating the pixel coordinates for the next loop iteration.

But it’s also important where to stop. We now achieved real-time performance with our example and doing more here doesn’t make sense right now. We could of course optimize this to death but again, this would also limit the set of people who can work on that code in the future. Also adding yet another code path - like the AVX-512 implementation, will increase the maintenance costs when, for example, a new feature has to be added.



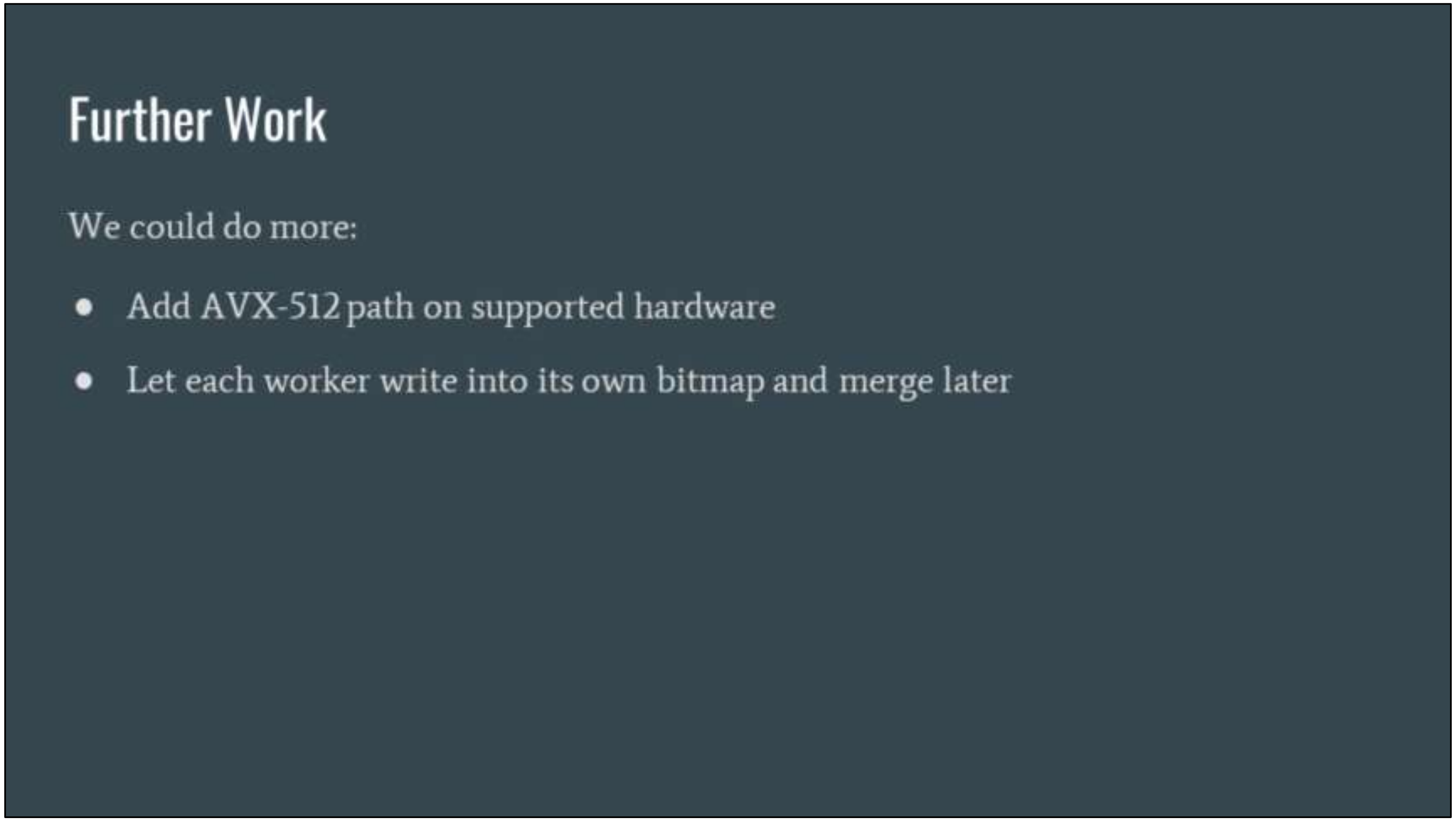
Further Work

We could do more:

- Add AVX-512 path on supported hardware

After doing the AVX2 implementation we could still do more.
We could add an AVX-512 code path for hardware with AVX-512 support.
Additionally we could let each worker work on it’s own bitmap and merge them later for potentially better cache performance
We could also manually prefetch the pixel data for the next loop iteration by precalculating the pixel coordinates for the next loop iteration.

But it’s also important where to stop. We now achieved real-time performance with our example and doing more here doesn’t make sense right now. We could of course optimize this to death but again, this would also limit the set of people who can work on that code in the future. Also adding yet another code path - like the AVX-512 implementation, will increase the maintenance costs when, for example, a new feature has to be added.

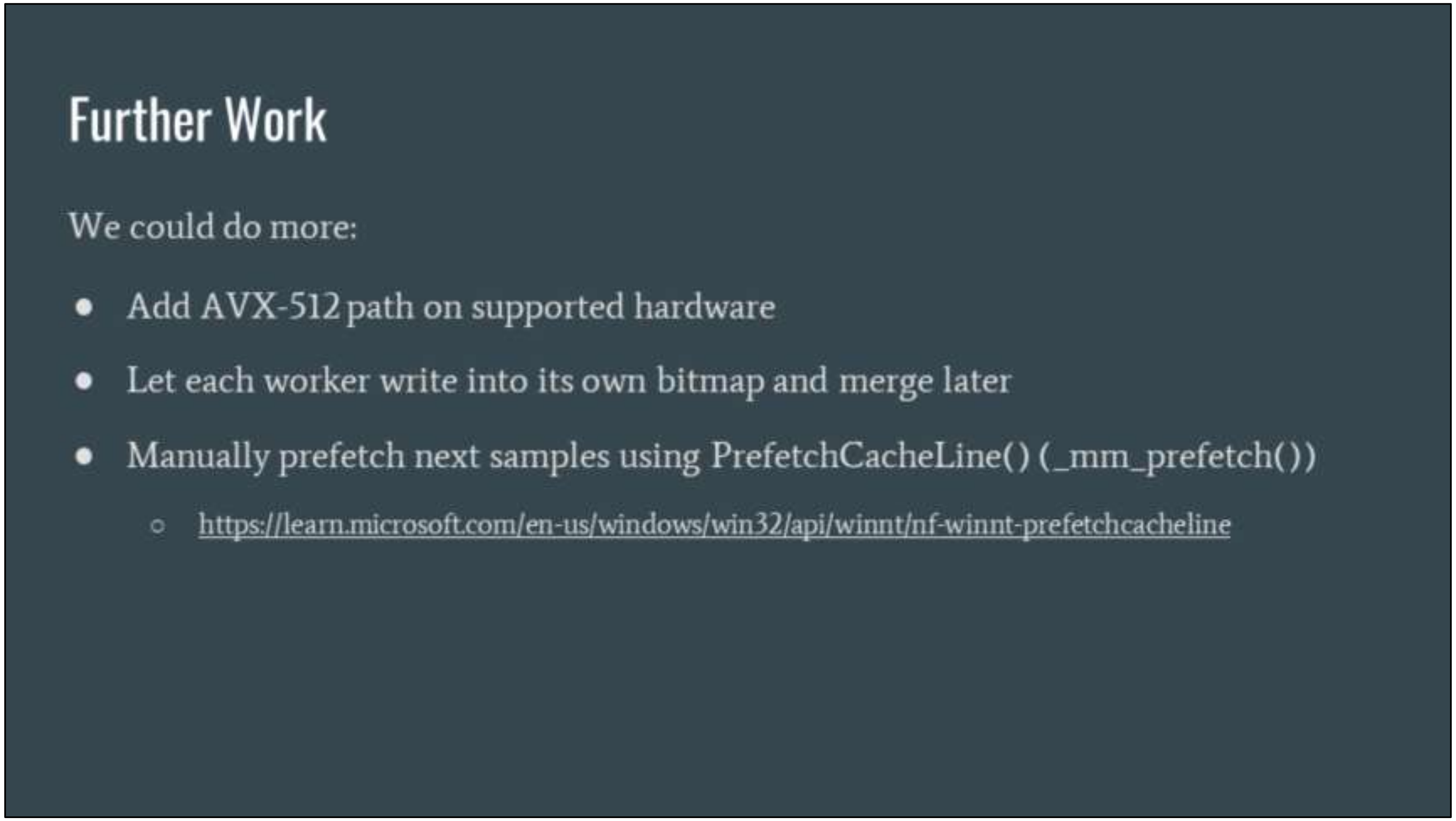

Further Work

We could do more:

- Add AVX-512 path on supported hardware
- Let each worker write into its own bitmap and merge later

After doing the AVX2 implementation we could still do more.
We could add an AVX-512 code path for hardware with AVX-512 support.
Additionally we could let each worker work on it’s own bitmap and merge them later for potentially better cache performance
We could also manually prefetch the pixel data for the next loop iteration by precalculating the pixel coordinates for the next loop iteration.

But it’s also important where to stop. We now achieved real-time performance with our example and doing more here doesn’t make sense right now. We could of course optimize this to death but again, this would also limit the set of people who can work on that code in the future. Also adding yet another code path - like the AVX-512 implementation, will increase the maintenance costs when, for example, a new feature has to be added.



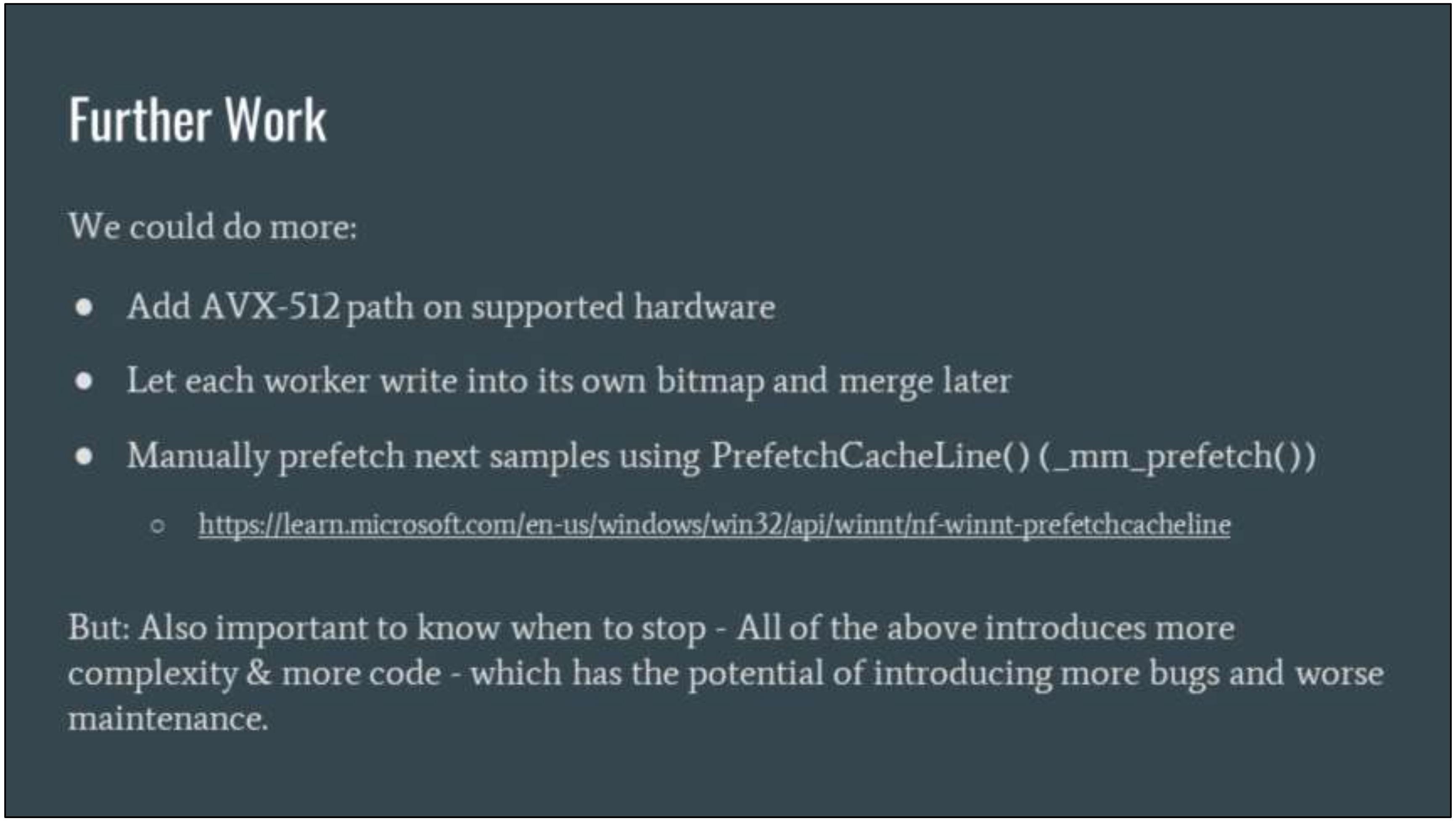
Further Work

We could do more:

- Add AVX-512 path on supported hardware
- Let each worker write into its own bitmap and merge later
- Manually prefetch next samples using PrefetchCacheLine() (_mm_prefetch())
    - https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-prefetchcacheline

After doing the AVX2 implementation we could still do more.
We could add an AVX-512 code path for hardware with AVX-512 support.
Additionally we could let each worker work on it’s own bitmap and merge them later for potentially better cache performance
We could also manually prefetch the pixel data for the next loop iteration by precalculating the pixel coordinates for the next loop iteration.

But it’s also important where to stop. We now achieved real-time performance with our example and doing more here doesn’t make sense right now. We could of course optimize this to death but again, this would also limit the set of people who can work on that code in the future. Also adding yet another code path - like the AVX-512 implementation, will increase the maintenance costs when, for example, a new feature has to be added.



Further Work

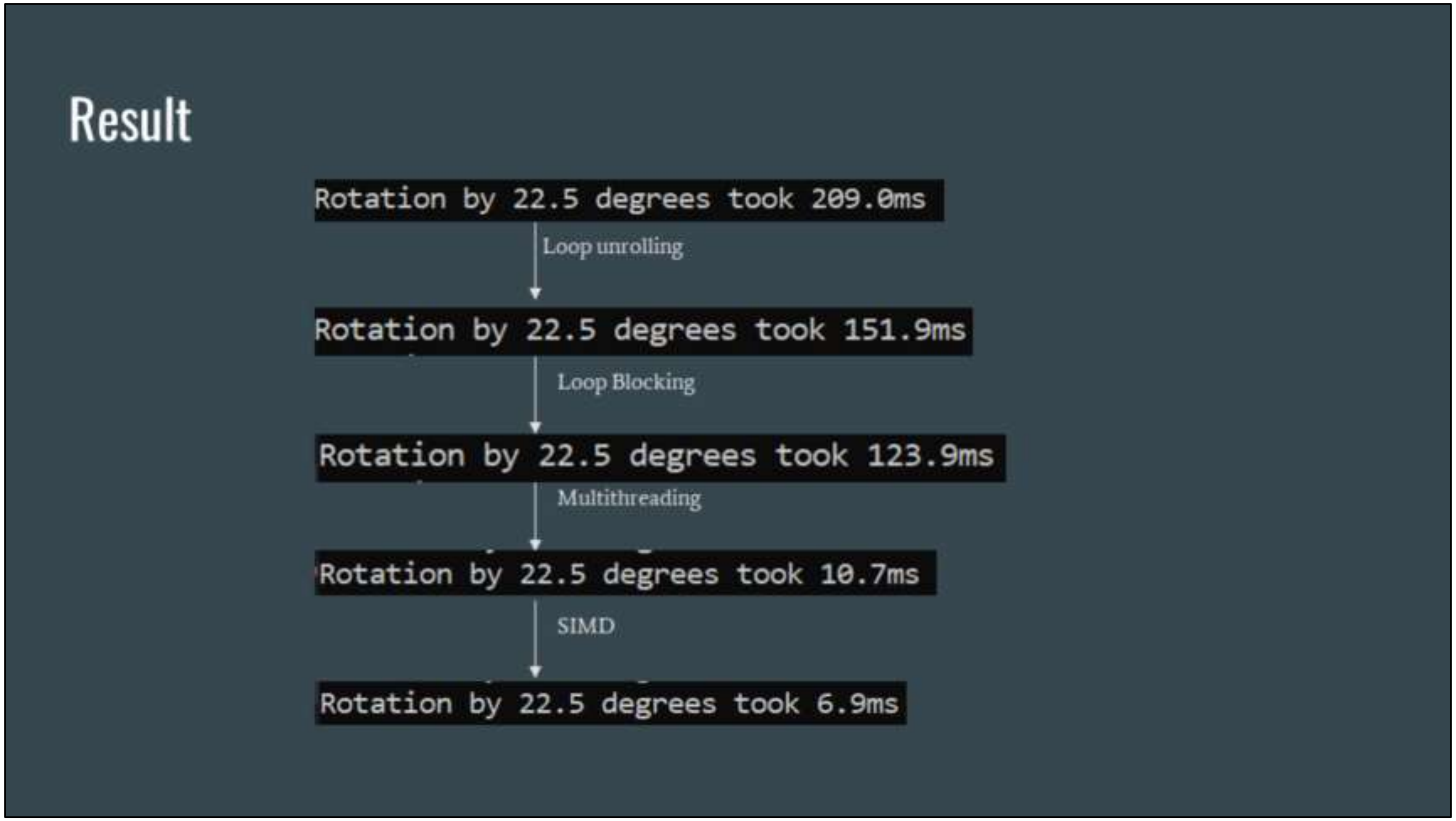
We could do more:

- Add AVX-512 path on supported hardware
- Let each worker write into its own bitmap and merge later
- Manually prefetch next samples using PrefetchCacheLine() (_mm_prefetch())
    - https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-prefetchcacheline

But: Also important to know when to stop - All of the above introduces more complexity & more code - which has the potential of introducing more bugs and worse maintenance.

After doing the AVX2 implementation we could still do more.
We could add an AVX-512 code path for hardware with AVX-512 support.
Additionally we could let each worker work on it's own bitmap and merge them later for potentially better cache performance
We could also manually prefetch the pixel data for the next loop iteration by precalculating the pixel coordinates for the next loop iteration.

But it's also important where to stop. We now achieved real-time performance with our example and doing more here doesn't make sense right now. We could of course optimize this to death but again, this would also limit the set of people who can work on that code in the future. Also adding yet another code path - like the AVX-512 implementation, will increase the maintenance costs when, for example, a new feature has to be added.



Result

```
Rotation by 22.5 degrees took 209.0ms
```

```
Rotation by 22.5 degrees took 151.9ms
```

```
Rotation by 22.5 degrees took 123.9ms
```

```
Rotation by 22.5 degrees took 10.7ms
```

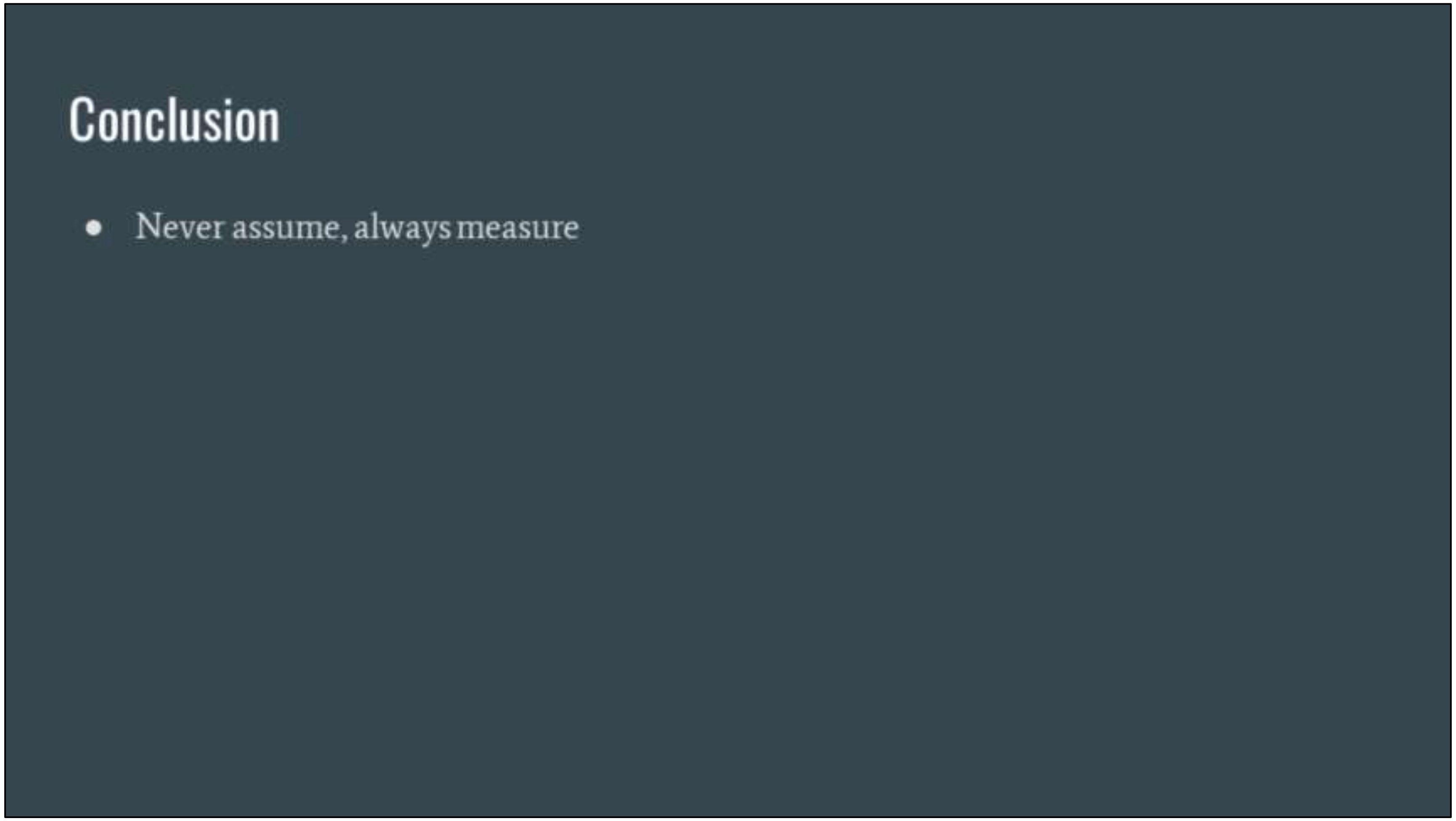
```
Rotation by 22.5 degrees took 6.9ms
```

So let's quickly recap where we started and how we got to our current result.
We first started with the super naive scalar version and then used VTune to figure out different bottlenecks.
The first optimization we applied was the loop unrolling.

After that we looked at the memory access analysis and added loop blocking to make the memory access pattern more local to decrease the amount of cache misses.

Then we took our single-threaded algorithm and made it work in a multi-core context by utilizing a job system.

Finally, we pulled out the SIMD hammer and created an AVX2 implementation of the rotation code which effectively works on 8 pixels in parallel.

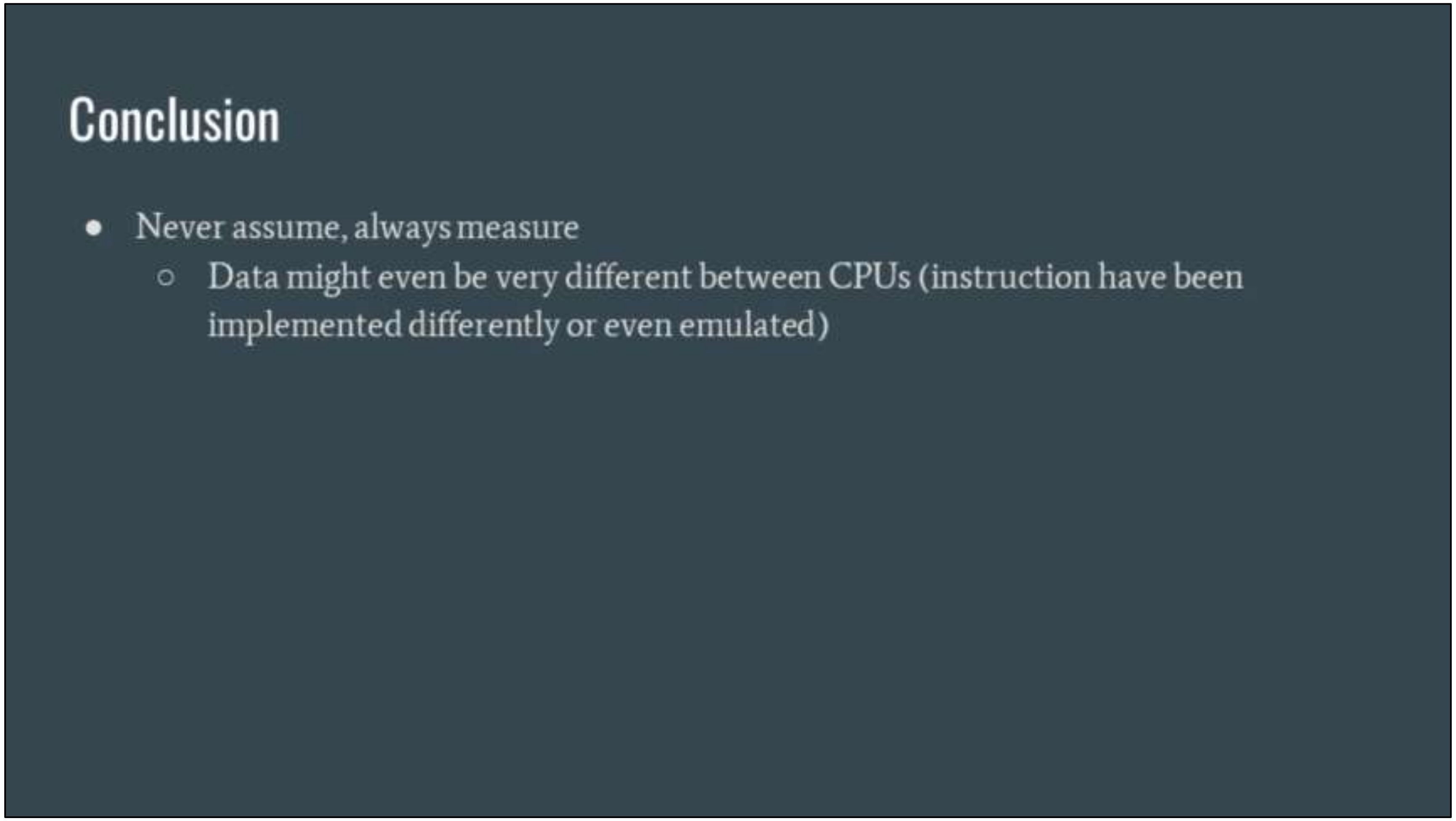


Conclusion

- Never assume, always measure

Let’s talk about somethings that I hope you can take away from this talk.
First of all: never assume, always measure!
This is even true if you try your code on a new CPU.

As an example:
I looked into the performance data of the instruction PDEP some while ago and found out the instruction took 2 orders of magnitude longer on AMD Zen2 vs AMD Zen3 CPUs. This could come down to several factors like the instruction being emulated in microcode on Zen2 and actually there being dedicated hardware for this instruction in Zen3 CPUs. But this is only a guess.

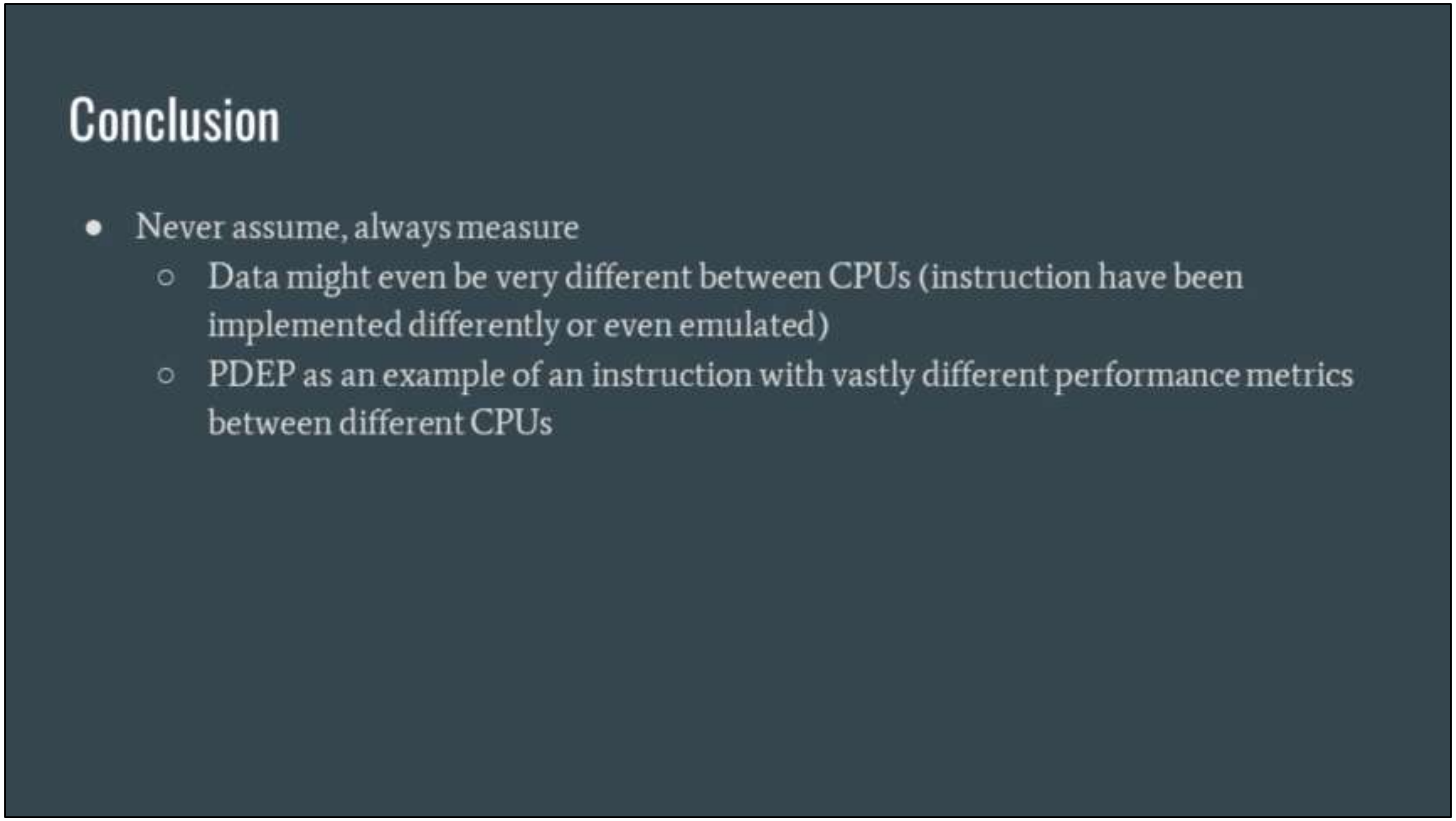


Conclusion

- Never assume, always measure
  - Data might even be very different between CPUs (instruction have been implemented differently or even emulated)

Let’s talk about somethings that I hope you can take away from this talk.
First of all: never assume, always measure!
This is even true if you try your code on a new CPU.

As an example:
I looked into the performance data of the instruction PDEP some while ago and found out the instruction took 2 orders of magnitude longer on AMD Zen2 vs AMD Zen3 CPUs. This could come down to several factors like the instruction being emulated in microcode on Zen2 and actually there being dedicated hardware for this instruction in Zen3 CPUs. But this is only a guess.

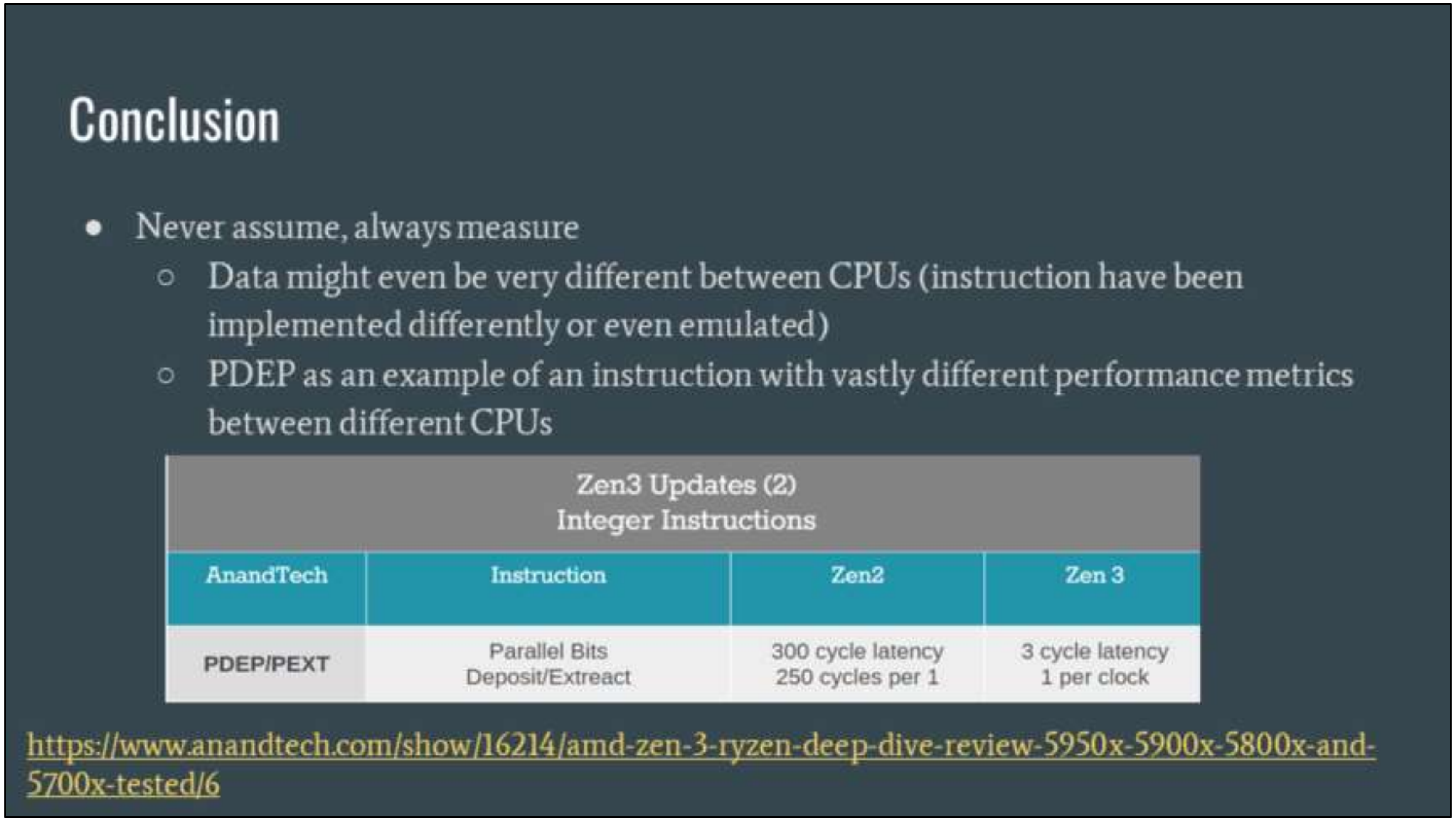


Conclusion

- Never assume, always measure
  - Data might even be very different between CPUs (instruction have been implemented differently or even emulated)
  - PDEP as an example of an instruction with vastly different performance metrics between different CPUs

Let’s talk about somethings that I hope you can take away from this talk.
First of all: never assume, always measure!
This is even true if you try your code on a new CPU.

As an example:
I looked into the performance data of the instruction PDEP some while ago and found out the instruction took 2 orders of magnitude longer on AMD Zen2 vs AMD Zen3 CPUs. This could come down to several factors like the instruction being emulated in microcode on Zen2 and actually there being dedicated hardware for this instruction in Zen3 CPUs. But this is only a guess.



Conclusion

- Never assume, always measure
  - Data might even be very different between CPUs (instruction have been implemented differently or even emulated)
  - PDEP as an example of an instruction with vastly different performance metrics between different CPUs

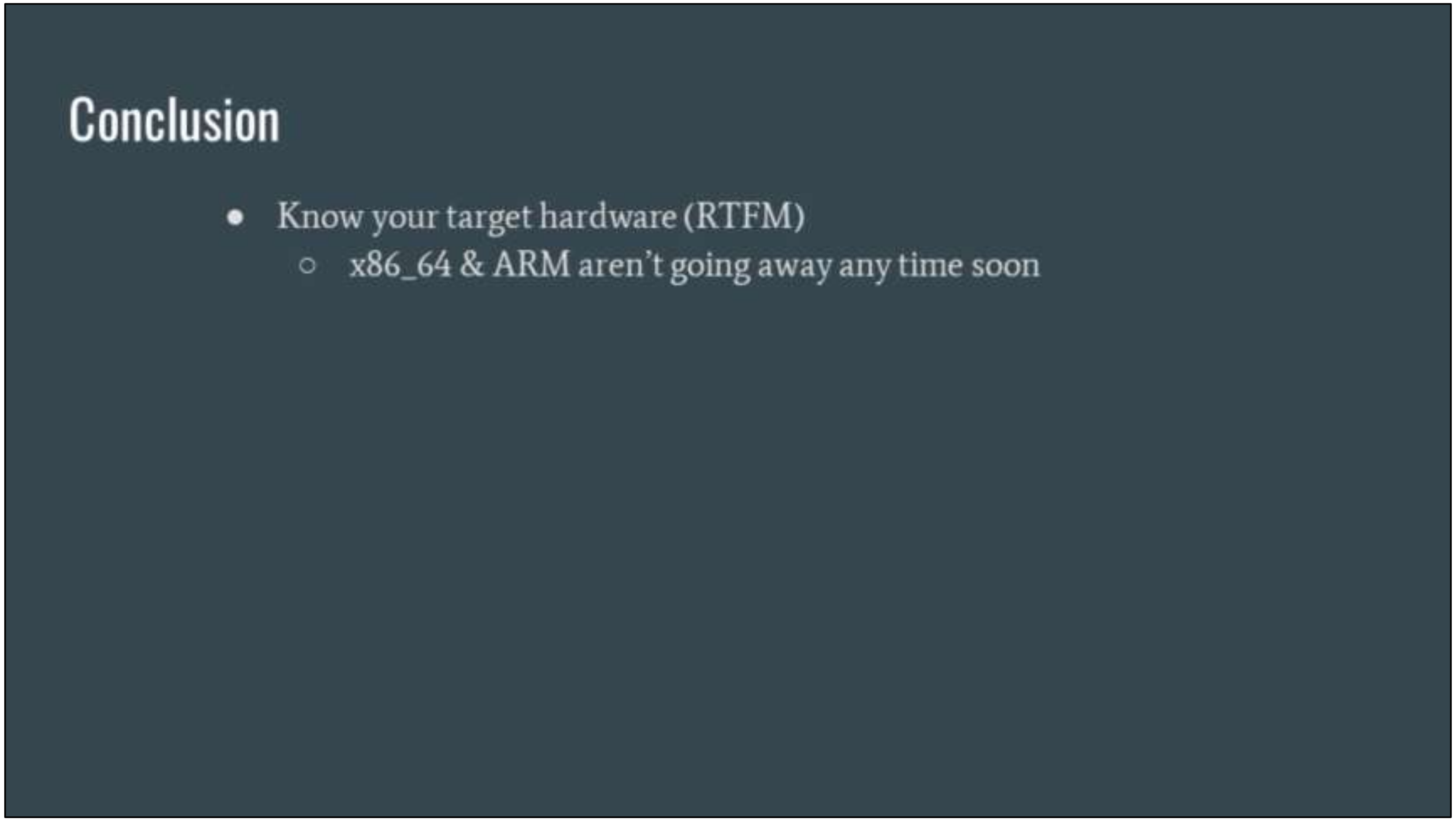
Zen3 Updates (2)
Integer Instructions

AnandTech	Instruction	Zen2	Zen 3
PDEP/PEXT	Parallel Bits Deposit/Extreact	300 cycle latency 250 cycles per 1	3 cycle latency 1 per clock

https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested/6

Let’s talk about somethings that I hope you can take away from this talk.
First of all: never assume, always measure!
This is even true if you try your code on a new CPU.

As an example:
I looked into the performance data of the instruction PDEP some while ago and found out the instruction took 2 orders of magnitude longer on AMD Zen2 vs AMD Zen3 CPUs. This could come down to several factors like the instruction being emulated in microcode on Zen2 and actually there being dedicated hardware for this instruction in Zen3 CPUs. But this is only a guess.



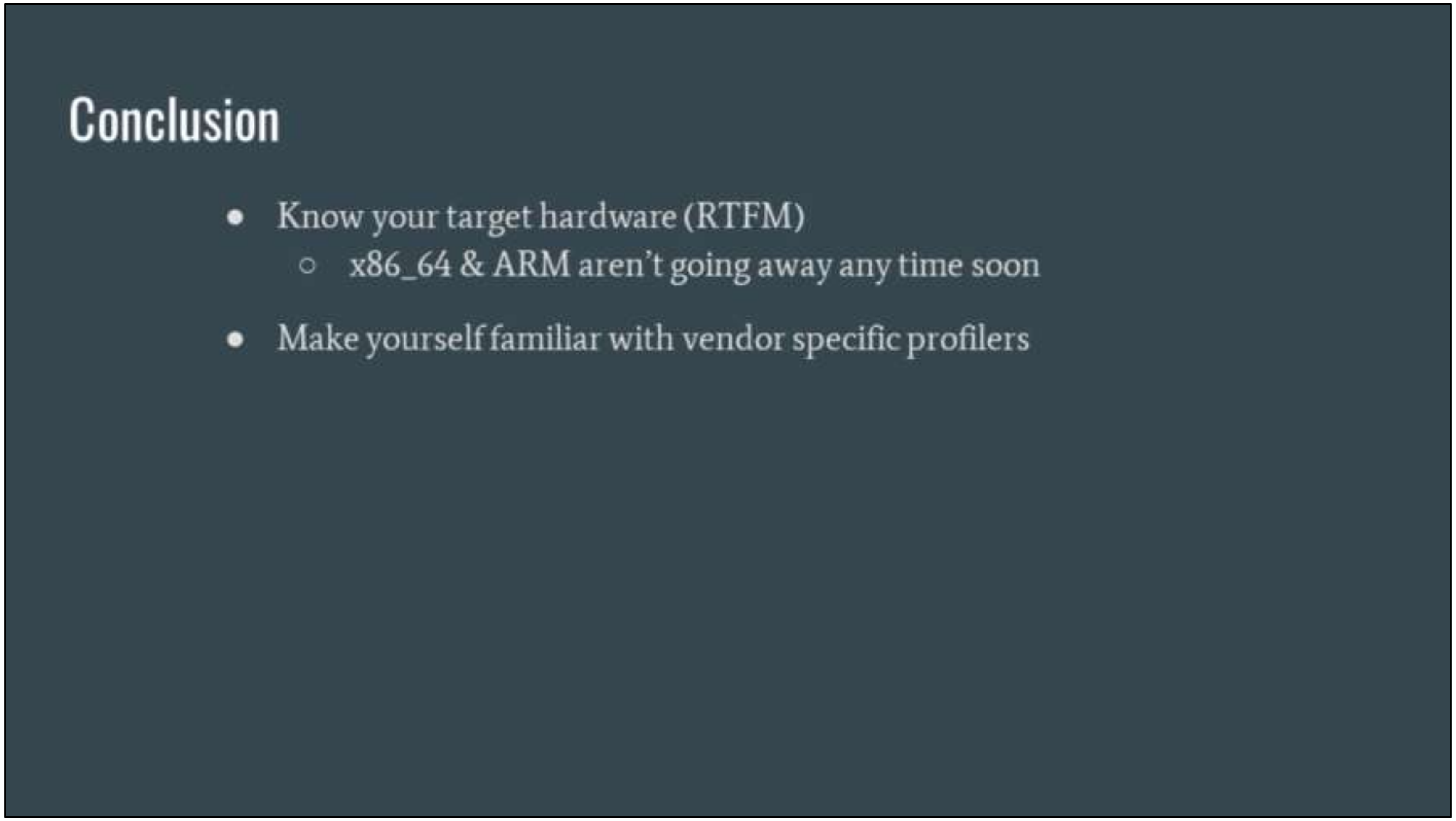
Conclusion

- Know your target hardware (RTFM)
  - x86_64 & ARM aren't going away any time soon

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.



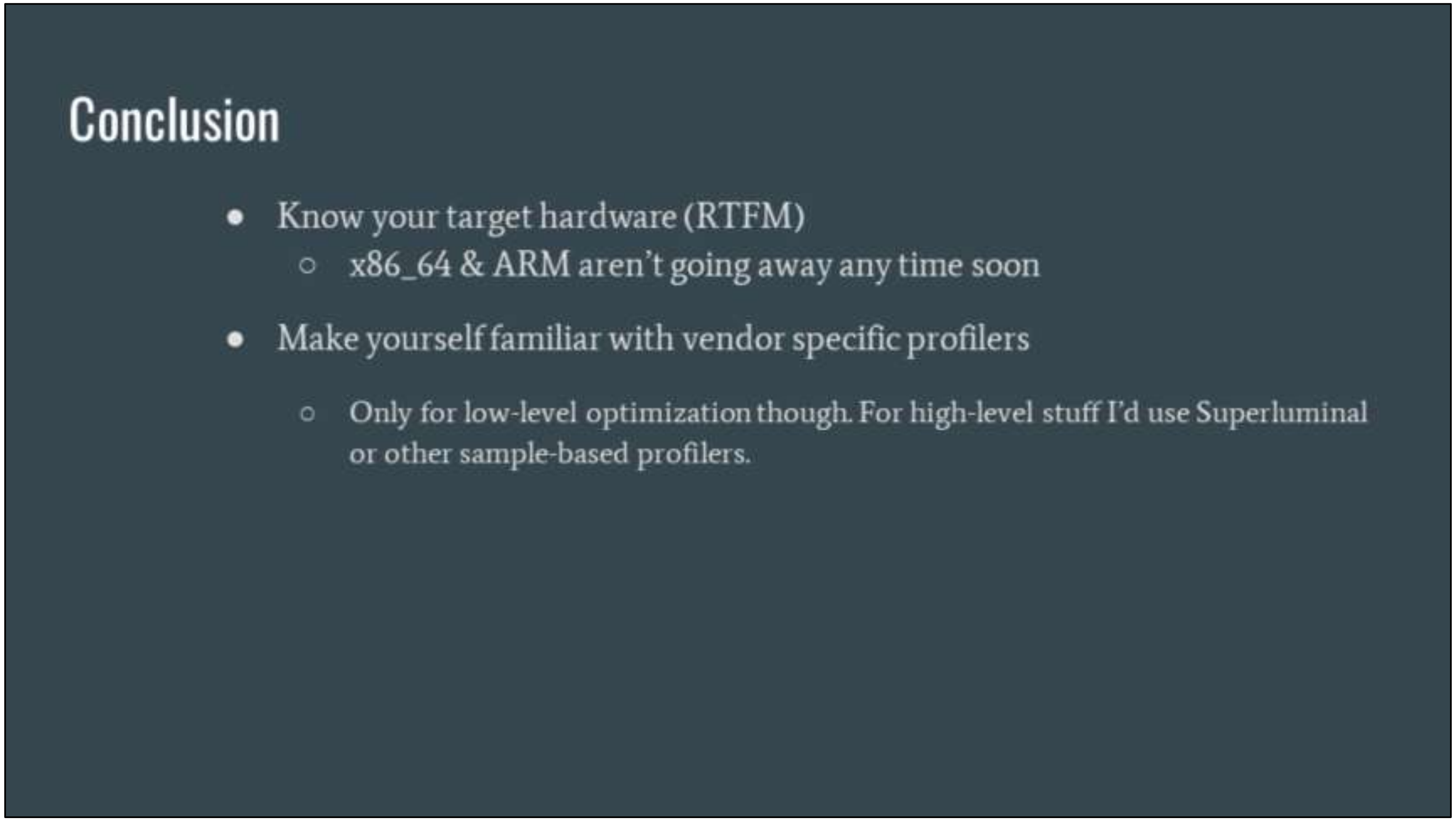
Conclusion

- Know your target hardware (RTFM)
  - x86_64 & ARM aren't going away any time soon
- Make yourself familiar with vendor specific profilers

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.



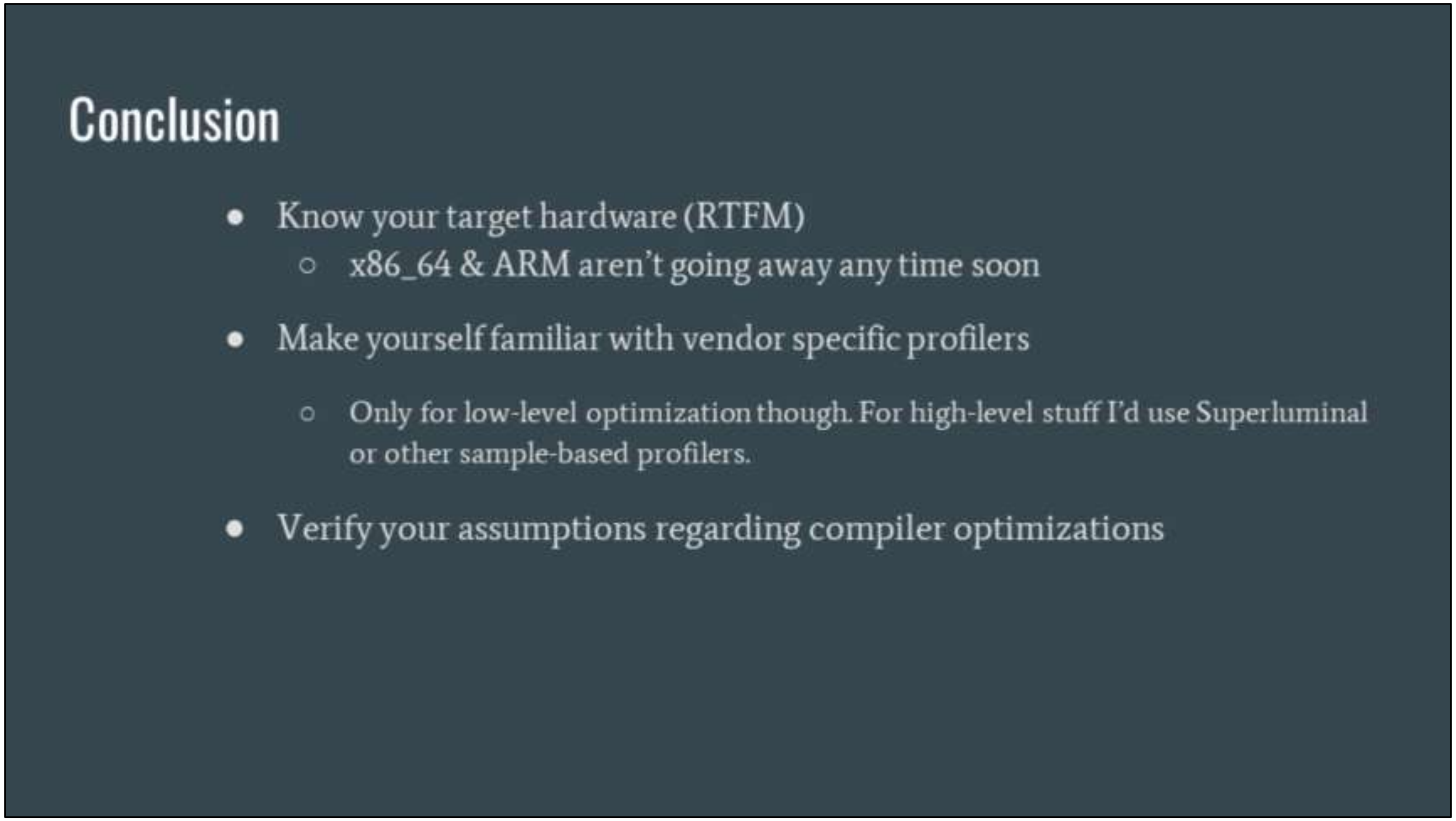
Conclusion

- Know your target hardware (RTFM)
    - x86_64 & ARM aren't going away any time soon
- Make yourself familiar with vendor specific profilers
    - Only for low-level optimization though. For high-level stuff I'd use Superluminal or other sample-based profilers.

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.



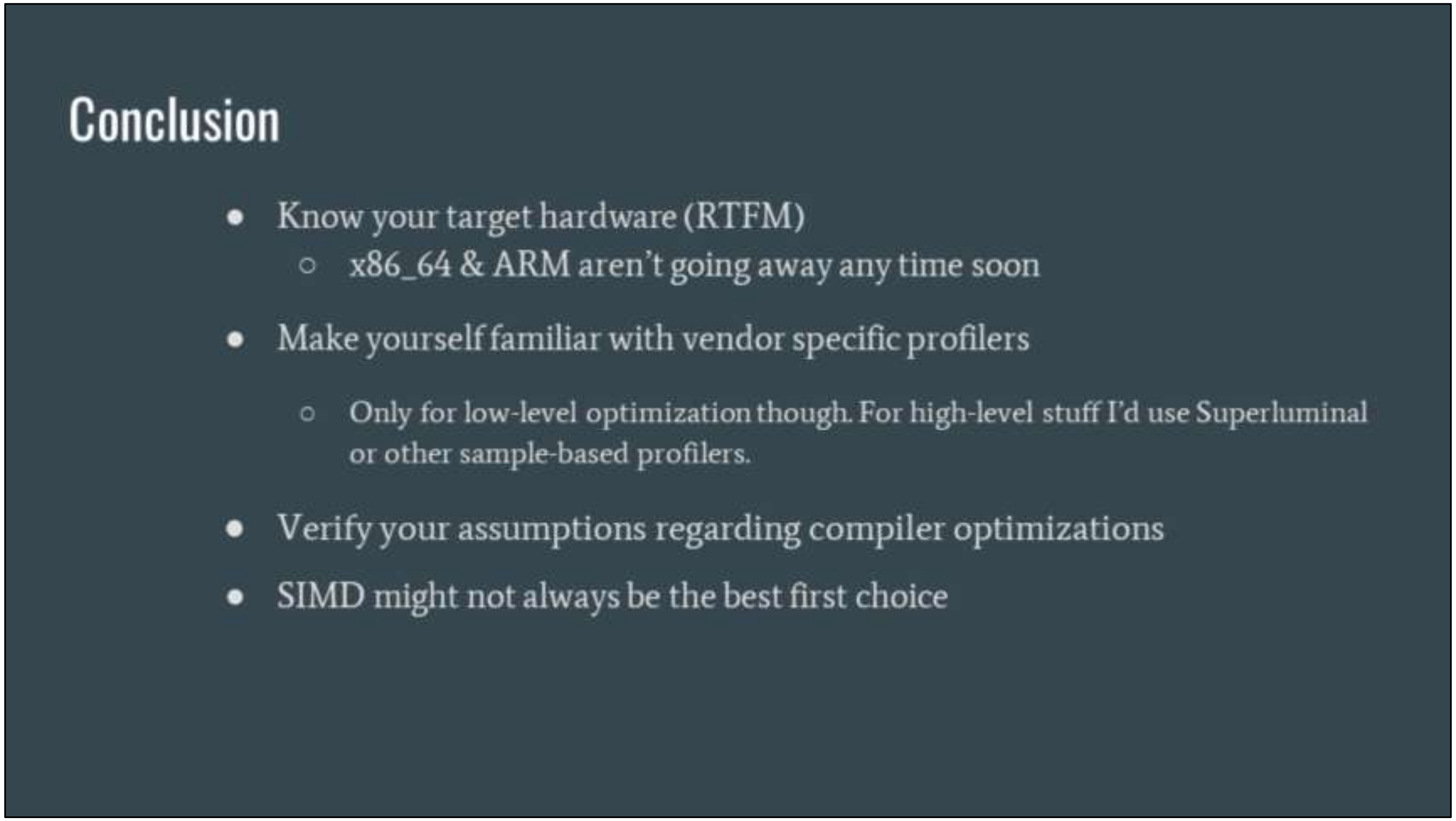
Conclusion

- Know your target hardware (RTFM)
    - x86_64 & ARM aren't going away any time soon
- Make yourself familiar with vendor specific profilers
    - Only for low-level optimization though. For high-level stuff I'd use Superluminal or other sample-based profilers.
- Verify your assumptions regarding compiler optimizations

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.



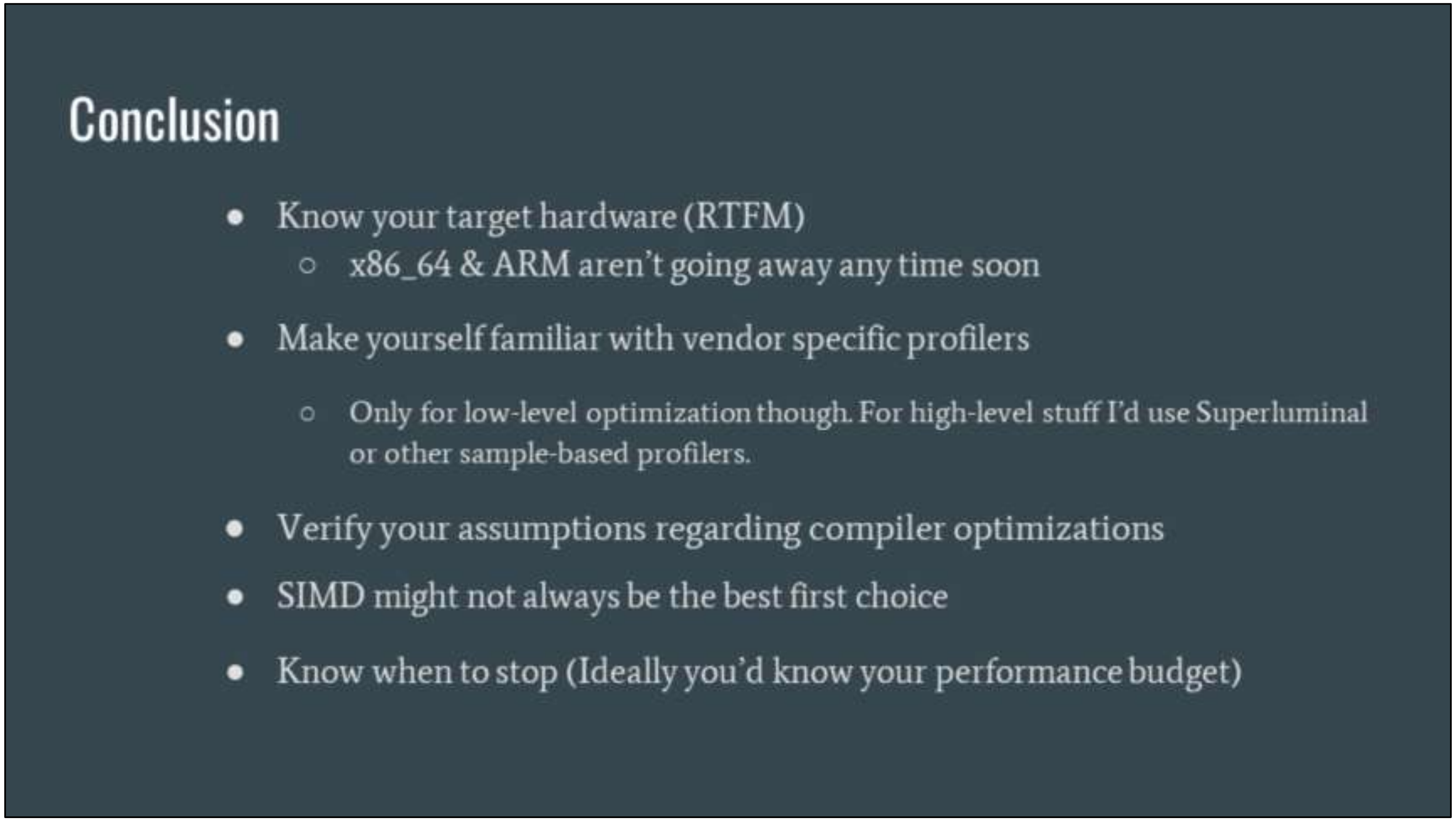
Conclusion

- Know your target hardware (RTFM)
  - x86_64 & ARM aren't going away any time soon
- Make yourself familiar with vendor specific profilers
  - Only for low-level optimization though. For high-level stuff I'd use Superluminal or other sample-based profilers.
- Verify your assumptions regarding compiler optimizations
- SIMD might not always be the best first choice

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.



Conclusion

- Know your target hardware (RTFM)
  - x86_64 & ARM aren't going away any time soon
- Make yourself familiar with vendor specific profilers
  - Only for low-level optimization though. For high-level stuff I'd use Superluminal or other sample-based profilers.
- Verify your assumptions regarding compiler optimizations
- SIMD might not always be the best first choice
- Know when to stop (Ideally you'd know your performance budget)

It's also important that you know your target hardware when you want to apply low-level optimizations. I can only recommend taking a look at the documentation. x86_64 and ARM are here to stay.

It's also valuable to make yourself familiar with vendor specific profilers. If you've got an Intel CPU at home, download VTune and try to profile a piece of code of yours.

Also make sure to verify and test your assumptions about compiler optimizations. There's lots of half-truths out there about this, be sure to not blindly trust the compiler.

Where to go from here?

- Mike Acton "Data-Oriented Design and C++"
  https://www.youtube.com/watch?v=rX0ItVEVjHc
- Casey Muratori "'Clean Code', Horrible Performance"
  https://www.youtube.com/watch?v=tD5NrevFtbU&t=1s
- Jon Blow "Preventing the Collapse of Civilization"
  https://www.youtube.com/watch?v=q3OCFfDStgM
- Ulrich Drepper "What every programmer should know about memory"
  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf
- John L. Hennessy, David A. Patterson "Computer Architecture"
  https://www.oreilly.com/library/view/computer-architecture-5th/9780123838735/
- Scott Meyers "CPU Cache and why you care"
  https://www.youtube.com/watch?v=WDIkqP4JbkE

Thanks for your attention!

Reach out in case of questions!

- @FelixK_15
- @FelixK15 (gamedev.place)
- Felix Klinge
- felix [at] k15tech [dot] com