SIMD at Insomniac Games

(How we do the shuffle)

Andreas Fredriksson
Lead Engine Programmer, Insomniac Games

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Hi, I’m Andreas!

- Lead programmer in Insomniac's Core team
- Small team - in-house engine
- Lots of diverse programming challenges

Good morning everyone!
My name is Andreas Fredriksson.
I'm one of the lead programmers at the Core team at Insomniac Games.
The Core team is pretty small, about 18 people total.
One of the cool things about Insomniac is that we have an in-house engine.
This means we get to work on a pretty diverse set of problems.
And when working on performance in our engine, we tend to use a lot of SIMD which is the topic of this talk.

Talk Outline

- Background
- Gameplay Example
- Techniques (tricks!)
- Best Practices
- Resources + Q & A

Today we're first going to cover some background on SIMD.
Then we're going to look at an example of how SIMD and data design can be applied to a gameplay problem.
And then we'll get into the more technical meat of the talk, which is centered around a number of tricks.
We'll move on to some best practices we've found helpful.
And then we'll wrap up with pointers to more resources and if we have time, some Q & A.
The slides will be available online, so there's no need to take everything down or photograph the slides.

Why CPU SIMD?

So CPU SIMD - what's that all about?
Maybe you've heard that everything is running on the GPU nowadays.
That's certainly not true. The truth is that CPU SIMD continues to rule
in a number of problem spaces.
First of all, we have way better latency on the CPU.
Typically if we want to transform some data on the GPU and later use it on the
CPU we're going to hit this incredible latency wall, especially on PC.
GPUs are designed for throughput, not latency.
This means that CPU SIMD is good speedup even for small batches. If we have
just a handful of things, CPU SIMD is going to speed us up, whereas
transforming just a handful of things on the GPU is rarely a good idea.
Plus the GPU tends to be full of graphics work, so it can get crowded on there.

And another reason to use CPU SIMD is that we have these GHz cores sitting around.
We want to maximize the usage of them, and not leave performance on the table.

And it turns out these CPU cores are really fast if we feed them good data and
write SIMD code for them.

Why CPU SIMD?

- Isn’t everything GPGPU nowadays? (Nope.)

So CPU SIMD - what's that all about?
Maybe you've heard that everything is running on the GPU nowadays.
That's certainly not true. The truth is that CPU SIMD continues to rule
in a number of problem spaces.
First of all, we have way better latency on the CPU.
Typically if we want to transform some data on the GPU and later use it on the
CPU we're going to hit this incredible latency wall, especially on PC.
GPUs are designed for throughput, not latency.
This means that CPU SIMD is good speedup even for small batches. If we have
just a handful of things, CPU SIMD is going to speed us up, whereas
transforming just a handful of things on the GPU is rarely a good idea.
Plus the GPU tends to be full of graphics work, so it can get crowded on there.

And another reason to use CPU SIMD is that we have these GHz cores sitting around.
We want to maximize the usage of them, and not leave performance on the table.

And it turns out these CPU cores are really fast if we feed them good data and
write SIMD code for them.

Why CPU SIMD?

- Isn’t everything GPGPU nowadays? (Nope.)
- We have a bunch of GHz cores sitting around

So CPU SIMD - what's that all about?
Maybe you've heard that everything is running on the GPU nowadays.
That's certainly not true. The truth is that CPU SIMD continues to rule
in a number of problem spaces.
First of all, we have way better latency on the CPU.
Typically if we want to transform some data on the GPU and later use it on the
CPU we're going to hit this incredible latency wall, especially on PC.
GPUs are designed for throughput, not latency.
This means that CPU SIMD is good speedup even for small batches. If we have
just a handful of things, CPU SIMD is going to speed us up, whereas
transforming just a handful of things on the GPU is rarely a good idea.
Plus the GPU tends to be full of graphics work, so it can get crowded on there.

And another reason to use CPU SIMD is that we have these GHz cores sitting around.
We want to maximize the usage of them, and not leave performance on the table.

And it turns out these CPU cores are really fast if we feed them good data and
write SIMD code for them.

Why CPU SIMD?

- Isn’t everything GPGPU nowadays? (Nope.)
- We have a bunch of GHz cores sitting around
- Don’t leave performance on the table

So CPU SIMD - what's that all about?
Maybe you've heard that everything is running on the GPU nowadays.
That's certainly not true. The truth is that CPU SIMD continues to rule
in a number of problem spaces.
First of all, we have way better latency on the CPU.
Typically if we want to transform some data on the GPU and later use it on the
CPU we're going to hit this incredible latency wall, especially on PC.
GPUs are designed for throughput, not latency.
This means that CPU SIMD is good speedup even for small batches. If we have
just a handful of things, CPU SIMD is going to speed us up, whereas
transforming just a handful of things on the GPU is rarely a good idea.
Plus the GPU tends to be full of graphics work, so it can get crowded on there.

And another reason to use CPU SIMD is that we have these GHz cores sitting around.
We want to maximize the usage of them, and not leave performance on the table.

And it turns out these CPU cores are really fast if we feed them good data and
write SIMD code for them.

SIMD at Insomniac Games

- Long history of SIMD programming in the studio
- Focus on SSE programming for this cycle
- Lots of old best practices don’t apply to SSE

And it's this idea of not leaving performance on the table that has been driving the use of SIMD at Insomniac Games.
As the studio has moved from PS2, PS3 with the SPUs of course and xbox 360 we've always relied heavily on the SIMD instruction sets these platforms have offered.
Now with Xbox One we're turning our attention to SSE. And that's interesting because the SIMD code we write applies to our tools as well. If we improve our engine performance it also helps our tools build on PC.
We found though as we were porting a lot of old SIMD code that we needed a new set of best practices to work better with SSE, and I'll try to cover some of those in this talk.

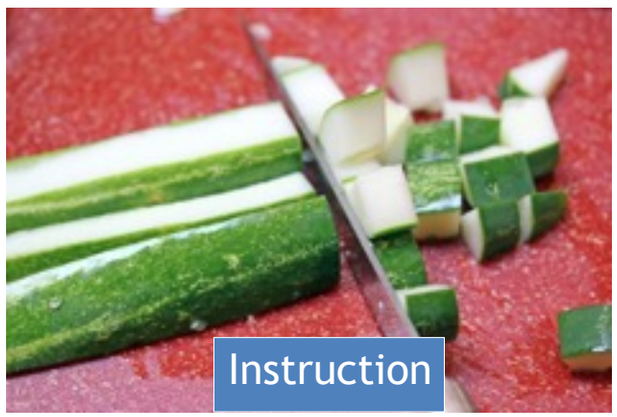
SIMD

SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

SIMD

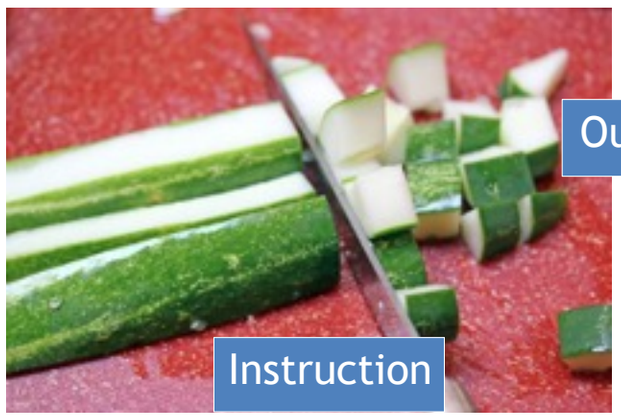
SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

SIMD



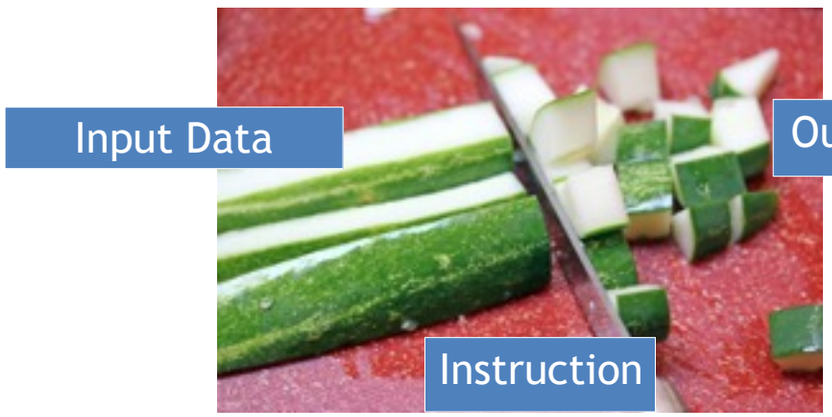
SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

SIMD



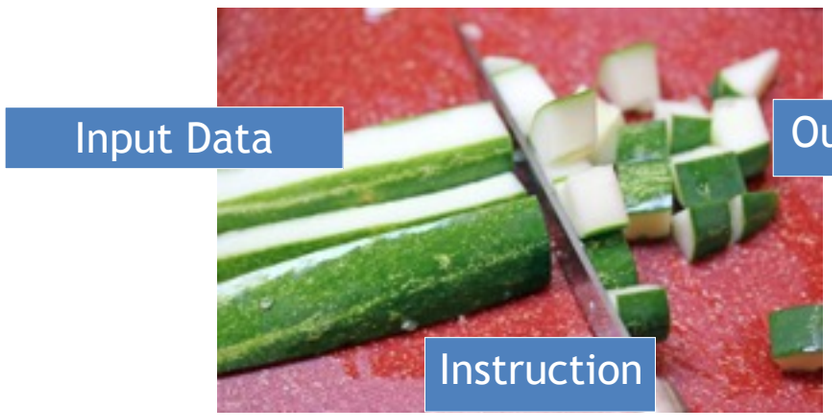
SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

SIMD



SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

SIMD



.. It’s just like dicing veggies!

SIMD stands for Single Instruction, Multiple Data.
To understand what SIMD is, we're going to have to go into the kitchen.
We are chopping veggies. I guess this is a zhuccini.
Consider for a moment what we're doing when we are chopping veggies.
We're trying to minimize the number of times we move the knife.
** The knife here is the instruction stream. It's the thing that does the work.
** With each stroke of the knife we're getting multiple pieces of output data. This is the MD in SIMD, the multiple pieces of output data.
And to get this nice parallel chopping going, we have to preprocess our input data. ** In this case we've cut our zhuccini along the length of it so we can get these benefits from the next stage of processing.
Consider how obvious this approach is in cooking. We don't even think about it.
The alternative we could have here is to say: no no, I'm going to cut each little piece of zhuccini out, by itself, put down the knife and move the little cube somewhere and then go cut the next cube.
And yet that's the sort of thing we do in software all the time.
We say that we COULD maybe do this in SIMD or on the GPU, but it's really
elegant now or really abstract and I don't want to change the code.
If we tried to do that in the kitchen with dinner guests, we might need that knife to defend ourselves.
I don't think they're going to care how elegantly we moved the knife if it means they have to wait hours for dinner.
** So if we do it correctly, I think SIMD should be just like dicing veggies.

Options for SSE and AVX SIMD

- Compiler auto-vectorization
- ISPC
- Intrinsics
- Assembly

So SIMD is important. How can we get those nice 2, 4 or 8x speedups?
Well, we can ignore the fact that we're running on actual hardware, and pretend that our compilers can do it for us. We'll see how that works out in a minute.
We can reach for some specialized tool, like ISPC which I'll cover briefly.
Or we can embrace the CPU architecture and use intrinsics or assembly.

Auto-vectorization

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    acc += input[i];
  }

  return acc;
}
```

Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)

So auto-vectorization is the kind of cool idea that - yeah we already have all this code so wouldn't it be sweet if our compilers just turned that into SIMD for us?
I decided to test the state of the art, so I'm using a recent version of clang
here and I wrote this function which is a simple routine to sum up a bunch of
floating point numbers.
I expected the compiler to go to town here, like there's NOTHING preventing it
from doing aggressive SIMD on this piece of code.
** Here's what it spit out. It's a scalar add!
I'm scratching my head and then I realized that oooh, it's because it's floats.
Because compilers have to play by the rules, they can't reorder floating point terms in an addition, because the result might not be bitwise equivalent.

Auto-vectorization

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    acc += input[i];
  }

  return acc;
}
```

```
addss   xmm0, dword ptr [rdi]
```

✗ SCALAR

Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)

So auto-vectorization is the kind of cool idea that - yeah we already have all this code so wouldn't it be sweet if our compilers just turned that into SIMD for us?
I decided to test the state of the art, so I'm using a recent version of clang
here and I wrote this function which is a simple routine to sum up a bunch of
floating point numbers.
I expected the compiler to go to town here, like there's NOTHING preventing it
from doing aggressive SIMD on this piece of code.
** Here's what it spit out. It's a scalar add!
I'm scratching my head and then I realized that oooh, it's because it's floats.
Because compilers have to play by the rules, they can't reorder floating point terms in an addition, because the result might not be bitwise equivalent.

Now with 100% more -ffast-math

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    acc += input[i];
  }

  return acc;
}
```

If we throw in the fast-math option though we give them that leeway.
In this case it's like treating the whole body when we have a headache, but we'll play along.
So if we add the fast-math option clang will actually turn this into SIMD code, and it even unrolled it for us.

Now with 100% more -ffast-math

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    acc += input[i];
  }

  return acc;
}
```

```
movaps   xmm2, xmm0
movaps   xmm3, xmm1
movups   xmm1, xmmword ptr [rdi + 4*rdx]
movups   xmm0, xmmword ptr [rdi + 4*rdx + 16]
addps    xmm1, xmm3
addps    xmm0, xmm2
add      rdx, 8
cmp      rcx, rdx
jne      loop
```

✓SIMD

If we throw in the fast-math option though we give them that leeway.
In this case it's like treating the whole body when we have a headache, but we'll play along.
So if we add the fast-math option clang will actually turn this into SIMD code, and it even unrolled it for us.

Let’s make some changes..

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
  }

  return acc;
}
```

The next thing I wanted to see was how well it dealt with branches.
So in our routine we're making a small change here where we're only summing up floats under some certain value.
And what do you know, it actually comes back as SIMD code.
We get an unrolling here as well. It might not be exactly what I'd write by
hand, but hey, we didn't actually have to write anything.
If you leave the talk now, you're making a mistake.
And the reason is that..

Let's make some changes..

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
  }

  return acc;
}
```

✓SIMD

```
movups  xmm3, xmmword ptr [rdi + 4*rdx]
movups  xmm4, xmmword ptr [rdi + 4*rdx + 16]
movaps  xmm5, xmm1
addps   xmm5, xmm3
cmpltps xmm3, xmm2
movaps  xmm6, xmm0
addps   xmm6, xmm4
cmpltps xmm4, xmm2
andps   xmm5, xmm3
andnps  xmm3, xmm1
movaps  xmm1, xmm3
orps    xmm1, xmm5
andps   xmm6, xmm4
andnps  xmm4, xmm0
movaps  xmm0, xmm4
orps    xmm0, xmm6
add     rdx, 8
cmp     rcx, rdx
jne     LBB0_3
```

The next thing I wanted to see was how well it dealt with branches.
So in our routine we're making a small change here where we're only summing up floats under some certain value.
And what do you know, it actually comes back as SIMD code.
We get an unrolling here as well. It might not be exactly what I'd write by
hand, but hey, we didn't actually have to write anything.
If you leave the talk now, you're making a mistake.
And the reason is that..

2 days before gold..

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;
  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
    else
      acc -= f;
  }
  return acc;
}
```

..if you bank on this and hope the compiler will always SIMD-ify your loops, you're making a fatal mistake.
This is what will happen two days before gold.
Someone will touch some code in some innocent way, here adding some else condition, and watch what happens with the code generation.
All of a sudden we fall out of this nice SIMD performance space and into scalar code with branches left and right. These branches can't be predicted correctly and this loop runs at a fraction of the performance, maybe 10% or less.

2 days before gold..

```
float Foo(const float input[ ], int n)
{
  float acc = 0.f;
  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
    else
      acc -= f;
  }
  return acc;
}
```

```
        movss   xmm2, dword ptr [rdi]
        ucomiss xmm1, xmm2
        jbe     LBB0_4

        addss   xmm0, xmm2
        jmp     LBB0_5

LBB0_4: subss   xmm0, xmm2

LBB0_5: add     rdi, 4
        dec     esi
        jne     loop
```

✗ SCALAR

..if you bank on this and hope the compiler will always SIMD-ify your loops, you're making a fatal mistake.
This is what will happen two days before gold.
Someone will touch some code in some innocent way, here adding some else condition, and watch what happens with the code generation.
All of a sudden we fall out of this nice SIMD performance space and into scalar code with branches left and right. These branches can't be predicted correctly and this loop runs at a fraction of the performance, maybe 10% or less.

Compiler Auto-vectorization

- Breaks (silently) during maintenance
- Black box programming model
- Zero portability
- Compilers are tools, not magic wands

And this is the MAJOR problem with compiler auto-vectorization.
It silently breaks during regular maintenance of the code.
Someone might touch a header file somewhere or change an inlined piece of code,
which could mean any number of loops fall out of SIMD and become scalar,
branchy loops. This means we'll have new performance bugs to track down all
over the place.
And it also leads to this black box programming model.
Because once we've identified that one of our key loops is no longer being
SIMD-ified, what are we going to do to fix it? Remember, all we have is
scalar C or C++. So it leads to a lot of experimentation and trial and error
because we're so far away from the result we're trying to get to.
To make things even worse, all of this is completely compiler-dependent.
So if we're trying to port our stuff to multiple platforms, there's just no way we
can rely on a bunch of compilers from different vendors to agree on what
can be vectorized.
There are no real surprises here, compilers are not magic, they're pretty simple tools.
What they are good at is performing the same mediocre optimizations over and over again.
So by all means we should have this feature enabled, but it should be
considered a small bonus if it works.

ISPC

- Shader-like language for SSE/AVX
- Main benefit: automatic SSE/AVX switching
- Requires investment in more abstraction levels

To actually get a guarantee about SIMD code generation we might reach for some specialized tool. One such tool is ISPC, and this is a tool where we write shader-like source code, but with additional SSE-specific information like "gangs". Basically it's a tool that KNOWS it's targeting SSE and AVX.

The main benefit here is that this tool can take your code and compile it for a range of different CPU architectures. This means you can have different object files generated for example for SSE2 and AVX.

This is nice in certain cases, for example we use Intel's BCT texture compressor which is written in ISPC. When we run it on our newer workstations it's almost twice as fast because it automatically switches to AVX instructions. So that's cool.

But we still have this problem of another level of abstraction. We're not actually programming the hardware, we're programming ISPC. And then when something doesn't work, we're left having to figure out if it's because we've not mapped our problem to ISPC correctly, or if ISPC is not generating the code we want - or it has a bug.

My recommendation for ISPC is this: if you're heavily invested in SIMD in your tools code, or you are a PC-only shop targeting a lot of different hardware specs then it might make sense.

At Insomniac it doesn't get much use as we're primarily a console developer.

Intrinsics

- Preferred way to write SIMD at Insomniac Games

So what do we use? We use intrinsic functions. All our SIMD code is currently written this way.

We like this because it's predictable. When we use an intrinsic, we're basically selecting the instruction to use. It's a very thin wrapper around the actual SIMD instructions the CPU has to offer, which means there are very few surprises when we program this way.

It's also good because it's the solution that offers the best debugging environment for us.

And it's more convenient to write and maintain than assembly.

To use intrinsics effectively we have to be familiar with how the CPU works. We need to know what its good at doing, and what its bad at doing.

I don't see this as an argument against using intrinsics, because to be effective in this space we have to know something about the hardware.

To quote Mike Acton who is the engine director at Insomniac Games:

All good programming is hard, and bad programming is easy

Intrinsics

- Preferred way to write SIMD at Insomniac Games
- Predictable—no invisible performance regressions

So what do we use? We use intrinsic functions. All our SIMD code is currently written this way.

We like this because it's predictable. When we use an intrinsic, we're basically selecting the instruction to use. It's a very thin wrapper around the actual SIMD instructions the CPU has to offer, which means there are very few surprises when we program this way.

It's also good because it's the solution that offers the best debugging environment for us.

And it's more convenient to write and maintain than assembly.

To use intrinsics effectively we have to be familiar with how the CPU works. We need to know what its good at doing, and what its bad at doing.

I don't see this as an argument against using intrinsics, because to be effective in this space we have to know something about the hardware.

To quote Mike Acton who is the engine director at Insomniac Games:

All good programming is hard, and bad programming is easy

Intrinsics

- Preferred way to write SIMD at Insomniac Games
- Predictable—no invisible performance regressions
- You'll have to become familiar with the CPU

So what do we use? We use intrinsic functions. All our SIMD code is currently written this way.

We like this because it's predictable. When we use an intrinsic, we're basically selecting the instruction to use. It's a very thin wrapper around the actual SIMD instructions the CPU has to offer, which means there are very few surprises when we program this way.

It's also good because it's the solution that offers the best debugging environment for us.

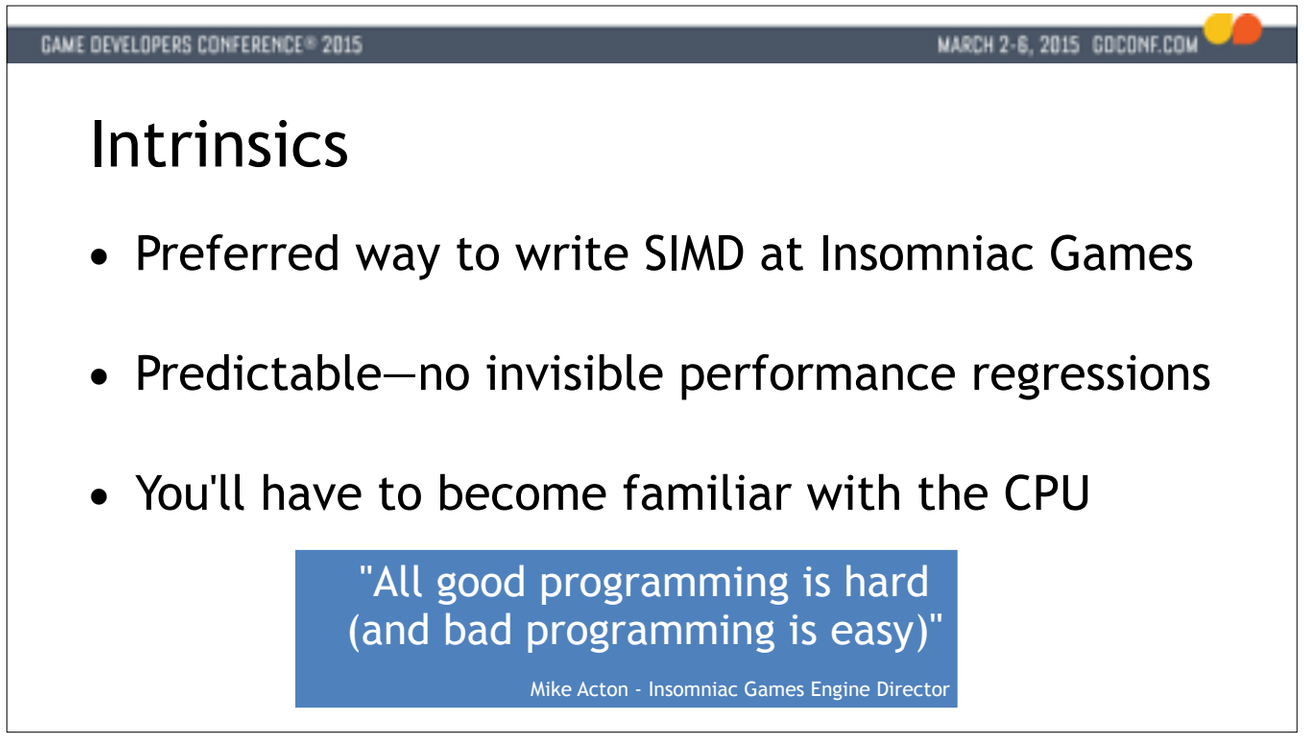
And it's more convenient to write and maintain than assembly.

To use intrinsics effectively we have to be familiar with how the CPU works. We need to know what its good at doing, and what its bad at doing.

I don't see this as an argument against using intrinsics, because to be effective in this space we have to know something about the hardware.

To quote Mike Acton who is the engine director at Insomniac Games:

All good programming is hard, and bad programming is easy



So what do we use? We use intrinsic functions. All our SIMD code is currently written this way.

We like this because it's predictable. When we use an intrinsic, we're basically selecting the instruction to use. It's a very thin wrapper around the actual SIMD instructions the CPU has to offer, which means there are very few surprises when we program this way.

It's also good because it's the solution that offers the best debugging environment for us.

And it's more convenient to write and maintain than assembly.

To use intrinsics effectively we have to be familiar with how the CPU works. We need to know what its good at doing, and what its bad at doing.

I don't see this as an argument against using intrinsics, because to be effective in this space we have to know something about the hardware.

To quote Mike Acton who is the engine director at Insomniac Games:

All good programming is hard, and bad programming is easy

Assembly

- Always an option!

Finally, if we're not happy with C++ we might drop down to assembly. That's always an option, and I love that option myself.

However it's become harder to do conveniently, especially because Visual Studio doesn't support inline assembly in their 64-bit compiler. So instead we have to invest in some external assembler like nasm or yasm and then try to figure out how to get debugging working. That's a bit of a hassle.

But even if we move past that hurdle there are a lot of things that can go wrong with 64-bit assembly. If you're a beginner starting out, it's really hard to get going these days.

Especially differences in register usage and stack layout between windows and every other OS is troublesome and leads to a lot of hoops to jump through if you want to do portable assembly.

So we stick to intrinsics.

Assembly

- Always an option!
- No inline assembly on 64-bit VS compilers 
  - Need external assembler (e.g. yasm)

Finally, if we're not happy with C++ we might drop down to assembly. That's always an option, and I love that option myself.

However it's become harder to do conveniently, especially because Visual Studio doesn't support inline assembly in their 64-bit compiler. So instead we have to invest in some external assembler like nasm or yasm and then try to figure out how to get debugging working. That's a bit of a hassle.

But even if we move past that hurdle there are a lot of things that can go wrong with 64-bit assembly. If you're a beginner starting out, it's really hard to get going these days.

Especially differences in register usage and stack layout between windows and every other OS is troublesome and leads to a lot of hoops to jump through if you want to do portable assembly.

So we stick to intrinsics.

Assembly

- Always an option!
- No inline assembly on 64-bit VS compilers 
  - Need external assembler (e.g. yasm)
- *Numerous* pitfalls for the beginner
  - Tricky to maintain ABI portability between OSs
  - Non-volatile registers
  - x64 Windows exception handling
  - Stack alignment, debugging, …

Finally, if we're not happy with C++ we might drop down to assembly. That's always an option, and I love that option myself.

However it's become harder to do conveniently, especially because Visual Studio doesn't support inline assembly in their 64-bit compiler. So instead we have to invest in some external assembler like nasm or yasm and then try to figure out how to get debugging working. That's a bit of a hassle.

But even if we move past that hurdle there are a lot of things that can go wrong with 64-bit assembly. If you're a beginner starting out, it's really hard to get going these days.

Especially differences in register usage and stack layout between windows and every other OS is troublesome and leads to a lot of hoops to jump through if you want to do portable assembly.

So we stick to intrinsics.

So it's 2015..

Technology	Year Introduced
SSE	1999
SSE2	2001
SSE3	2004
SSSE3	2006
SSE4.1	2007
SSE4.2	2008
AVX	2011
AVX2	2013

Anyway, it's 2015, and SSE is really old by now.
The first version, which wasn't awesome, came out in 1999.
That's ancient.
SSE2, which is still a perfectly viable target came out in 2001.

Why isn’t SSE used more?

TARGET 13:00
So you'd think everyone is using SSE effectively in a lot of places, but in my experience we're not doing that as a dev community. There's a LOT of scalar code out there. What's up with that?

Well I hear these three arguments a lot when I talk to other programmers about this problem:

"We don't know what the CPU is going to be, so we can't use it"

As we just saw, SSE has been around for a long time. If you're a PC developer, your minspec probably is more recent than a 2001 era machine. If you have a 64-bit build, you WILL have SSE2 instructions. And if you're a console developer, you have all the latest features, guaranteed. So this argument is just completely bogus.

The other argument I hear a lot is "we can't use SSE because of the way we lay out our data". We'll look at this one in more detail later, but here's a spoiler: It's probably because your data is laid out incorrectly.

And my favorite one is: "we tried it and it didn't help". Let's look at that one first.

Why isn’t SSE used more?

- “We don't know what the CPU is going to be”

TARGET 13:00
So you'd think everyone is using SSE effectively in a lot of places, but in my experience we're not doing that as a dev community. There's a LOT of scalar code out there. What's up with that?

Well I hear these three arguments a lot when I talk to other programmers about this problem:

"We don't know what the CPU is going to be, so we can't use it"

As we just saw, SSE has been around for a long time. If you're a PC developer, your minspec probably is more recent than a 2001 era machine. If you have a 64-bit build, you WILL have SSE2 instructions. And if you're a console developer, you have all the latest features, guaranteed. So this argument is just completely bogus.

The other argument I hear a lot is "we can't use SSE because of the way we lay out our data". We'll look at this one in more detail later, but here's a spoiler: It's probably because your data is laid out incorrectly.

And my favorite one is: "we tried it and it didn't help". Let's look at that one first.

Why isn’t SSE used more?

- “We don't know what the CPU is going to be”
- “It doesn’t fit our data layout”

TARGET 13:00
So you'd think everyone is using SSE effectively in a lot of places, but in my experience we're not doing that as a dev community. There's a LOT of scalar code out there. What's up with that?

Well I hear these three arguments a lot when I talk to other programmers about this problem:

"We don't know what the CPU is going to be, so we can't use it"

As we just saw, SSE has been around for a long time. If you're a PC developer, your minspec probably is more recent than a 2001 era machine. If you have a 64-bit build, you WILL have SSE2 instructions. And if you're a console developer, you have all the latest features, guaranteed. So this argument is just completely bogus.

The other argument I hear a lot is "we can't use SSE because of the way we lay out our data". We'll look at this one in more detail later, but here's a spoiler: It's probably because your data is laid out incorrectly.

And my favorite one is: "we tried it and it didn't help". Let's look at that one first.

Why isn’t SSE used more?

- “We don't know what the CPU is going to be”
- “It doesn’t fit our data layout”
- “We’ve tried it and it didn’t help”

TARGET 13:00
So you'd think everyone is using SSE effectively in a lot of places, but in my experience we're not doing that as a dev community. There's a LOT of scalar code out there. What's up with that?

Well I hear these three arguments a lot when I talk to other programmers about this problem:

"We don't know what the CPU is going to be, so we can't use it"

As we just saw, SSE has been around for a long time. If you're a PC developer, your minspec probably is more recent than a 2001 era machine. If you have a 64-bit build, you WILL have SSE2 instructions. And if you're a console developer, you have all the latest features, guaranteed. So this argument is just completely bogus.

The other argument I hear a lot is "we can't use SSE because of the way we lay out our data". We'll look at this one in more detail later, but here's a spoiler: It's probably because your data is laid out incorrectly.

And my favorite one is: "we tried it and it didn't help". Let's look at that one first.

“We’ve tried it and it didn’t help”

- Common translation: “We wrote class Vec4…”

```
class Vec4 {
  __m128 data;

  operator+ (…)
  operator- (…)
};
```

So whenever I hear this, my ears automatically translate this to
"We wrote this class Vec4 and that didn't help".
I've been guilty of this myself and I think we all have when we're first exposed to intrinsics and SIMD. The first impulse is to wrap up the SIMD data type, m128, in a class, and then provide all these operators and functions and whatever.

class Vec4

__m128 has X/Y/Z/W.

So clearly it’s a 4D vector.

And it's easy to see why we do this. As programmers we're good at putting labels on things.
So we take one look at this thing and we see it has X, Y, Z and W and we think "Clearly it's a 4D vector" - and we wrap it in a class.

class Vec4, later that day

Addition and multiply is going great!

This will be really fast!

And we think, YEAH, this is actually going to be great because addition, multiply and all these simple things fall out to single instructions. It's going so well!

class Vec4, that night

Oh no!

Dot products and other common operations are really awkward and slow!

But later that night, we hit a snag in that dot products and other essential stuff we might want to do with our vec4 class turns out really awkward.
In fact it's almost no faster than scalar code!
And we can't really see why some people are so excited about SSE.

The Awkward Vec4 Dot Product

```
??? Vec4Dot(Vec4 a, Vec4 b)
{
  __m128 a0  = _mm_mul_ps(a.data, b.data);

  // Wait, how are we going to add the products together?


  return ???; // And what do we return?
}
```

If you haven't done this exercise, let me show you really quickly how this goes
down when we try to make a dot product function for this class.
We have this function, and it takes two Vec4 objects A and B, which are laid out
horizontally in two registers.
What we want to do is component-wise multiples and then add all the products up
to get our dot product.
Which means we're making a scalar, right?
So the first problem is what we return here.
Are we going to return a float?
That feels weird because we're crossing over from the SIMD domain back into the scalar domain.
So a common thing to do is to return a vector with the same dot product repeated four times.
The other reason this is awkward is that SSE doesn't have a good horizontal add instruction.
So there are a number of red flags here..

The Awkward Vec4 Dot Product

```
Vec4 Vec4Dot(Vec4 a, Vec4 b)
{
  __m128 a0  = _mm_mul_ps(a.data, b.data);
  __m128 a1  = _mm_shuffle_ps(a0, a0, _MM_SHUFFLE(2, 3, 0, 1));
  __m128 a2  = _mm_add_ps(a1, a0);
  __m128 a3  = _mm_shuffle_ps(a2, a2, _MM_SHUFFLE(0, 1, 3, 2));
  __m128 dot = _mm_add_ps(a3, a2);
  return dot; // WAT: the same dot product in all four lanes
}
```

So we end up with something like this.
We do our multiples with one instruction, but then we have a huge infrastructure cost to pay
where we have to shuffle things around and add repeatedly to sum the products up horizontally
and we're wasting a lot of the hardware.
So it works out to be 5 operations - one mul, two shuffles and two adds - to compute one dot product.
This isn't a lot faster than scalar code which just needs 7.

class Vec4, the next morning

Well, at least we've tried it.

I guess SSE sucks.

And we look a this and we say: "well, I guess SSE sucks and I can keep using scalar code"

class Vec4

- Wrong conclusion: SSE sucks because Vec4 is slow

But that's the wrong conclusion. SSE doesn't suck.

The right conclusion is that this whole Vec4 idea sucks.

class Vec4

- Wrong conclusion: SSE sucks because Vec4 is slow
- Correct conclusion: The whole Vec4 idea sucks

But that's the wrong conclusion. SSE doesn't suck.

The right conclusion is that this whole Vec4 idea sucks.

Better SSE Dot Products (plural)

```
__m128 dx   = _mm_mul_ps(ax, bx);   // dx   = ax * bx
__m128 dy   = _mm_mul_ps(ay, by);   // dy   = ay * by
__m128 dz   = _mm_mul_ps(az, bz);   // dz   = az * bz
__m128 dw   = _mm_mul_ps(aw, bw);   // dw   = aw * bw

__m128 a0   = _mm_add_ps(dx, dy);   // a0   = dx + dy
__m128 a1   = _mm_add_ps(dz, dw);   // a1   = dz + dw
__m128 dots = _mm_add_ps(a0, a1);   // dots = a0 + a1
```

And today I'm convinced that this is what we want to see when we're computing dot products.
In this code we're doing 4 multiples and 3 adds.
If we write our code like this, we're doing 7 operations to compute 4 independent dot products.
This is a LOT better than spending 5 operations to compute just one dot product.
Now this is working WITH the grain of the hardware, we're doing what the hardware designers intended for us to do.
** And now we're actually back on that nice kitchen chopping board, getting the most benefit we can from each instruction we execute.

Better SSE Dot Products (plural)

```
__m128 dx
__m128 dy
__m128 dz
__m128 dw

__m128 a0
__m128 a1
__m128 dots
```

```
dx   = ax * bx
dy   = ay * by
dz   = az * bz
dw   = aw * bw

a0   = dx + dy
a1   = dz + dw
dots = a0 + a1
```

And today I'm convinced that this is what we want to see when we're computing dot products.
In this code we're doing 4 multiples and 3 adds.
If we write our code like this, we're doing 7 operations to compute 4 independent dot products.
This is a LOT better than spending 5 operations to compute just one dot product.
Now this is working WITH the grain of the hardware, we're doing what the hardware designers intended for us to do.
** And now we're actually back on that nice kitchen chopping board, getting the most benefit we can from each instruction we execute.

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow

So it's easy to get out of this, just don't spend time on trying to wrap up SSE functionality. Because what we're trying to do is to take a hardware feature that's really engineered to work on separate streams of data and we're trying to abstract it using the C++ class mechanism which does the complete opposite. So we're doomed to run slowly and waste a lot of time trying to do this.

** SSE really wants to be this thing you can just reach for and pull out anywhere in your code. Any wrappers of frameworks are just going to get in the way and slow us down.

** There's still the need for abstraction, of course. We can't solve every problem with first principles. What we do instead is to write helper functions as needed and we can share and reuse these as appropriate. For example we have a large collision detection library written using basic functions as utilities.

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow
- SSE code wants to be free!
  - Best performance without wrappers or frameworks

So it's easy to get out of this, just don't spend time on trying to wrap up SSE functionality. Because what we're trying to do is to take a hardware feature that's really engineered to work on separate streams of data and we're trying to abstract it using the C++ class mechanism which does the complete opposite. So we're doomed to run slowly and waste a lot of time trying to do this.

** SSE really wants to be this thing you can just reach for and pull out anywhere in your code. Any wrappers of frameworks are just going to get in the way and slow us down.

** There's still the need for abstraction, of course. We can't solve every problem with first principles. What we do instead is to write helper functions as needed and we can share and reuse these as appropriate. For example we have a large collision detection library written using basic functions as utilities.

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow
- SSE code wants to be free!
  - Best performance without wrappers or frameworks
- Just write small helper routines as needed

So it's easy to get out of this, just don't spend time on trying to wrap up SSE functionality. Because what we're trying to do is to take a hardware feature that's really engineered to work on separate streams of data and we're trying to abstract it using the C++ class mechanism which does the complete opposite. So we're doomed to run slowly and waste a lot of time trying to do this.

** SSE really wants to be this thing you can just reach for and pull out anywhere in your code. Any wrappers of frameworks are just going to get in the way and slow us down.

** There's still the need for abstraction, of course. We can't solve every problem with first principles. What we do instead is to write helper functions as needed and we can share and reuse these as appropriate. For example we have a large collision detection library written using basic functions as utilities.

“It doesn’t fit our data layout”

- void SpawnParticle(float pos[3], …);
  - Stored in struct Particle { float pos[3]; … }
  - Awkward to work with array of Particle in SSE

The other argument we talked about is "we can't use SSE because of the way we organize our data".
How this happens is also pretty easy to see.
Let's consider some hypothetical particle system.
We have some high level spawn function, that takes a bunch of parameters.
It's pretty likely this function will just tend to forward its parameters straight into some struct.
And yeah, if you have an array like that in memory, it's going to be really clumsy to try to do SIMD.

** So don't do that!
We can keep our nice high-level spawn function and just change the way we organize the data in memory.
The problem is with our structures, not with SSE.
No one is forcing us to keep our APIs in sync with the internal data layout we choose.

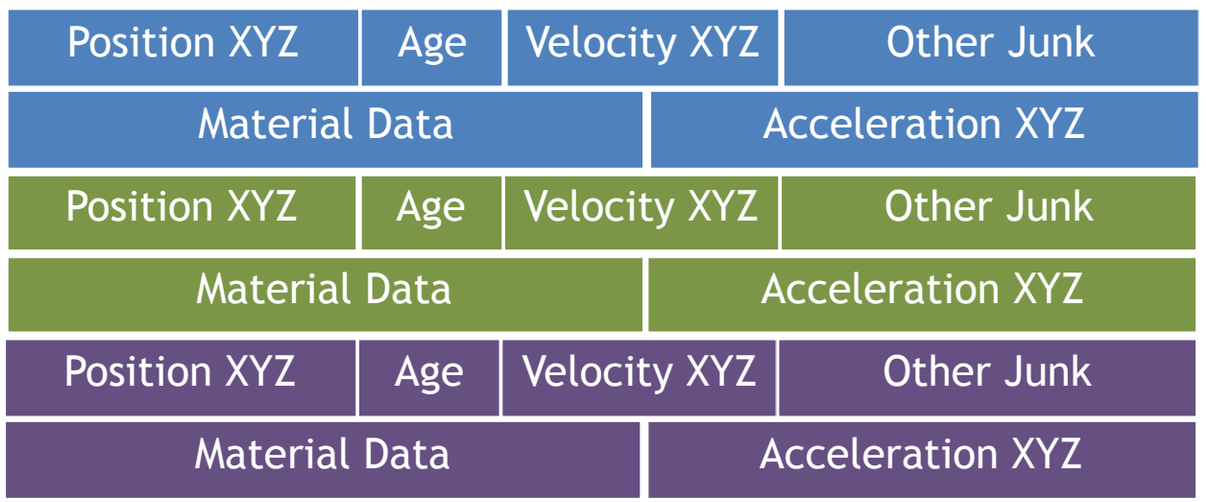
“It doesn’t fit our data layout”

- void SpawnParticle(float pos[3], …);
  - Stored in struct Particle { float pos[3]; … }
  - Awkward to work with array of Particle in SSE
- So don’t do that
  - Keep the spawn function, change the memory layout
  - The problem is with struct Particle, not SSE
  - Separate data layout from interfaces entirely

The other argument we talked about is "we can't use SSE because of the way we organize our data".
How this happens is also pretty easy to see.
Let's consider some hypothetical particle system.
We have some high level spawn function, that takes a bunch of parameters.
It's pretty likely this function will just tend to forward its parameters straight into some struct.
And yeah, if you have an array like that in memory, it's going to be really clumsy to try to do SIMD.

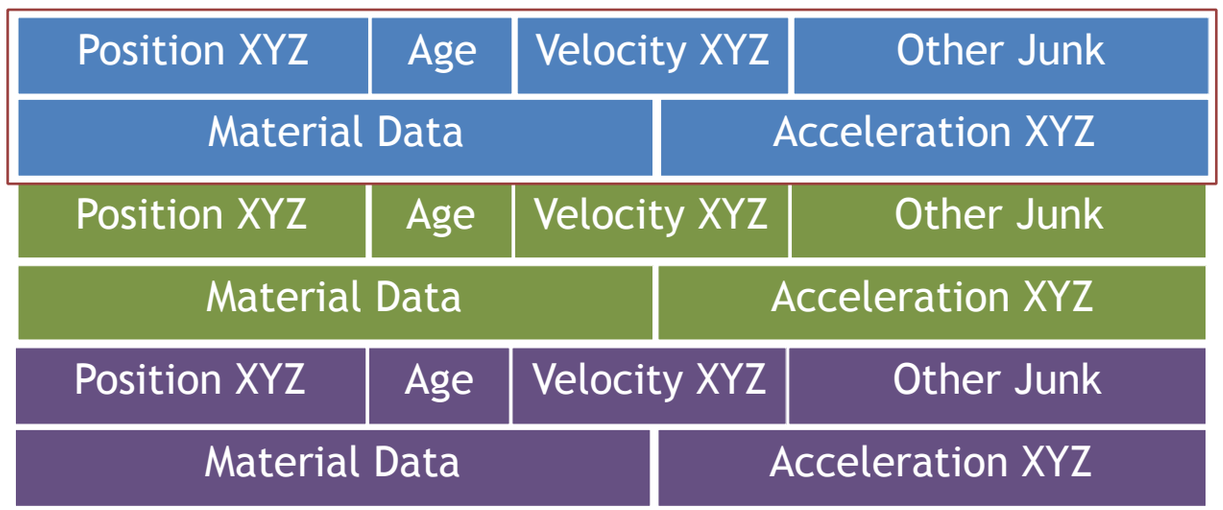
** So don't do that!
We can keep our nice high-level spawn function and just change the way we organize the data in memory.
The problem is with our structures, not with SSE.
No one is forcing us to keep our APIs in sync with the internal data layout we choose.

Struct Particle in memory (AOS)



So if we look at this example in more detail we might have something like this going on in memory.
Memory here is going from left to right, top to bottom.
Each block of memory represents one particle, right.
** This is convenient because it allows us to think about a particle as a thing that occupies some
contiguous range of memory, and we can talk about the identity of a particle using a pointer.
Every language gives you this by default, right. C, C++, Pascal, you name it.
This is what we're born into, and this is what we're comfortable with. It's like comfort food for us.
But for SSE it's really awkward, because it's a hardware feature geared towards SOA data.
If we want to get four AGE parameters here to do some SIMD we're going to have to go to four places in memory.
I don't even have space for four age fields on the slide!
This is of course inefficient, and a lot of times this is where the argument comes from.

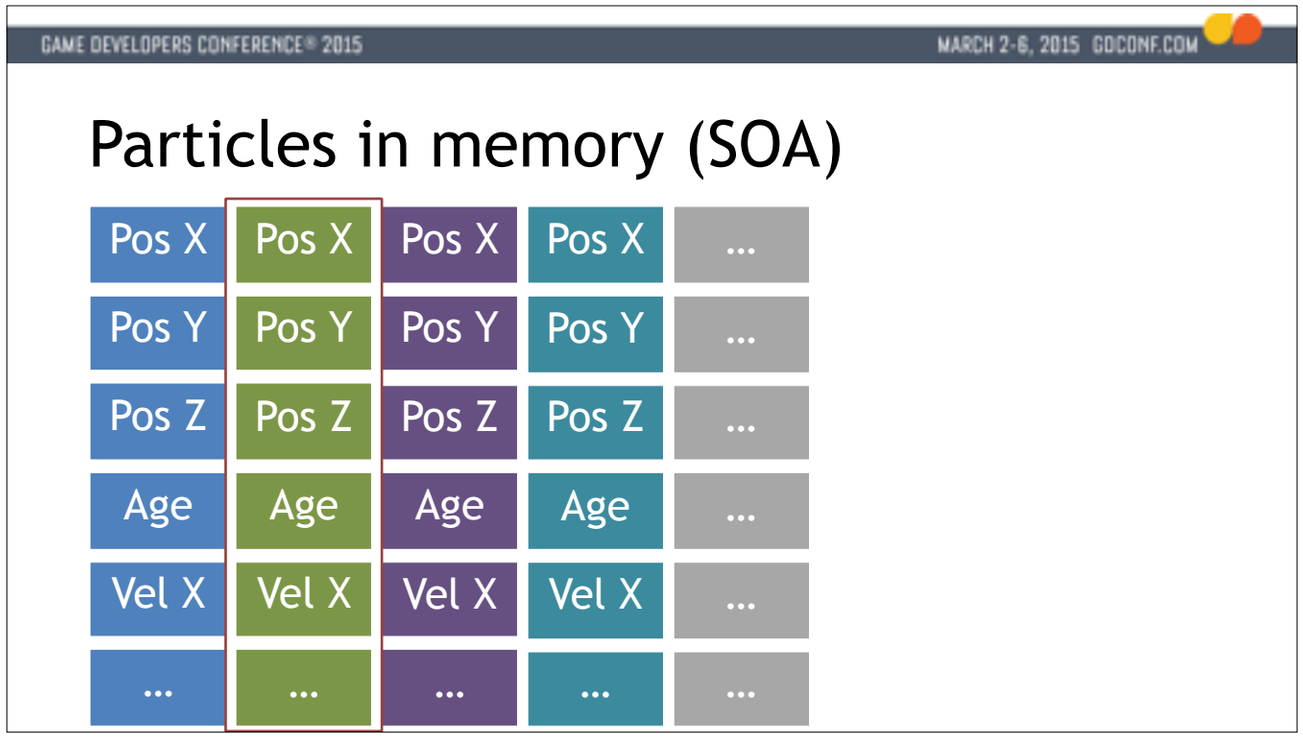
Struct Particle in memory (AOS)



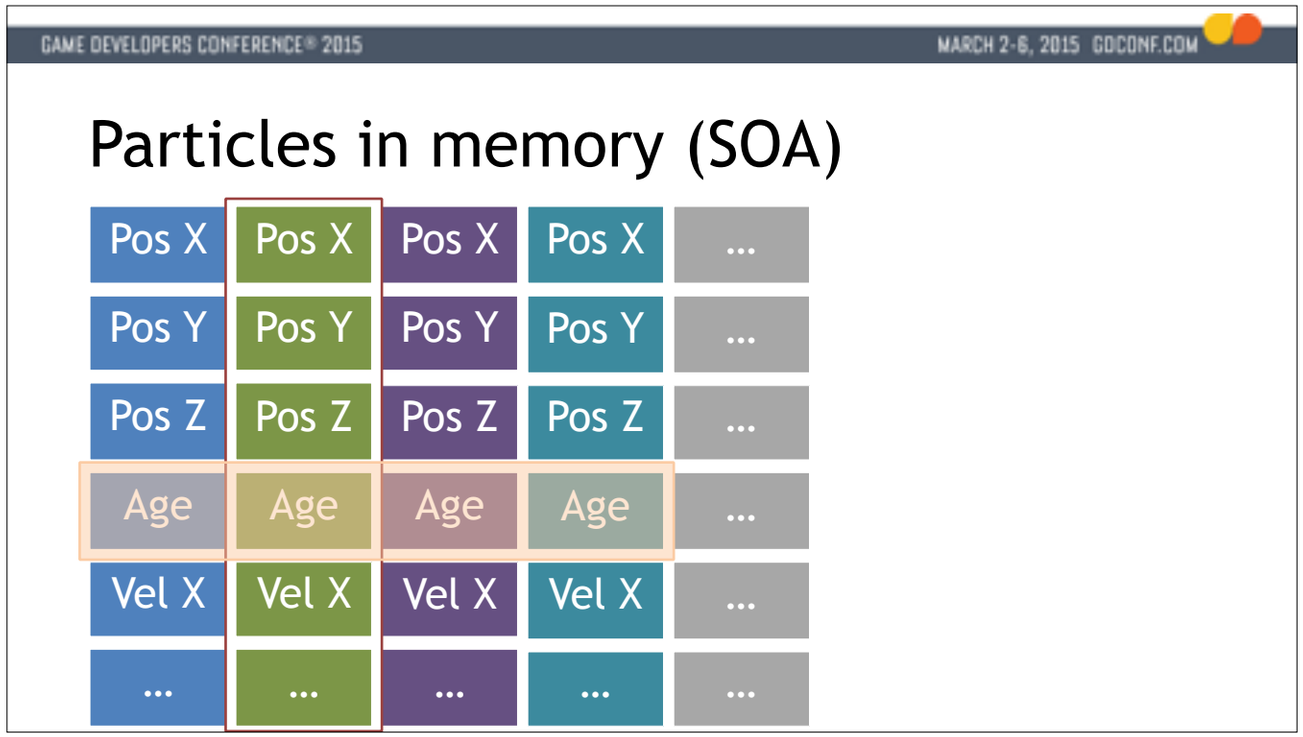
So if we look at this example in more detail we might have something like this going on in memory.
Memory here is going from left to right, top to bottom.
Each block of memory represents one particle, right.
** This is convenient because it allows us to think about a particle as a thing that occupies some
contiguous range of memory, and we can talk about the identity of a particle using a pointer.
Every language gives you this by default, right. C, C++, Pascal, you name it.
This is what we're born into, and this is what we're comfortable with. It's like comfort food for us.
But for SSE it's really awkward, because it's a hardware feature geared towards SOA data.
If we want to get four AGE parameters here to do some SIMD we're going to have to go to four places in memory.
I don't even have space for four age fields on the slide!
This is of course inefficient, and a lot of times this is where the argument comes from.

Particles in memory (SOA)

But if we're willing to change our data layout, we can move things around to SOA form.
We have a bunch of parallel arrays, one for Xs, one for Ys, one for Age etc.
This is a little uncomfortable, because our particle struct is now split into
a number of little pieces and we can't have that nice pointer anymore.
** To reconstruct the particle we have to go looking in multiple places in memory.
The good news is that we can use an index as our canonical identity instead of
a pointer, which is good anyway because pointers are big and indices are small.
** But the great news is that we can now reach into memory and load 4 or 8 or so
attributes all at once with a single read from memory.
** When we do this, we're doing what the hardware designers intended and we're
working with the grain of the hardware.



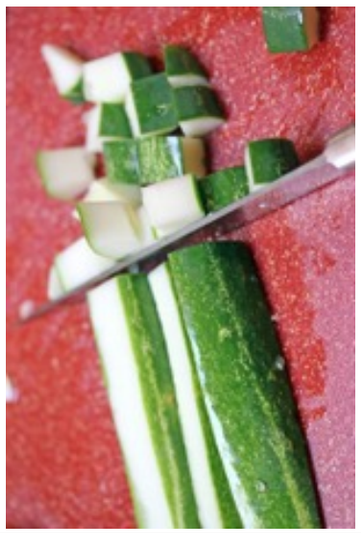
But if we're willing to change our data layout, we can move things around to SOA form.
We have a bunch of parallel arrays, one for Xs, one for Ys, one for Age etc.
This is a little uncomfortable, because our particle struct is now split into
a number of little pieces and we can't have that nice pointer anymore.
** To reconstruct the particle we have to go looking in multiple places in memory.
The good news is that we can use an index as our canonical identity instead of
a pointer, which is good anyway because pointers are big and indices are small.
** But the great news is that we can now reach into memory and load 4 or 8 or so
attributes all at once with a single read from memory.
** When we do this, we're doing what the hardware designers intended and we're
working with the grain of the hardware.



But if we're willing to change our data layout, we can move things around to SOA form.
We have a bunch of parallel arrays, one for Xs, one for Ys, one for Age etc.
This is a little uncomfortable, because our particle struct is now split into
a number of little pieces and we can't have that nice pointer anymore.
** To reconstruct the particle we have to go looking in multiple places in memory.
The good news is that we can use an index as our canonical identity instead of
a pointer, which is good anyway because pointers are big and indices are small.
** But the great news is that we can now reach into memory and load 4 or 8 or so
attributes all at once with a single read from memory.
** When we do this, we're doing what the hardware designers intended and we're
working with the grain of the hardware.

Particles in memory (SOA)

Pos X	Pos X	Pos X	Pos X	…
Pos Y	Pos Y	Pos Y	Pos Y	…
Pos Z	Pos Z	Pos Z	Pos Z	…
Age	Age	Age	Age	…
Vel X	Vel X	Vel X	Vel X	…
…	…	…	…	…



But if we're willing to change our data layout, we can move things around to SOA form.
We have a bunch of parallel arrays, one for Xs, one for Ys, one for Age etc.
This is a little uncomfortable, because our particle struct is now split into
a number of little pieces and we can't have that nice pointer anymore.
** To reconstruct the particle we have to go looking in multiple places in memory.
The good news is that we can use an index as our canonical identity instead of
a pointer, which is good anyway because pointers are big and indices are small.
** But the great news is that we can now reach into memory and load 4 or 8 or so
attributes all at once with a single read from memory.
** When we do this, we're doing what the hardware designers intended and we're
working with the grain of the hardware.

Data Layout Recap

- SOA form usually *much* better for SSE code

So to recap this really quickly.

SOA form should be your go-to default for SSE code, because SSE instructions really work well with it. Especially if you're targeting SSE2. Definitely start here.

** That doesn't mean AOS form is completely out. Sometimes we have index lookups and that sort of thing where it's beneficial to get at a group of things with a single cache miss. We're then going to have a shuffle cost to move things around.

** And that type of shuffling is also useful sometimes when you're working with some fixed data layout you can't change, like data coming from some third party library. Then we might generate the SOA data we need locally in our transform. But it shouldn't be the default.

Data layout is THE most important part of SIMD programming. If the data layout is poor, then no amount of SIMD is going to help.

Data Layout Recap

- SOA form usually *much* better for SSE code
- AOS form can be beneficial in some cases

So to recap this really quickly.

SOA form should be your go-to default for SSE code, because SSE instructions really work well with it. Especially if you're targeting SSE2. Definitely start here.

** That doesn't mean AOS form is completely out. Sometimes we have index lookups and that sort of thing where it's beneficial to get at a group of things with a single cache miss. We're then going to have a shuffle cost to move things around.

** And that type of shuffling is also useful sometimes when you're working with some fixed data layout you can't change, like data coming from some third party library. Then we might generate the SOA data we need locally in our transform. But it shouldn't be the default.

Data layout is THE most important part of SIMD programming. If the data layout is poor, then no amount of SIMD is going to help.

Data Layout Recap

- SOA form usually *much* better for SSE code
- AOS form can be beneficial in some cases
- Generate SOA data locally in transform if needed

So to recap this really quickly.

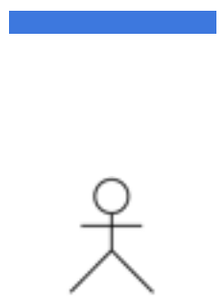
SOA form should be your go-to default for SSE code, because SSE instructions really work well with it. Especially if you're targeting SSE2. Definitely start here.

** That doesn't mean AOS form is completely out. Sometimes we have index lookups and that sort of thing where it's beneficial to get at a group of things with a single cache miss. We're then going to have a shuffle cost to move things around.

** And that type of shuffling is also useful sometimes when you're working with some fixed data layout you can't change, like data coming from some third party library. Then we might generate the SOA data we need locally in our transform. But it shouldn't be the default.

Data layout is THE most important part of SIMD programming. If the data layout is poor, then no amount of SIMD is going to help.

Doors



So with that in mind, we're going to move into our first example.
We're going to look at a gameplay problem. I've chosen a gameplay problem
just to show that SIMD has a wider applicability than you might think.

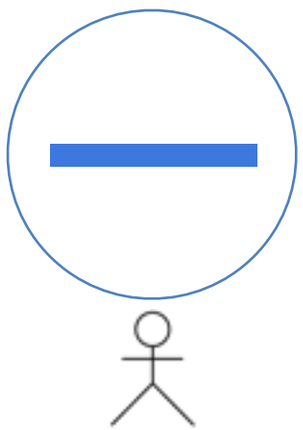
So we have these doors, and they're sort of like Star Trek doors that animate open.
We have a door here, seen from above and a character.

** The door has a radius around it where it will open if the character is of the right team.

** And when the character moves into the radius, the door opens automatically.

The problem we're going to talk about is how you deal with this when you have a bunch
of doors and a bunch of characters.

Doors



So with that in mind, we're going to move into our first example.
We're going to look at a gameplay problem. I've chosen a gameplay problem
just to show that SIMD has a wider applicability than you might think.

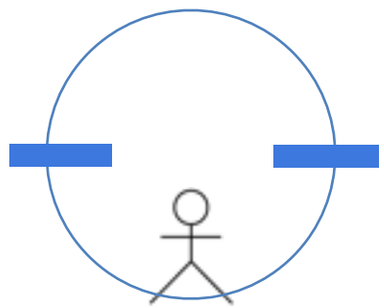
So we have these doors, and they're sort of like Star Trek doors that animate open.
We have a door here, seen from above and a character.

** The door has a radius around it where it will open if the character is of the right team.

** And when the character moves into the radius, the door opens automatically.

The problem we're going to talk about is how you deal with this when you have a bunch
of doors and a bunch of characters.

Doors



So with that in mind, we're going to move into our first example.
We're going to look at a gameplay problem. I've chosen a gameplay problem
just to show that SIMD has a wider applicability than you might think.

So we have these doors, and they're sort of like Star Trek doors that animate open.
We have a door here, seen from above and a character.

** The door has a radius around it where it will open if the character is of the right team.

** And when the character moves into the radius, the door opens automatically.

The problem we're going to talk about is how you deal with this when you have a bunch
of doors and a bunch of characters.

Door Update

- Typical many-to-many game problem

TARGET: 24:00
It's a typical many-to-many problem which is common in gameplay code.

** In a test I set up I had 100 doors, and 30 characters. It's a lot, but you
could have something like that in a big level with a lot of players.
Because all the doors have to test against all the characters, we're looking at 3000 tests.

Door Update

- Typical many-to-many game problem
- 100 doors, 30 characters - 3,000 tests

TARGET: 24:00
It's a typical many-to-many problem which is common in gameplay code.

** In a test I set up I had 100 doors, and 30 characters. It's a lot, but you
could have something like that in a big level with a lot of players.
Because all the doors have to test against all the characters, we're looking at 3000 tests.

Original Door Update

```
void Door::Update(float dt)
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Original Door Update

Scalar by definition

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Original Door Update

Scalar by definition

Repeated work

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Original Door Update

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Scalar by definition

Repeated work

Multiple L2 misses

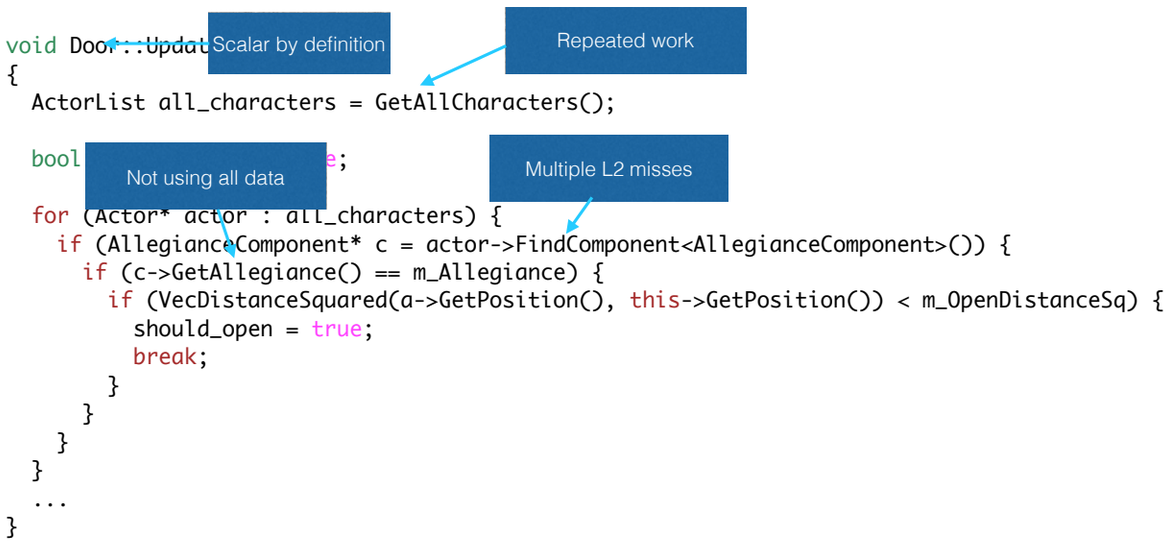
If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Original Door Update



If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Original Door Update

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool                                    e;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Scalar by definition

Repeated work

Not using all data

Multiple L2 misses

Scalar compute

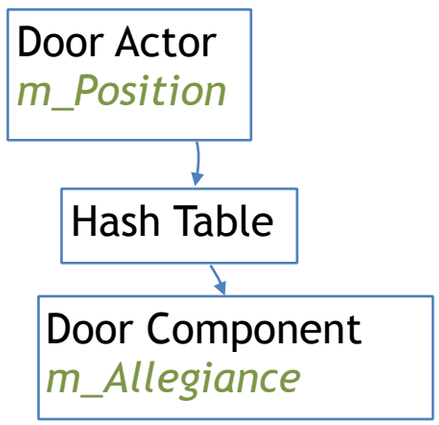
If we look at the original update we had for this - which I think was some prototype code to start with - the details aren't super important, so you don't have to read all this.

But what it was doing, was to update each door individually, classic OO style.
The first thing it would do is to go grab a list of all the characters in the world.
Then for each character, it would go and see if it could find out what allegiance the character has.
If you're a non-native English speaker like me, allegiance basically means team in this context.
Once we've found the allegiance of this character, we check to see if it's the right allegiance
that is allowed to open this door.
If it is, it would then check the position of this guy to see if he is inside the open radius.

So what are the problems?
Well if you have one door and one character, it's probably fine. Hard to improve on.

** However, for this talk we're trying to do more than one - SIMD - and Door is singular - scalar - so that not going to fly.
** And every door would go get this list of all the characters in the world, which is repeated expensive work.
** For each of those characters it would then take multiple L2 misses to traverse a bunch of data structures, just to try to figure out what team this guy is on.
** And once it got that data, it would only use a single byte of it. That's a lot of waste.
** And then finally here's the cutest part of this I think. In the middle of this sea of L2 misses, thousands of cycles of memory access penalties, it's trying really hard to avoid a square root, which is something like 15 cycles. Just to see if the guy was close enough.

Input Data in Original Update



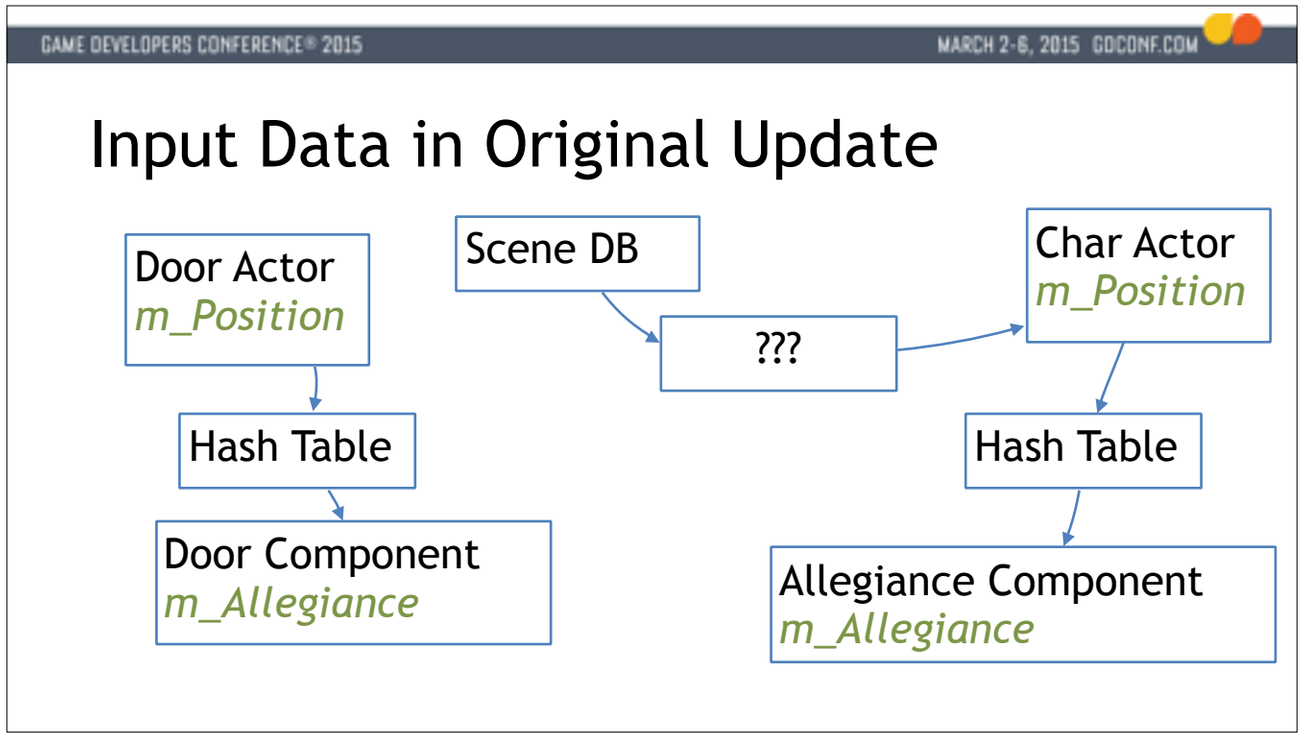
So to understand this better we have to look at the data.
In our engine we have two concepts that all gameplay code is built on.
We have ACTORS which are basically empty shells with a position in the world.
Actors can contain a bunch of COMPONENTS which is where all the gameplay code goes.
These things are associated with a hash table, because it's highly dynamic.
In this case we're actually talking about the update function for this door component.

To find all the characters in the world, the door component update would go into
the scene database and traverse all these structures to find a list of characters, which are also actors.
Then it would walk their hash table structures to find these allegiance components which tells it what
team they're on.

So I don't know about you but when I see this I think of a ball of twine.

** And apparently this is a thing here in the united states, you can go to Cawker City and it has this world's largest ball of twine.. I just thought that was relevant here.

** OK, so we're traversing all of this, what do we get out of it?
Well we need these positions, which are three floats, so 12 bytes.
We also need the teams, and those are just one byte.
That's a lot of pointer traversal to find not a lot of data.



So to understand this better we have to look at the data.
In our engine we have two concepts that all gameplay code is built on.
We have ACTORS which are basically empty shells with a position in the world.
Actors can contain a bunch of COMPONENTS which is where all the gameplay code goes.
These things are associated with a hash table, because it's highly dynamic.
In this case we're actually talking about the update function for this door component.

To find all the characters in the world, the door component update would go into
the scene database and traverse all these structures to find a list of characters, which are also actors.
Then it would walk their hash table structures to find these allegiance components which tells it what
team they're on.

So I don't know about you but when I see this I think of a ball of twine.

** And apparently this is a thing here in the united states, you can go to Cawker City and it has this world's largest ball of twine.. I just thought that was relevant here.

** OK, so we're traversing all of this, what do we get out of it?
Well we need these positions, which are three floats, so 12 bytes.
We also need the teams, and those are just one byte.
That's a lot of pointer traversal to find not a lot of data.

Input Data in Original Update

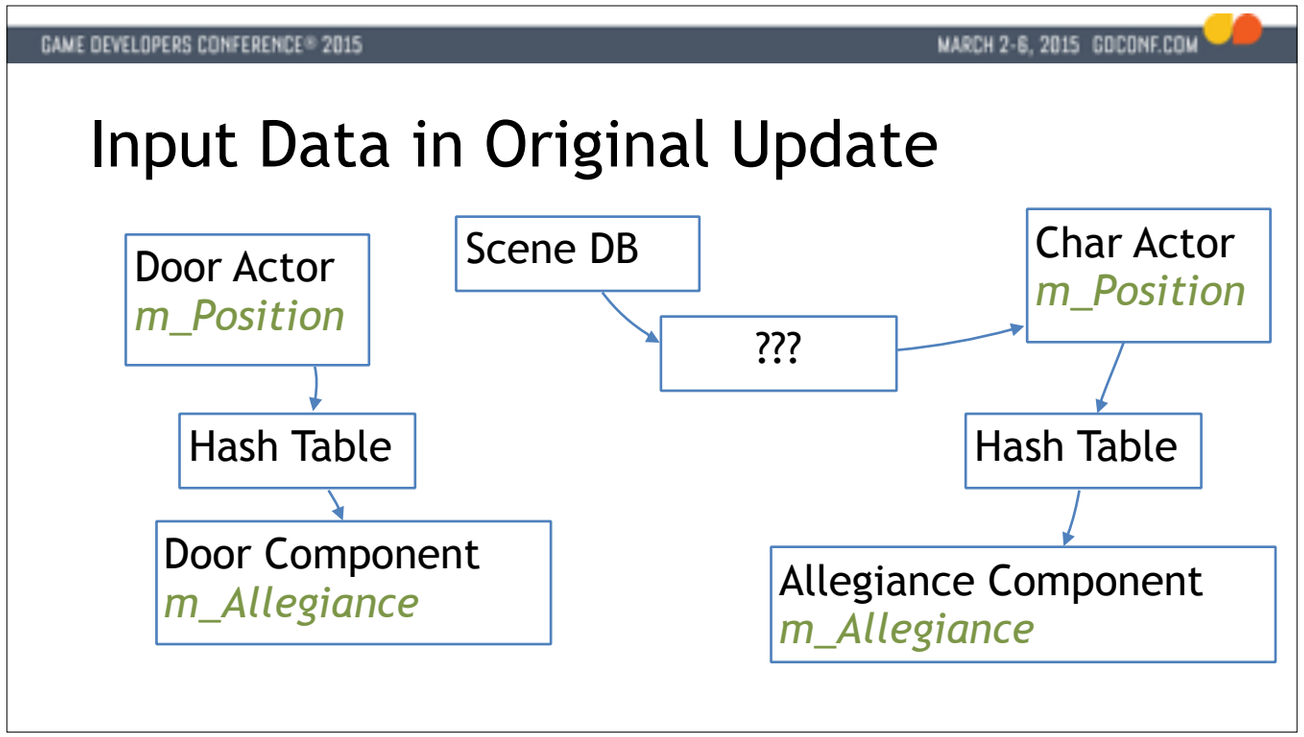
So to understand this better we have to look at the data.
In our engine we have two concepts that all gameplay code is built on.
We have ACTORS which are basically empty shells with a position in the world.
Actors can contain a bunch of COMPONENTS which is where all the gameplay code goes.
These things are associated with a hash table, because it's highly dynamic.
In this case we're actually talking about the update function for this door component.

To find all the characters in the world, the door component update would go into
the scene database and traverse all these structures to find a list of characters, which are also actors.
Then it would walk their hash table structures to find these allegiance components which tells it what
team they're on.

So I don't know about you but when I see this I think of a ball of twine.

** And apparently this is a thing here in the united states, you can go to Cawker City and it has this world's largest ball of twine.. I just thought that was relevant here.

** OK, so we're traversing all of this, what do we get out of it?
Well we need these positions, which are three floats, so 12 bytes.
We also need the teams, and those are just one byte.
That's a lot of pointer traversal to find not a lot of data.



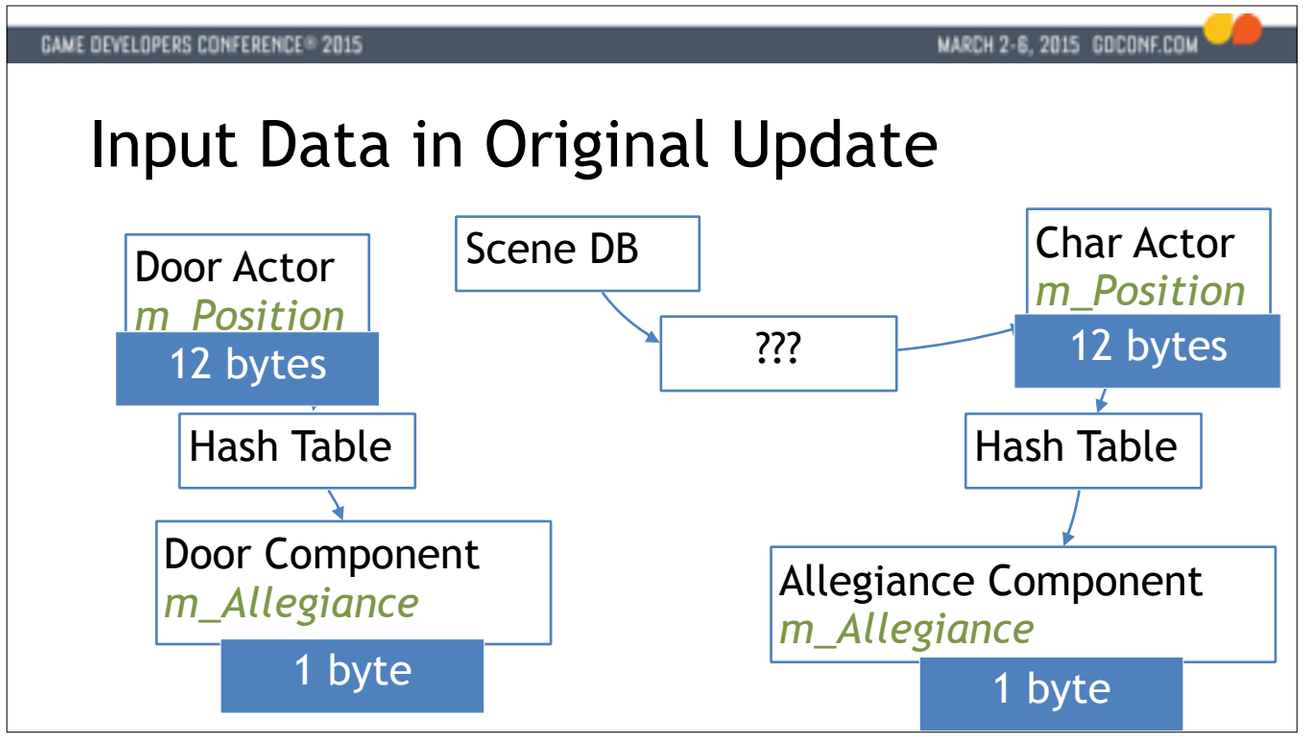
So to understand this better we have to look at the data.
In our engine we have two concepts that all gameplay code is built on.
We have ACTORS which are basically empty shells with a position in the world.
Actors can contain a bunch of COMPONENTS which is where all the gameplay code goes.
These things are associated with a hash table, because it's highly dynamic.
In this case we're actually talking about the update function for this door component.

To find all the characters in the world, the door component update would go into
the scene database and traverse all these structures to find a list of characters, which are also actors.
Then it would walk their hash table structures to find these allegiance components which tells it what
team they're on.

So I don't know about you but when I see this I think of a ball of twine.

** And apparently this is a thing here in the united states, you can go to Cawker City and it has this world's largest ball of twine.. I just thought that was relevant here.

** OK, so we're traversing all of this, what do we get out of it?
Well we need these positions, which are three floats, so 12 bytes.
We also need the teams, and those are just one byte.
That's a lot of pointer traversal to find not a lot of data.



So to understand this better we have to look at the data.
In our engine we have two concepts that all gameplay code is built on.
We have ACTORS which are basically empty shells with a position in the world.
Actors can contain a bunch of COMPONENTS which is where all the gameplay code goes.
These things are associated with a hash table, because it's highly dynamic.
In this case we're actually talking about the update function for this door component.

To find all the characters in the world, the door component update would go into
the scene database and traverse all these structures to find a list of characters, which are also actors.
Then it would walk their hash table structures to find these allegiance components which tells it what
team they're on.

So I don't know about you but when I see this I think of a ball of twine.

** And apparently this is a thing here in the united states, you can go to Cawker City and it has this world's largest ball of twine.. I just thought that was relevant here.

** OK, so we're traversing all of this, what do we get out of it?
Well we need these positions, which are three floats, so 12 bytes.
We also need the teams, and those are just one byte.
That's a lot of pointer traversal to find not a lot of data.

What is it actually computing?

```
for each door:
  door.should_be_open = 0
  for each character:
    if InRadius(…) and door.team == char.team:
      door.should_be_open = 1
```

Now that we know a little bit more, let's look at the code again to make sure we really understand it.

What does the radius test compute?

$$(x_0-x_1)^2 + (y_0-y_1)^2 + (z_0-z_1)^2 <= r^2$$

The only thing we haven't looked at is this radius test.
And what it does, is to avoid a square root by computing the squared length of a 3D vector and comparing that with a squared radius.

SIMD Prep Work

- Move door data to central SOA table

Really just a bag of values in SOA form
Each door has an index into central data stash
Once per update, not 100 times
Stash in simple array on stack (alloca for variable size - or use scratch allocator)

We start off by gathering our door data in a central place so we can reason about it.
We'll set up a SOA data block of all door X Y Z coordinates and their allegiances, as well as their "open radiuses"
We can access and update this data by keeping an index into this stash from each door instance, for scalar code that needs to interface with our new design.
The other thing we'll do is to chase through the pointers once, and store the results on the stack.
We're now in a position to update all the doors together.

SIMD Prep Work

- Move door data to central SOA table
- Build actor table once, locally in update

Really just a bag of values in SOA form
Each door has an index into central data stash
Once per update, not 100 times
Stash in simple array on stack (alloca for variable size - or use scratch allocator)

We start off by gathering our door data in a central place so we can reason about it.
We'll set up a SOA data block of all door X Y Z coordinates and their allegiances, as well as their "open radiuses"
We can access and update this data by keeping an index into this stash from each door instance, for scalar code that needs to interface with our new design.
The other thing we'll do is to chase through the pointers once, and store the results on the stack.
We're now in a position to update all the doors together.

Door Update Data Design

```
// In memory, SOA
struct DoorData {
  uint32_t  Count;
  float    *X;
  float    *Y;
  float    *Z;
  float    *RadiusSq;
  uint32_t *Allegiance;
  // Output data
  uint32_t *ShouldBeOpen;
} s_Doors;
```

```
// On the stack, AOS
struct CharData {
  float    X;
  float    Y;
  float    Z;
  uint32_t Allegiance;
} c[MAXCHARS];
```

This then is the data we'll need for our update.
The s_Doors struct is persistent and lives in memory. We have a count of doors, and a bunch of parallel arrays for all the different attributes.
We also store a parallel array of computed values - whether the doors should be open or not.
In reality this feeds in to later stages, but I have only so much space on this slide.
To the right you have the character data, the other part of the problem. We'll build this once, on the stack.

SIMD Door Update

- New update does all doors in one go
  - Test 4 doors vs 1 actor in inner loop

This update is now going to write itself basically, but we're going to do things 4 vs 1 using SIMD.

We'll loop over all actors and broadcast their data one at a time into another set of SIMD registers
When we're through all actors we're left with a status bit for each door telling us whether it should be open or not.
Let's look at some code for this.

SIMD Door Update

- New update does all doors in one go
  - Test 4 doors vs 1 actor in inner loop
- Massive benefits from the data layout
  - All compute naturally falls out as SIMD operations

This update is now going to write itself basically, but we're going to do things 4 vs 1 using SIMD.

We'll loop over all actors and broadcast their data one at a time into another set of SIMD registers
When we're through all actors we're left with a status bit for each door telling us whether it should be open or not.
Let's look at some code for this.

Outer Loop Prologue

```
for (int d = 0; d < door_count; d += 4) {

  __m128  door_x  = _mm_load_ps(&s_Doors.X[d]);
  __m128  door_y  = _mm_load_ps(&s_Doors.Y[d]);
  __m128  door_z  = _mm_load_ps(&s_Doors.Z[d]);
  __m128  door_r2 = _mm_load_ps(&s_Doors.RadiusSq[d]);
  __m128i door_a  = _mm_load_si128((__m128i*) &s_Doors.Allegiance[d]);

  __m128i state   = _mm_setzero_si128();
```

Load attributes for 4 doors, clear 4 “open” accumulators

We’ll zip through these quickly, but I wanted to show you what the actual SIMD routine looks like.
We're going to be loading up data for 4 doors at a time. If we have say 3 doors we'll have to pad our arrays and memset them to zero, which is fine.
We load up a bunch of attributes, and also clear out a 4-wide accumulator that will keep track of whether these 4 doors should be open or not when we're done.

Inner Loop Prologue

```
...

for (int cc = 0; cc < char_count; ++cc) {
  __m128  char_x = _mm_broadcast_ss(&c[cc].x);
  __m128  char_y = _mm_broadcast_ss(&c[cc].y);
  __m128  char_z = _mm_broadcast_ss(&c[cc].z);
  __m128i char_a = _mm_set1_epi32(c[cc].allegiance);

  ...
```

Load attributes for 1 character, broadcast to all 4 lanes

Next is our inner loop. We're looping over all the characters, loading one attribute for X, Y, Z and team and then broadcasting that to each lane of a SIMD register, to line up with the 4 attributes we have for the doors.

Inner Loop Math

```
...
__m128  ddx    = _mm_sub_ps(door_x, char_x);
__m128  ddy    = _mm_sub_ps(door_y, char_y);
__m128  ddz    = _mm_sub_ps(door_z, char_z);
__m128  dtx    = _mm_mul_ps(ddx, ddx);
__m128  dty    = _mm_mul_ps(ddy, ddy);
__m128  dtz    = _mm_mul_ps(ddz, ddz);
__m128  dst_2  = _mm_add_ps(_mm_add_ps(dtx, dty), dtz);
...
```

Compute squared distance between character & the 4 doors

Now we're set up to run the test. We start by doing the math for the radius test. We compute the squared distance between this one actor and the 4 doors. It's the exact same number of operations as we saw before in the slide.

Inner Loop Epilogue

```
    ...
    __m128  rmask  = _mm_cmple_ps(dst_2, door_r2);
    __m128i amask  = _mm_cmpeq_epi32(char_a, door_a);
    __m128i mask   = _mm_and_si128(_mm_castps_si128(amask), rmask);

            state  = _mm_or_si128(mask, state);
}
...
```

Compare against door open radii AND allegiance => OR into state

Now we wrap up this group of doors vs this character by doing two tests:
- We check the squared radius against the door's squared radii
- We check the teams to see if they match

Both of these have to be pass to make the door want to open, so that's the AND you see in there.
Next we OR that into our accumulator. It doesn't matter if we have 1 or 100 guys in front of a door, it should still just open.

Outer Loop Epilogue

```
    ...

    _mm_store_si128((__m128i*) &s_Doors.ShouldBeOpen[d], state);

}
```

Store “should open” for these 4 doors, ready for next group of 4

We finish off the loop by storing the accumulated "should be open" state for the group of 4 doors.

```
for (int d = 0; d < door_count; d += 4) {
  __m128  door_x    = _mm_load_ps(&s_Doors.X[d]);
  __m128  door_y    = _mm_load_ps(&s_Doors.Y[d]);
  __m128  door_z    = _mm_load_ps(&s_Doors.Z[d]);
  __m128  door_r2   = _mm_load_ps(&s_Doors.RadiusSq[d]);
  __m128i door_a    = _mm_load_si128((__m128i*) &s_Doors.Allegiance[d]);
  __m128i state     = _mm_setzero_si128();

  for (int cc = 0; cc < char_count; ++cc) {
    __m128  char_x = _mm_broadcast_ss(&c[cc].x);
    __m128  char_y = _mm_broadcast_ss(&c[cc].y);
    __m128  char_z = _mm_broadcast_ss(&c[cc].z);
    __m128i char_a = _mm_set1_epi32(c[cc].allegiance);

    __m128  ddx    = _mm_sub_ps(door_x, char_x);
    __m128  ddy    = _mm_sub_ps(door_y, char_y);
    __m128  ddz    = _mm_sub_ps(door_z, char_z);
    __m128  dtx    = _mm_mul_ps(ddx, ddx);
    __m128  dty    = _mm_mul_ps(ddy, ddy);
    __m128  dtz    = _mm_mul_ps(ddz, ddz);
    __m128  dst_2  = _mm_add_ps(_mm_add_ps(dtx, dty), dtz);

    __m128  rmask  = _mm_cmple_ps(dst_2, door_r2);
    __m128i amask  = _mm_cmpeq_epi32(char_a, door_a);
    __m128i mask   = _mm_and_si128(_mm_castps_si128(amask), rmask);

            state  = _mm_or_si128(mask, state);
  }

  _mm_store_si128((__m128i*) &s_Doors.ShouldBeOpen[d], state);
}
```

Here it is all at once - it's not a lot of code.

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

When we check our work in the disassembly we see this inner loop. On a modern Intel chip this runs in 6 cycles per loop, and remember that each loop tests 4 actor/door pairs at once. So the cost is 1.5 cycles per test. This works out to about 4,500 cycles for the actual testing, and we've transformed a cache miss bound problem into a compute bound problem.

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

~6 cycles per loop

When we check our work in the disassembly we see this inner loop. On a modern Intel chip this runs in 6 cycles per loop, and remember that each loop tests 4 actor/door pairs at once. So the cost is 1.5 cycles per test. This works out to about 4,500 cycles for the actual testing, and we've transformed a cache miss bound problem into a compute bound problem.

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

~6 cycles per loop

4 door/actor tests per loop

When we check our work in the disassembly we see this inner loop. On a modern Intel chip this runs in 6 cycles per loop, and remember that each loop tests 4 actor/door pairs at once. So the cost is 1.5 cycles per test. This works out to about 4,500 cycles for the actual testing, and we've transformed a cache miss bound problem into a compute bound problem.

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

~6 cycles per loop

4 door/actor tests per loop

6/4 = 1.5 cycles per test

When we check our work in the disassembly we see this inner loop. On a modern Intel chip this runs in 6 cycles per loop, and remember that each loop tests 4 actor/door pairs at once. So the cost is 1.5 cycles per test. This works out to about 4,500 cycles for the actual testing, and we've transformed a cache miss bound problem into a compute bound problem.

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

~6 cycles per loop

4 door/actor tests per loop

6/4 = 1.5 cycles per test

100 doors * 30 characters ->

~4,500 cycles total

When we check our work in the disassembly we see this inner loop. On a modern Intel chip this runs in 6 cycles per loop, and remember that each loop tests 4 actor/door pairs at once. So the cost is 1.5 cycles per test. This works out to about 4,500 cycles for the actual testing, and we've transformed a cache miss bound problem into a compute bound problem.

Door Results

- 20-100x speedup

The result is a massive speedup, around 20-100x depending on the number of doors and actors involved.
And now that we have the data in a better arrangement for the problem at hand it's easy to find other improvements - we could for example sort the tables based on allegiance first and avoid half the work in each loop which would cut the time even more, but it's likely so fast now that it's not worth more attention. Until a designer gets an idea for a door game..

The point of this example was really to show that this type of brute force SIMD can optimize a thorny cache-missy problem and turn it into a compute bound problem. x86 CPUs are fantastic at burning through linear arrays, and by using that to our advantages we can have lots more objects in our games.

Typically these problems are relatively small blips, but there are many of them. Put together they can mean the difference between shipping in frame or shipping with framerate problems.

Door Results

- 20-100x speedup
- Brute force SIMD for “reasonable # of things”

The result is a massive speedup, around 20-100x depending on the number of doors and actors involved.
And now that we have the data in a better arrangement for the problem at hand it’s easy to find other improvements - we could for example sort the tables based on allegiance first and avoid half the work in each loop which would cut the time even more, but it’s likely so fast now that it’s not worth more attention. Until a designer gets an idea for a door game..

The point of this example was really to show that this type of brute force SIMD can optimize a thorny cache-missy problem and turn it into a compute bound problem. x86 CPUs are fantastic at burning through linear arrays, and by using that to our advantages we can have lots more objects in our games.

Typically these problems are relatively small blips, but there are many of them. Put together they can mean the difference between shipping in frame or shipping with framerate problems.

Door Results

- 20-100x speedup
- Brute force SIMD for “reasonable # of things”
- Solves “death by a thousand cuts” problems

The result is a massive speedup, around 20-100x depending on the number of doors and actors involved.
And now that we have the data in a better arrangement for the problem at hand it’s easy to find other improvements - we could for example sort the tables based on allegiance first and avoid half the work in each loop which would cut the time even more, but it’s likely so fast now that it’s not worth more attention. Until a designer gets an idea for a door game..

The point of this example was really to show that this type of brute force SIMD can optimize a thorny cache-missy problem and turn it into a compute bound problem. x86 CPUs are fantastic at burning through linear arrays, and by using that to our advantages we can have lots more objects in our games.

Typically these problems are relatively small blips, but there are many of them. Put together they can mean the difference between shipping in frame or shipping with framerate problems.

Techniques & Tricks

TARGET: 34:00
OK - so we've seen the power of linear data in action, but what if it's not that simple? That's when the tricks of the trade come into play. Any SIMD programmer worth his salt has a trick bag to reach into to work with problematic data.

We'll look at two such tricks here: left packing, and dynamic mask generation.

I'll first show how you can apply these tricks with more modern instructions — SSSE3 and later — and then we'll revisit them and see what madness we must descend into to make things work for SSE2.

Techniques & Tricks

- Need to cope with messy data & constraints

TARGET: 34:00
OK - so we've seen the power of linear data in action, but what if it's not that simple? That's when the tricks of the trade come into play. Any SIMD programmer worth his salt has a trick bag to reach into to work with problematic data.

We'll look at two such tricks here: left packing, and dynamic mask generation.

I'll first show how you can apply these tricks with more modern instructions — SSSE3 and later — and then we'll revisit them and see what madness we must descend into to make things work for SSE2.

Techniques & Tricks

- Need to cope with messy data & constraints
- We'll look at two tricks today

TARGET: 34:00
OK - so we've seen the power of linear data in action, but what if it's not that simple? That's when the tricks of the trade come into play. Any SIMD programmer worth his salt has a trick bag to reach into to work with problematic data.

We'll look at two such tricks here: left packing, and dynamic mask generation.

I'll first show how you can apply these tricks with more modern instructions — SSSE3 and later — and then we'll revisit them and see what madness we must descend into to make things work for SSE2.

Techniques & Tricks

- Need to cope with messy data & constraints
- We'll look at two tricks today
- First SSSE3+ (easier) then SSE2 (harder)

TARGET: 34:00
OK - so we've seen the power of linear data in action, but what if it's not that simple? That's when the tricks of the trade come into play. Any SIMD programmer worth his salt has a trick bag to reach into to work with problematic data.

We'll look at two such tricks here: left packing, and dynamic mask generation.

I'll first show how you can apply these tricks with more modern instructions — SSSE3 and later — and then we'll revisit them and see what madness we must descend into to make things work for SSE2.

Problem: Filtering Data

- Discarding data while streaming
  - Not a 1:1 relationship between input and output
  - N inputs, M outputs, M <= N
  - Not writing multiple of SIMD register width to output!
- Want to express as SIMD kernel, but how?

The first problem is that of data filtering. This comes up all the time in SIMD programming. Whether we're compacting a set of indices or filtering some normals we often want to discard some values and write a smaller number of elements to our output buffer.

It's not obvious how to do this, because reads and writes operate on 128-bit quantities. How can we write only some elements to our output buffer?

Scalar Filtering

```
int FilterFloats_Reference(const float input[], float output[],
                           int count, float limit)
{
  float *outputp = output;

  for (int i = 0; i < count; ++i) {
    if (input[i] >= limit)
      *outputp++ = input[i];
  }

  return (int) (outputp - output);
}
```

Let’s start by looking at a simplified scalar routine to do some filtering.
We’re looping over some array of floats, discarding values that compare lower than some threshold.
We return the number of elements written to the output buffer.
Seems straight forward. How to we go from here to a SIMD routine?

Scalar Filtering

```
int FilterFloats_Reference(const float input[], float output[],
                           int count, float limit)
{
  float *outputp = output;

  for (int i = 0; i < count; ++i) {
    if (input[i] >= limit)
      *outputp++ = input[i];
  }

  return (int) (outputp - output);
}
```

Let’s start by looking at a simplified scalar routine to do some filtering.
We’re looping over some array of floats, discarding values that compare lower than some threshold.
We return the number of elements written to the output buffer.
Seems straight forward. How to we go from here to a SIMD routine?

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

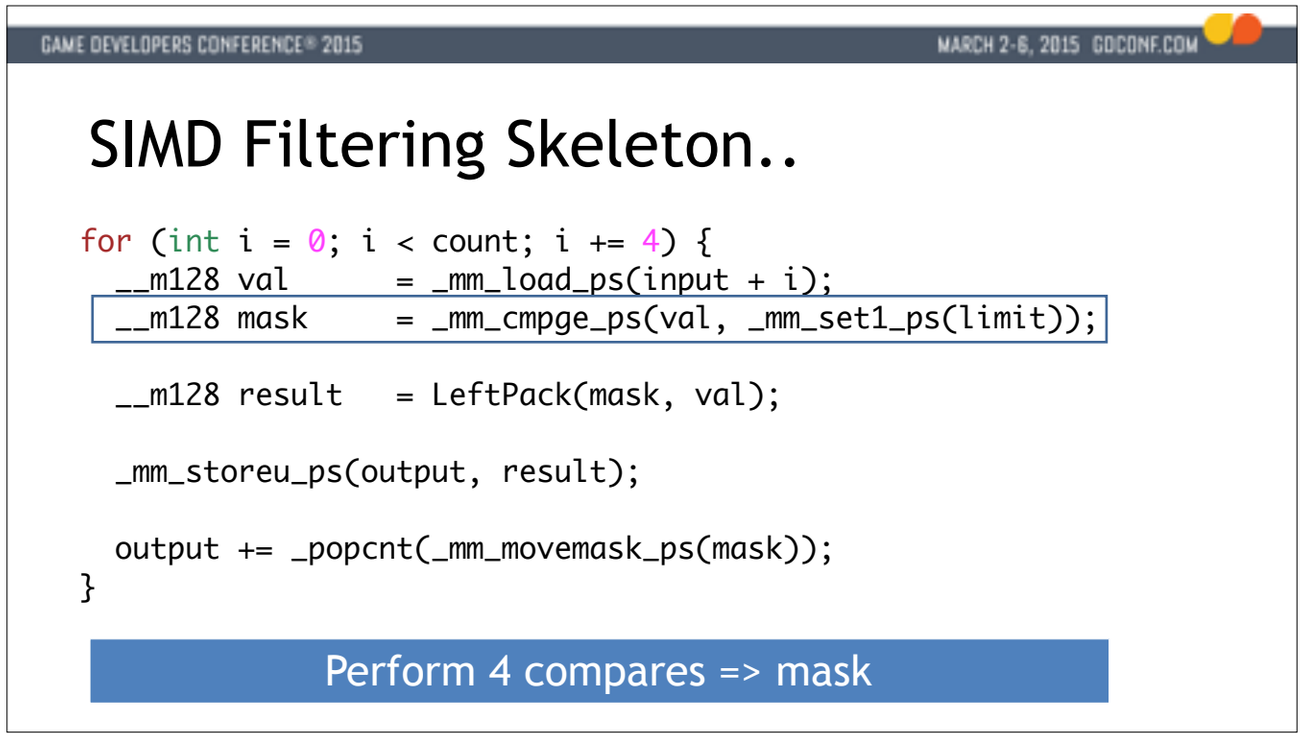
Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.

The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.



The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.

SIMD Filtering Skeleton..

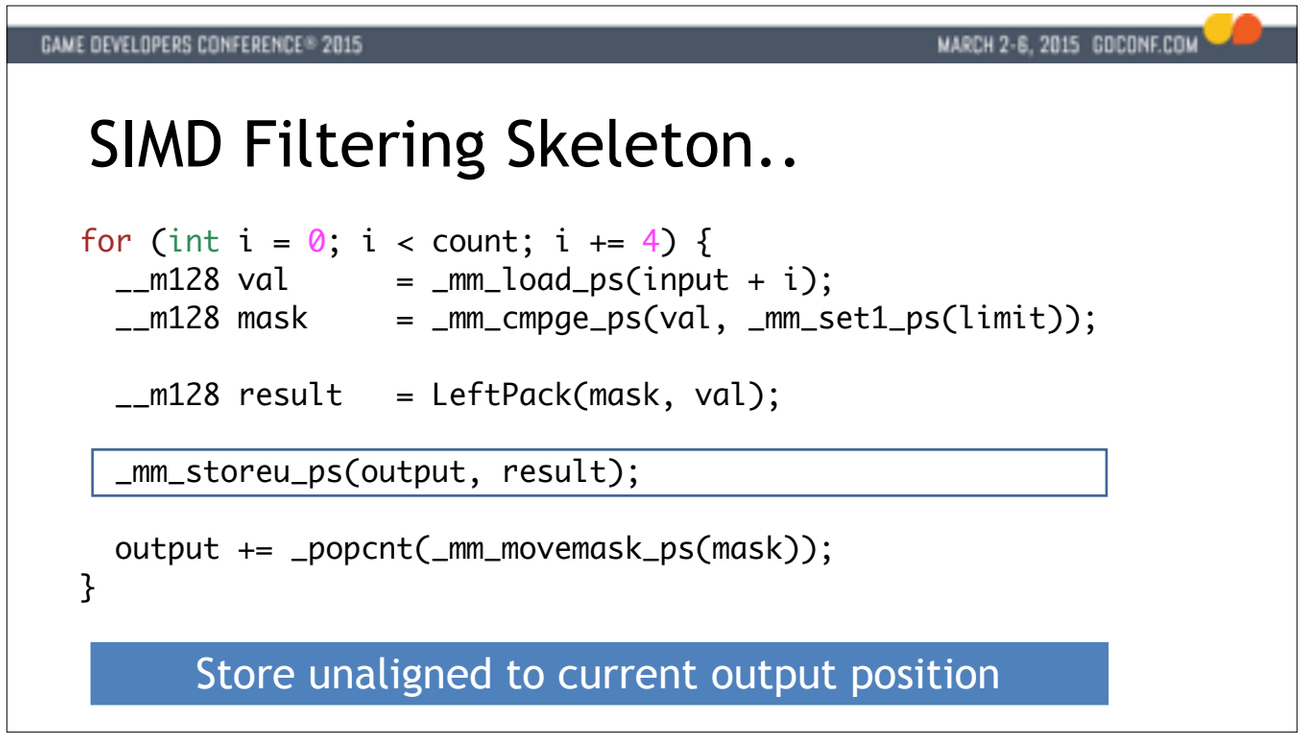
```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Left-pack valid elements to front of register

The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.



The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Advance output position based on mask

The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.

SIMD Filtering Skeleton..

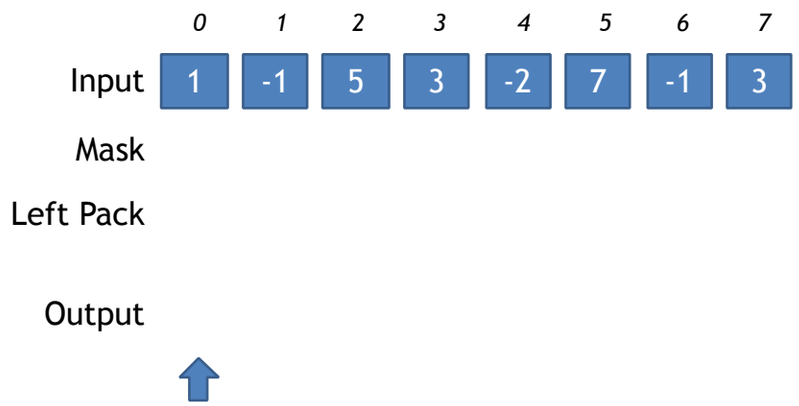
```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

The answer is by left packing our output data. The idea is to always load and store full SIMD registers, with the difference that input is always read at an even rate, output is going to be written in an unaligned fashion that arranges for only the valid elements to be retained in the output array.

Here's a skeleton of a SIMD loop to do filtering.
We start off by loading four floats into a SIMD register.
We can perform 4 comparisons simultaneously to generate a mask of which ones we want to keep.
Using the mask, we'll need to somehow move the elements we want to keep horizontally in the register to the left. There can be between 0 and 4 elements we want to keep!
We then store 4 elements using an unaligned store to the current output position, and advance the output pointer by the number of elements that were valid. Here I'm using popcnt() to figure out how many were valid.
Our valid elements will be packed to the left in the output register, and we carefully move the output pointer by an unaligned amount, so that the next write will store its valid elements right after it.

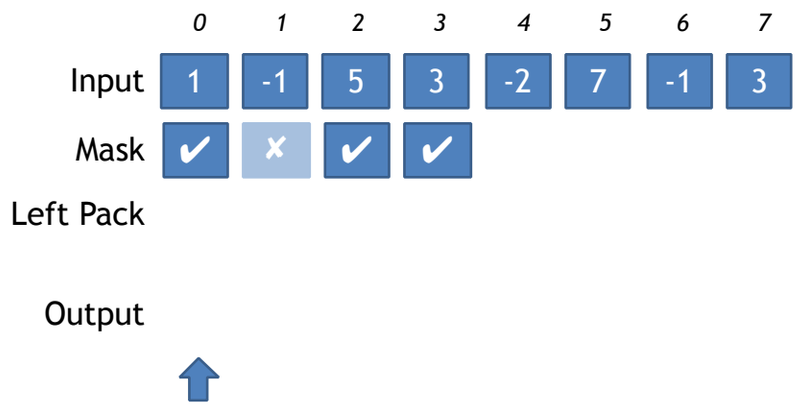
This may sound a little confusing, so let's look at a the SIMD filtering algorithm in detail before we get into the left packing.

Left Packing Problem (4-wide, limit=0)



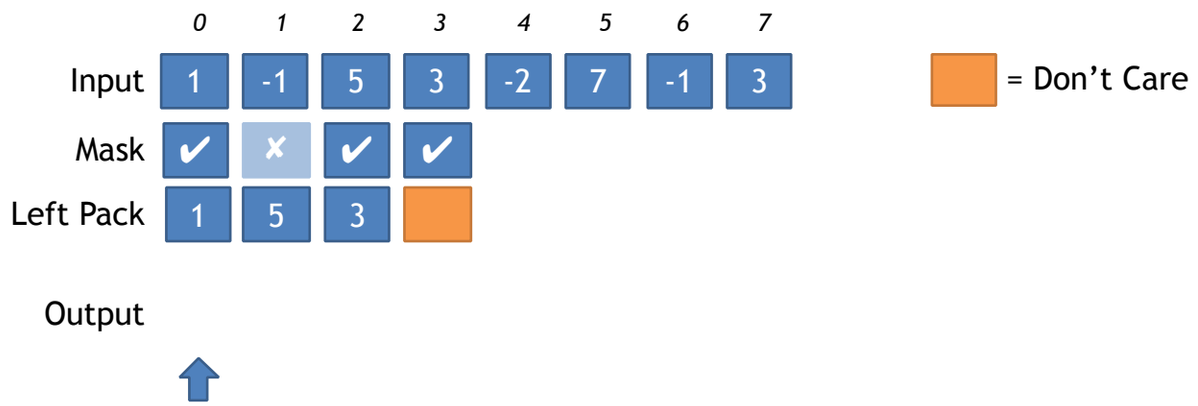
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



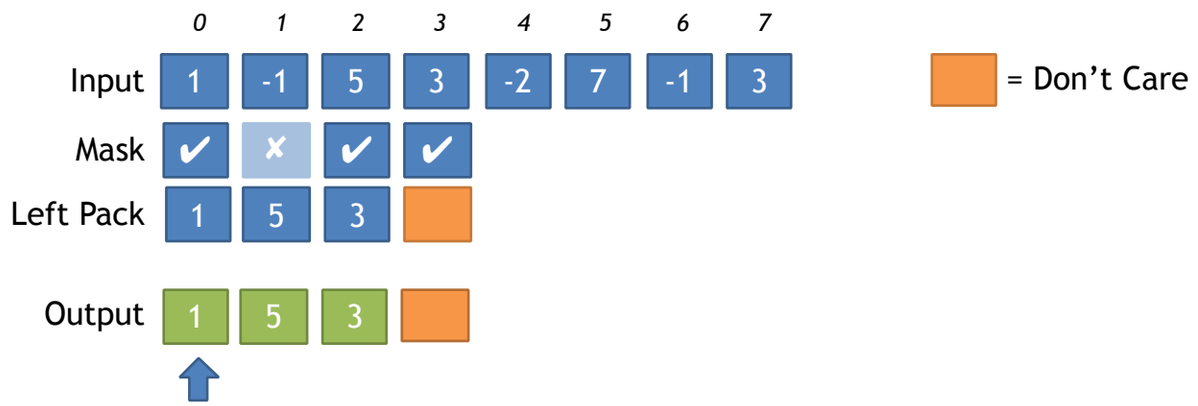
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



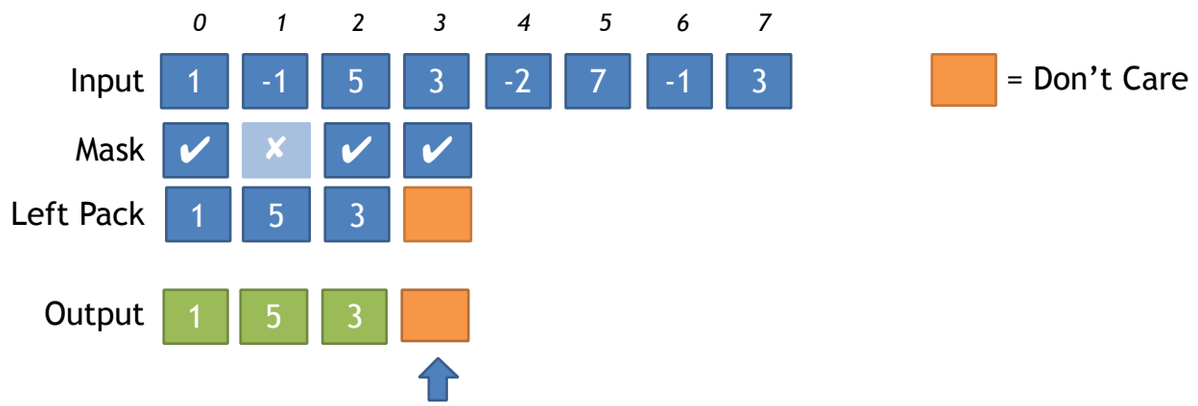
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



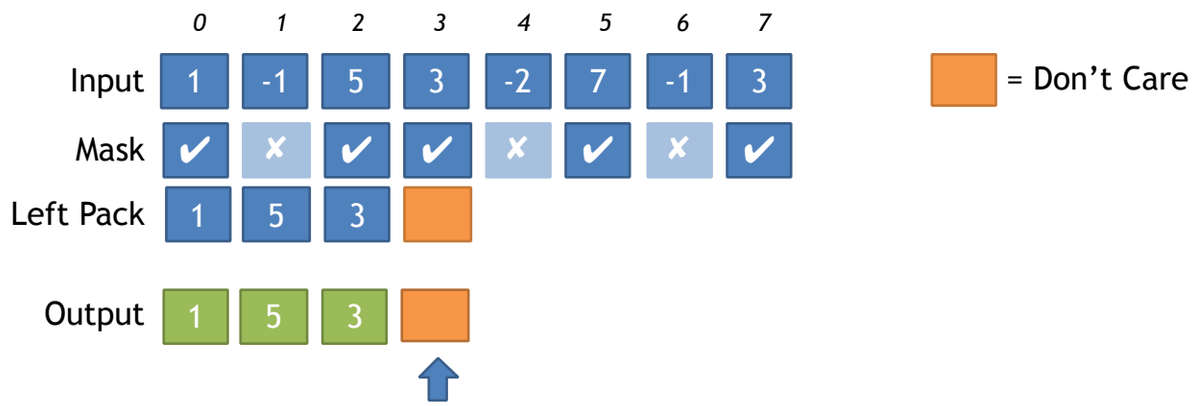
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



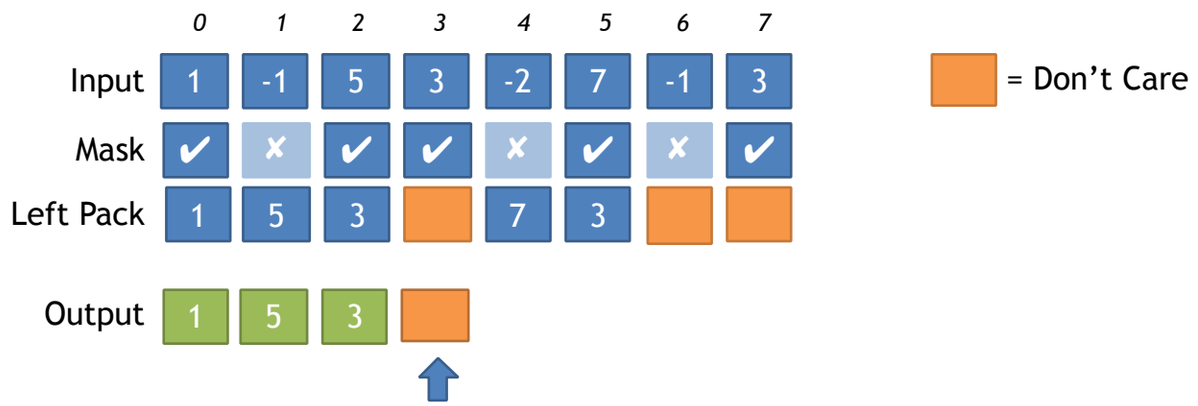
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



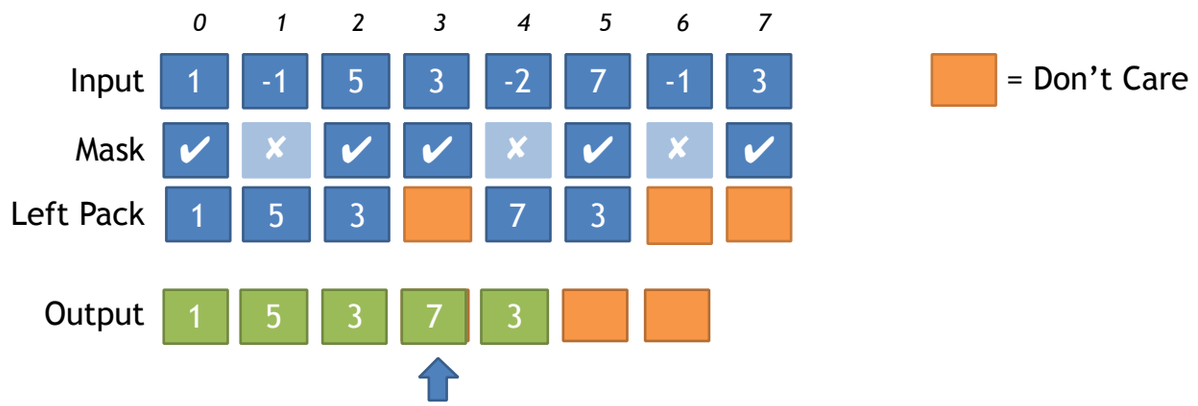
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



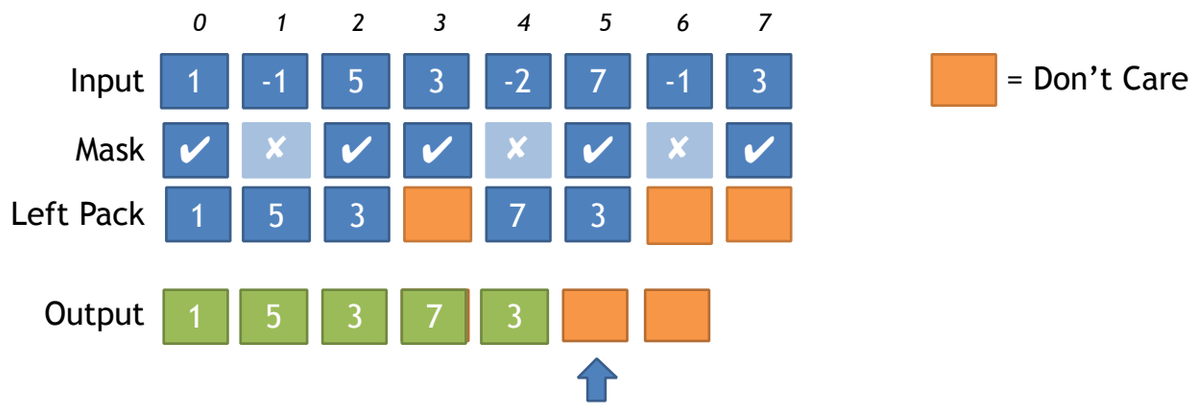
Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing Problem (4-wide, limit=0)



Here's our example input stream.
We load four values, and compare to generate a mask.
Using the mask we left-pack the element we want to keep.
We store a full 128-bit word to the output array.
The output pointer is incremented by the number of valid elements.
And we repeat the process for the next group of 4 elements.

Left Packing (SSSE3+)

- Leverage indirect shuffle via PSHUFB

a.k.a _mm_shuffle_epi8()

We can get an idea of what elements we want to keep by gathering the mask from our compare into a scalar register via _mm_movemask_ps(). This yields a 4-bit mask for the 4-wide case. We know from binary arithmetic that 4 bits can represent values between 0 and 15.
So we can use this 4-bit integer value directly as an index into a lookup table. In the lookup table we're going to store control words that control PSHUFB which is an SSSE3 instruction. PSHUFB is the only SSE instruction that can dynamically shuffle bytes around in a registers based on a control word.
Our lookup table will be 16 128-bit words, or 256 bytes of data which we'll align to a cache line boundary.
Let's see what this looks like in code.

Left Packing (SSSE3+)

- Leverage indirect shuffle via PSHUFB
- _mm_movemask_ps() = bit mask of valid lanes

a.k.a _mm_shuffle_epi8()

We can get an idea of what elements we want to keep by gathering the mask from our compare into a scalar register via _mm_movemask_ps(). This yields a 4-bit mask for the 4-wide case. We know from binary arithmetic that 4 bits can represent values between 0 and 15.
So we can use this 4-bit integer value directly as an index into a lookup table. In the lookup table we're going to store control words that control PSHUFB which is an SSSE3 instruction. PSHUFB is the only SSE instruction that can dynamically shuffle bytes around in a registers based on a control word.
Our lookup table will be 16 128-bit words, or 256 bytes of data which we'll align to a cache line boundary.
Let's see what this looks like in code.

Left Packing (SSSE3+)

- Leverage indirect shuffle via PSHUFB
- _mm_movemask_ps() = bit mask of valid lanes
- Lookup table of 16 shuffles (4-wide case)

a.k.a _mm_shuffle_epi8()

We can get an idea of what elements we want to keep by gathering the mask from our compare into a scalar register via _mm_movemask_ps(). This yields a 4-bit mask for the 4-wide case. We know from binary arithmetic that 4 bits can represent values between 0 and 15.
So we can use this 4-bit integer value directly as an index into a lookup table. In the lookup table we're going to store control words that control PSHUFB which is an SSSE3 instruction. PSHUFB is the only SSE instruction that can dynamically shuffle bytes around in a registers based on a control word.
Our lookup table will be 16 128-bit words, or 256 bytes of data which we'll align to a cache line boundary.
Let's see what this looks like in code.

Left Packing (SSSE3+)

- Leverage indirect shuffle via PSHUFB
- _mm_movemask_ps() = bit mask of valid lanes
- Lookup table of 16 shuffles (4-wide case)
- Need 16 x 16 = 256 bytes (4 cache lines) of LUT

a.k.a _mm_shuffle_epi8()

We can get an idea of what elements we want to keep by gathering the mask from our compare into a scalar register via _mm_movemask_ps(). This yields a 4-bit mask for the 4-wide case. We know from binary arithmetic that 4 bits can represent values between 0 and 15.
So we can use this 4-bit integer value directly as an index into a lookup table. In the lookup table we're going to store control words that control PSHUFB which is an SSSE3 instruction. PSHUFB is the only SSE instruction that can dynamically shuffle bytes around in a registers based on a control word.
Our lookup table will be 16 128-bit words, or 256 bytes of data which we'll align to a cache line boundary.
Let's see what this looks like in code.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.

We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.

The shuffle data is precomputed and can be stored in .rodata.

For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.

There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

04	05	06	07	0C	0D	0E	0F	80	80	80	80	80	80	80	80

= YW00

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Here's how the SSSE3+ left packing code falls out.
We simply generate the 4-bit mask, look up a shuffle control word and perform the left-packing in a single shuffle.
The shuffle data is precomputed and can be stored in .rodata.
For example, if we're left-packing Y and W into the first two positions, the PSHUFB control data we need looks like this.
There are many variants of this problem that you will encounter in SIMD programming, for example generating index lists. So it's a good technique to know.

Problem: Dynamic Masks

- Want mask that isolates ***n*** lower bits per lane
  - ***n*** varies across SIMD register
  - Useful for dynamic fixed point & many other things
- Easy in scalar code: `(1 << n) - 1`
- No SSE instruction to do variable shifts per lane

TARGET: 39:00

The next problem we'll look at is that of dynamically generating a mask of the N lower set bits. We'd need a mask like this to isolate some variable number of bits in each SIMD lane. This can come up when you're working with fixed-point data which has been encoded with varying precision, or when doing certain other tricks with IEEE floats.

In scalar code this is pretty easy, to isolate **n** bits we simply shift 1 to the left by **n** steps and subtract 1 to get our mask.

This same approach works in some other SIMD instruction sets, but SSE doesn't have a shift instruction that can shift each lane by a different amount.

LUT Low Mask Generation, n = 17

	LUT
0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

Let’s take a little sidetrack here. Besides the shifting, another way to generate a low mask is to use a table lookup for each byte in the result. Each byte in a mask like this can take on 9 distinct values, from 0 to 255 (or FF) as you can see in the LUT here to the right.

If we take our **n** and clamp it to various ranges representing each byte in a 32-bit mask we can perform 4 table lookups to build the output.

In scalar code it would be madness to do this, as the shift and subtract method is a lot faster. And it doesn’t seem to help our SIMD code either - how can we possible hope to do 16 (4 x 4) table lookups?

LUT Low Mask Generation, n = 17

Index 0: Clamp(n , 0, 8) = 8

	LUT
0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

Let’s take a little sidetrack here. Besides the shifting, another way to generate a low mask is to use a table lookup for each byte in the result. Each byte in a mask like this can take on 9 distinct values, from 0 to 255 (or FF) as you can see in the LUT here to the right.

If we take our **n** and clamp it to various ranges representing each byte in a 32-bit mask we can perform 4 table lookups to build the output.

In scalar code it would be madness to do this, as the shift and subtract method is a lot faster. And it doesn’t seem to help our SIMD code either - how can we possible hope to do 16 (4 x 4) table lookups?

LUT Low Mask Generation, n = 17


```
Index 0: Clamp(n   , 0, 8)    = 8
Index 1: Clamp(n-8 , 0, 8)    = 8
```

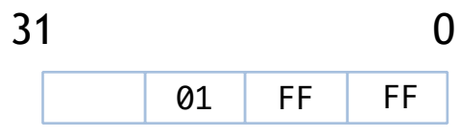
	LUT
0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

Let’s take a little sidetrack here. Besides the shifting, another way to generate a low mask is to use a table lookup for each byte in the result. Each byte in a mask like this can take on 9 distinct values, from 0 to 255 (or FF) as you can see in the LUT here to the right.

If we take our **n** and clamp it to various ranges representing each byte in a 32-bit mask we can perform 4 table lookups to build the output.

In scalar code it would be madness to do this, as the shift and subtract method is a lot faster. And it doesn’t seem to help our SIMD code either - how can we possible hope to do 16 (4 x 4) table lookups?

LUT Low Mask Generation, n = 17




```
Index 0: Clamp(n   , 0, 8)    = 8
Index 1: Clamp(n-8 , 0, 8)    = 8
Index 2: Clamp(n-16, 0, 8)    = 1
```

LUT

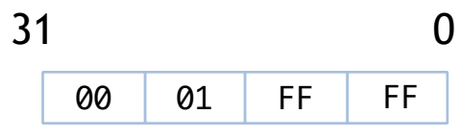
0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

Let’s take a little sidetrack here. Besides the shifting, another way to generate a low mask is to use a table lookup for each byte in the result. Each byte in a mask like this can take on 9 distinct values, from 0 to 255 (or FF) as you can see in the LUT here to the right.

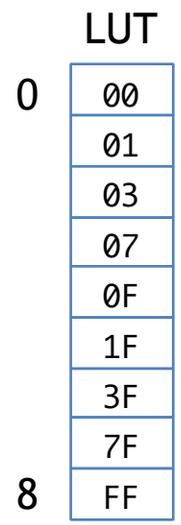
If we take our **n** and clamp it to various ranges representing each byte in a 32-bit mask we can perform 4 table lookups to build the output.

In scalar code it would be madness to do this, as the shift and subtract method is a lot faster. And it doesn’t seem to help our SIMD code either - how can we possible hope to do 16 (4 x 4) table lookups?

LUT Low Mask Generation, n = 17




```
Index 0: Clamp(n   , 0, 8)    = 8
Index 1: Clamp(n-8 , 0, 8)    = 8
Index 2: Clamp(n-16, 0, 8)    = 1
Index 3: Clamp(n-24, 0, 8)    = 0
```



Let’s take a little sidetrack here. Besides the shifting, another way to generate a low mask is to use a table lookup for each byte in the result. Each byte in a mask like this can take on 9 distinct values, from 0 to 255 (or FF) as you can see in the LUT here to the right.

If we take our **n** and clamp it to various ranges representing each byte in a 32-bit mask we can perform 4 table lookups to build the output.

In scalar code it would be madness to do this, as the shift and subtract method is a lot faster. And it doesn’t seem to help our SIMD code either - how can we possible hope to do 16 (4 x 4) table lookups?

Dynamic Masking (SSSE3+)

- PSHUFB can be used as nibble->byte lookup
  - 16 parallel lookups
  - Works for this problem because we only have 9 cases
- Index computation
  - Use *saturated* addition & subtraction
  - Compute all 16 offset & clamped indices in parallel

The saving grace is that PSHUFB can be considered a parallel lookup table instruction.
One SIMD register can be set up to hold our lookup table of 9 byte values, and we can then generate all 16 bytes of mask output with one instruction.
The question then is how do we generate all these clamped indices we will need?
SSE offers saturated addition and subtraction instructions that I think are underused - they combine an addition and a subtraction but prevent the results from leaving the range 0-255 (for the byte case.) So one way to look at a saturated add is as an add followed by a MIN with 255, and saturated subtract as a sub followed by MAX with 0.
If we carefully craft constants for a pair of saturated add and sub instructions we can shift all indices into range and clamp them at the same time.

PSHUFB Dynamic Mask Generation

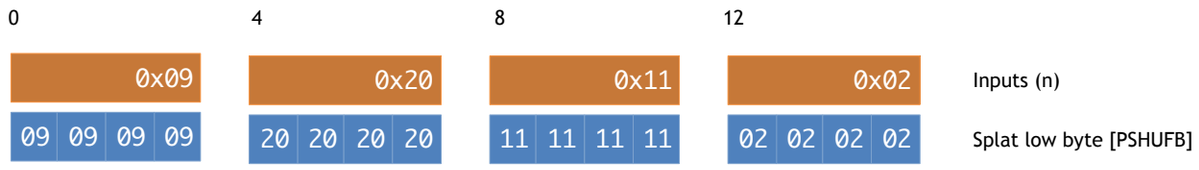
0 4 8 12

Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation

Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation

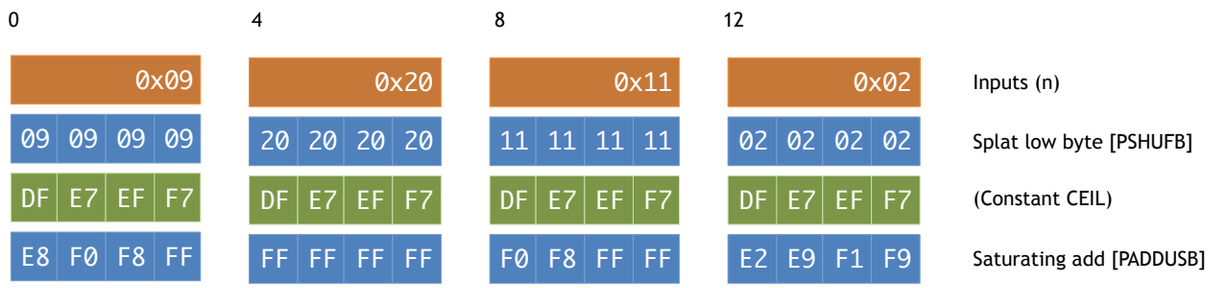


Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation

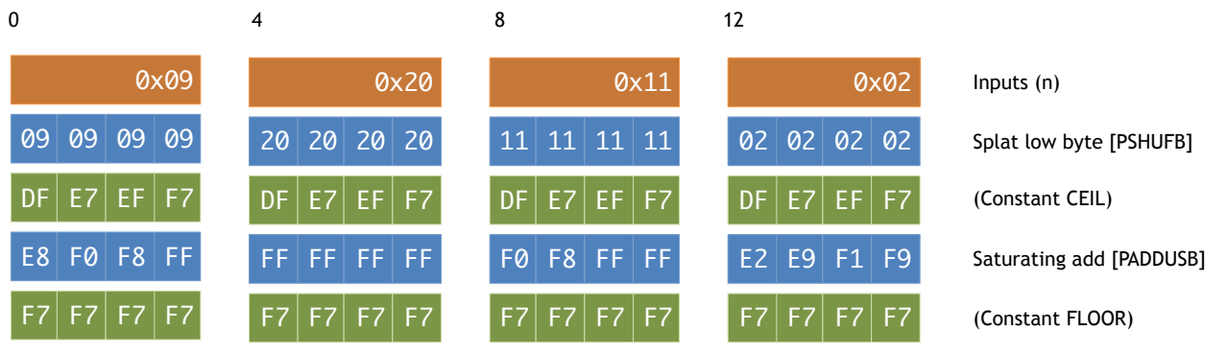
Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation



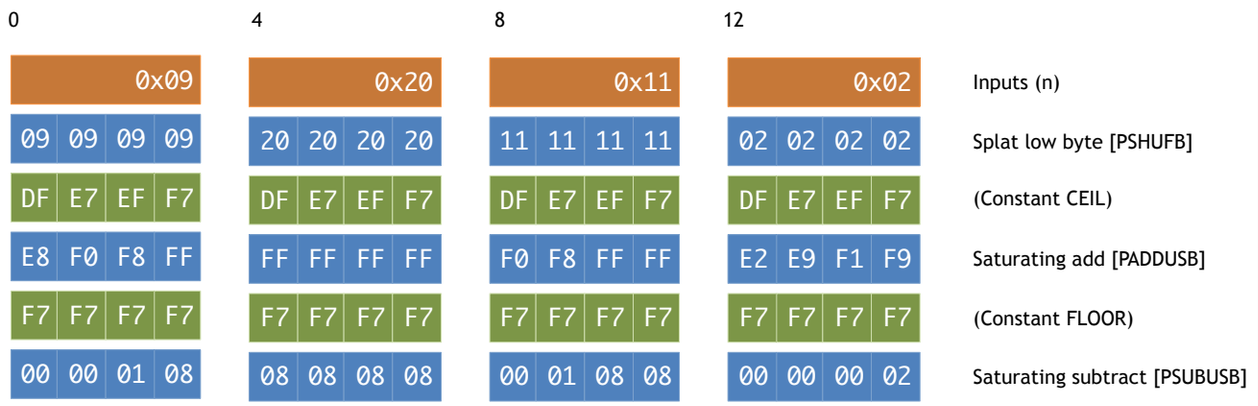
Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation



Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation

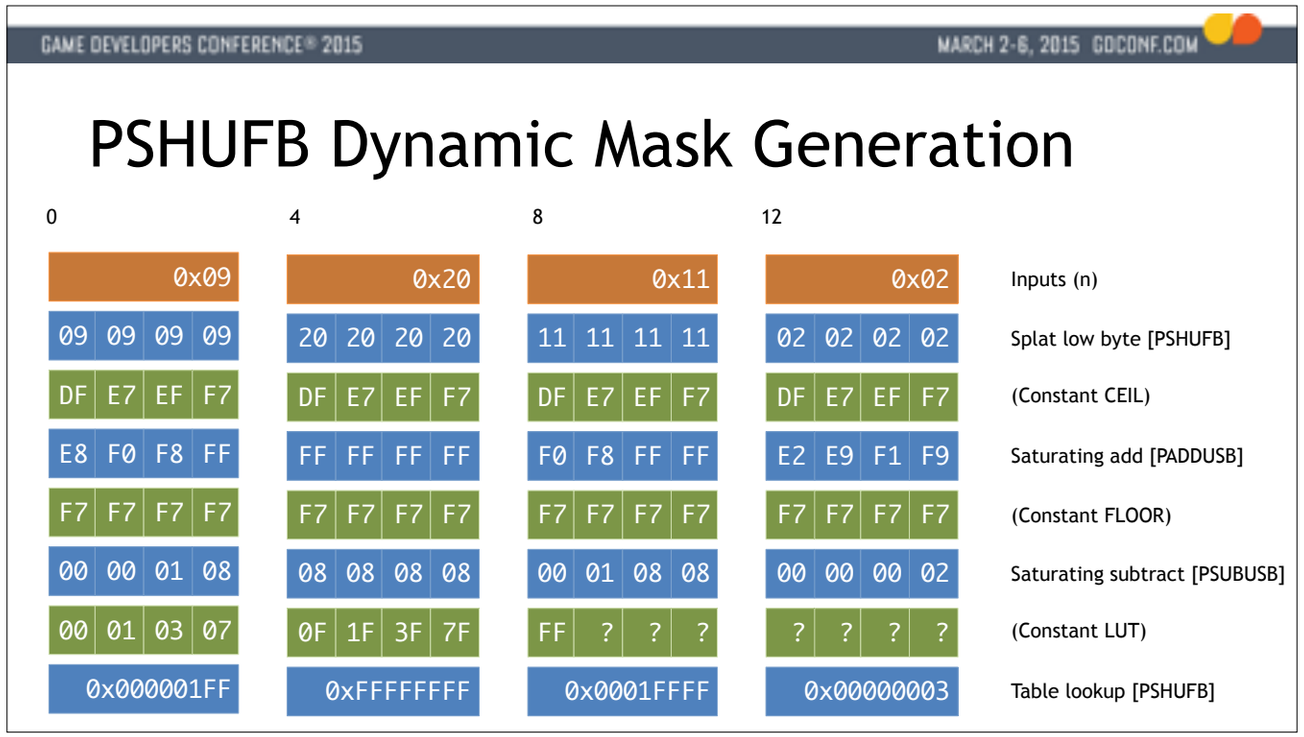


Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

PSHUFB Dynamic Mask Generation

0	4	8	12	
0x09	0x20	0x11	0x02	Inputs (n)
09 09 09 09	20 20 20 20	11 11 11 11	02 02 02 02	Splat low byte [PSHUFB]
DF E7 EF F7	DF E7 EF F7	DF E7 EF F7	DF E7 EF F7	(Constant CEIL)
E8 F0 F8 FF	FF FF FF FF	F0 F8 FF FF	E2 E9 F1 F9	Saturating add [PADDUSB]
F7 F7 F7 F7	F7 F7 F7 F7	F7 F7 F7 F7	F7 F7 F7 F7	(Constant FLOOR)
00 00 01 08	08 08 08 08	00 01 08 08	00 00 00 02	Saturating subtract [PSUBUSB]
00 01 03 07	0F 1F 3F 7F	FF ? ? ?	? ? ? ?	(Constant LUT)

Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.



Here's how that plays out.
We start with 4 distinct 32-bit N values (our number of bits)
We replicate the lowest byte 4 times.
Next we'll need a constant that is staggered to push successive bytes up towards the 255 ceiling. We chose the values so that a value over 8 will be clamped to 255.
We perform a saturating add, which is the MIN part.
Next we need up a constant to bring the values back again. We'll subtract 0xF7 across the board.
We perform a saturating subtract - this is the MAX part
At this point we have indices in the range 0-8 inclusively in every byte.
Now we set up the control word for PSHUFB - we need the 8 distinct bit patterns to form the mask loaded in the first 8 bytes.
We perform the shuffle, and out falls the 32-bit masks.

Dynamic Mask Routine (SSSE3)

```
__m128i MaskLowBits_SSSE3(__m128i n)
{
  __m128i ii = _mm_shuffle_epi8(n, BYTES);
  __m128i si = _mm_adds_epu8(ii, CEIL);
          si = _mm_subs_epu8(si, FLOOR);
  return       _mm_shuffle_epi8(LUT, si);
}
```

This is 3 or 4 cycles on most Intel chips, to compute 4 dynamic masks.

SSE2 Techniques

TARGET: 44:00

Emulation in terms of “simpler” instructions can be faster, they have gotten a lot of attention.
SSE2 really strictly SOA with only fixed shuffles, hard to do tricks there.

SSE2 Techniques

- SSE2 is ancient, but fine for basic SOA SIMD
  - Massive speedups still possible
  - Sometimes basic SSE2 will beat SSE4.1 on same HW

TARGET: 44:00

Emulation in terms of “simpler” instructions can be faster, they have gotten a lot of attention.
SSE2 really strictly SOA with only fixed shuffles, hard to do tricks there.

SSE2 Techniques

- SSE2 is ancient, but fine for basic SOA SIMD
  - Massive speedups still possible
  - Sometimes basic SSE2 will beat SSE4.1 on same HW
- Harder to do “unusual” things with SSE2
  - Only fixed shuffles
  - Integer support lackluster

TARGET: 44:00

Emulation in terms of “simpler” instructions can be faster, they have gotten a lot of attention.
SSE2 really strictly SOA with only fixed shuffles, hard to do tricks there.

SSE2 Left Packing: Move Distances

- No dynamic shuffles
  - Need divide & conquer algorithm
- How far does each lane have to travel?

To perform left packing with only fixed shuffles we need a concept called move distances.
This indicates how far each lane has to travel to the left in the packed result.
For example for YW??, we need to move X nowhere, because it's invalid. Y moves 1 lane over, to X.
Z is thrown away and W needs to move two lanes over.

SSE2 Left Packing: Move Distances

- No dynamic shuffles
  - Need divide & conquer algorithm
- How far does each lane have to travel?

Mask	Output	Move Distances			
0000	....	0	0	0	0
0001	X...	0	0	0	0
0010	Y...	0	1	0	0
0011	XY..	0	0	0	0
0100	Z...	0	0	2	0
0101	XZ..	0	0	1	0
0110	YZ..	0	1	1	0
0111	XYZ.	0	0	0	0
1000	W...	0	0	0	3
1001	XW..	0	0	0	2
1010	YW..	0	1	0	2
1011	XYW.	0	0	0	1
1100	ZW..	0	0	2	2
1101	XZW.	0	0	1	1
1110	YZW.	0	1	1	1
1111	XYZW	0	0	0	0

To perform left packing with only fixed shuffles we need a concept called move distances.
This indicates how far each lane has to travel to the left in the packed result.
For example for YW??, we need to move X nowhere, because it's invalid. Y moves 1 lane over, to X.
Z is thrown away and W needs to move two lanes over.

SSE2 Left Packing: Move Distances

- No dynamic shuffles
  - Need divide & conquer algorithm
- How far does each lane have to travel?

Mask	Output	Move Distances			
0000	....	0	0	0	0
0001	X...	0	0	0	0
0010	Y...	0	1	0	0
0011	XY..	0	0	0	0
0100	Z...	0	0	2	0
0101	XZ..	0	0	1	0
0110	YZ..	0	1	1	0
0111	XYZ.	0	0	0	0
1000	W...	0	0	0	3
1010	YW..	0	1	0	2
1100	ZW..	0	0	2	2
1101	XZW.	0	0	1	1
1110	YZW.	0	1	1	1
1111	XYZW	0	0	0	0

To perform left packing with only fixed shuffles we need a concept called move distances.
This indicates how far each lane has to travel to the left in the packed result.
For example for YW??, we need to move X nowhere, because it's invalid. Y moves 1 lane over, to X.
Z is thrown away and W needs to move two lanes over.

Left Packing with Move Distances

- Process move distances (MD) bit by bit
  - Rotate left by 1 – Select based on Bit 0 of MD
  - Rotate left by 2 – Select based on Bit 1 of MD
  - And so on..

We can now process the individual bits of the move distances by moving with fixed shuffles.
Rotate by 1, select based on bit zero. Then 2, then 4, etc...
Generalizes. For example 16-bit left pack, or 8x AVX float left pack
2 for 4-wide case, 3 for 8-wide case, …

Left Packing with Move Distances

- Process move distances (MD) bit by bit
  - Rotate left by 1 – Select based on Bit 0 of MD
  - Rotate left by 2 – Select based on Bit 1 of MD
  - And so on..
- Generalizes to wider registers & more elements
  - $\log_2(n)$ rounds of rotate + select required

We can now process the individual bits of the move distances by moving with fixed shuffles.
Rotate by 1, select based on bit zero. Then 2, then 4, etc...
Generalizes. For example 16-bit left pack, or 8x AVX float left pack
2 for 4-wide case, 3 for 8-wide case, …

Left-packing `YW..` (simplified)

We can now do fixed shuffles and selects to perform our left packing.

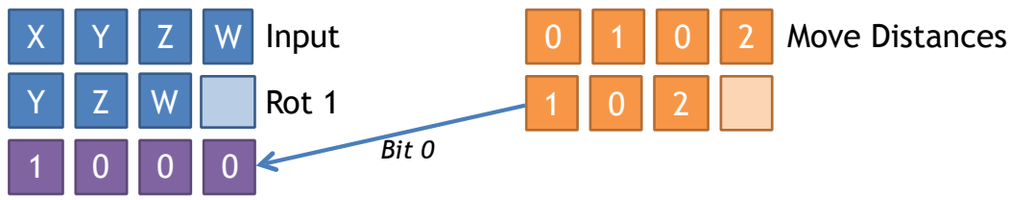
Left-packing `YW..` (simplified)

We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)

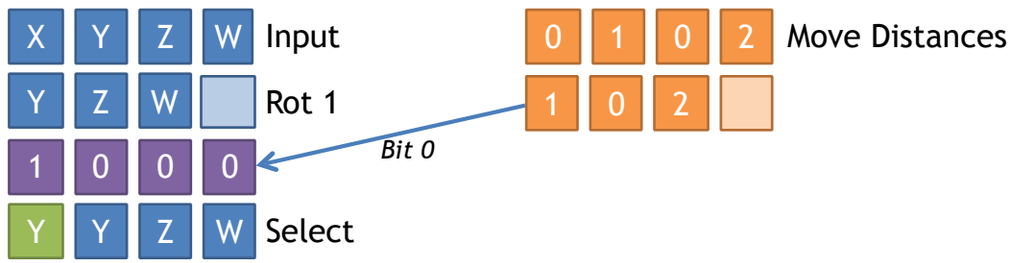
We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)



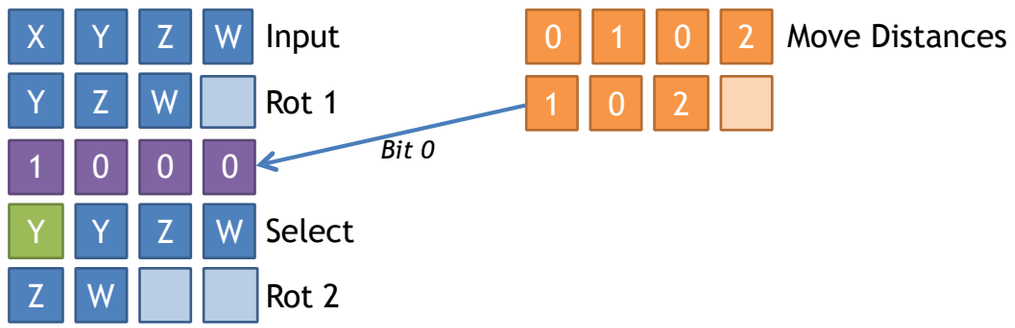
We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)



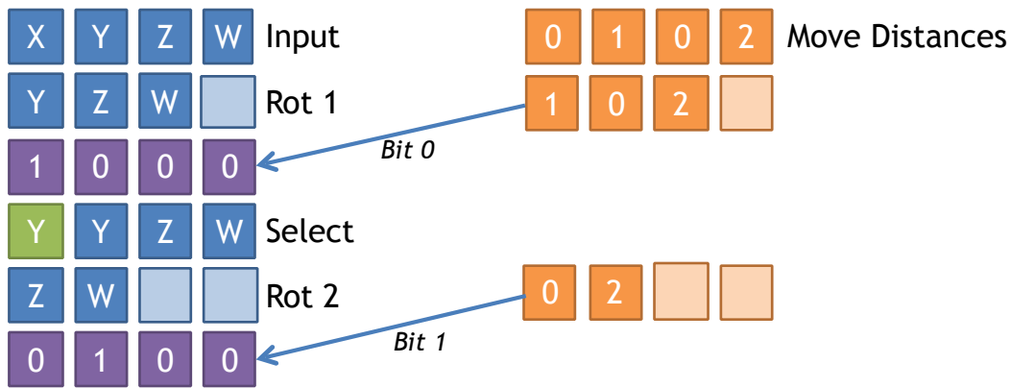
We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)



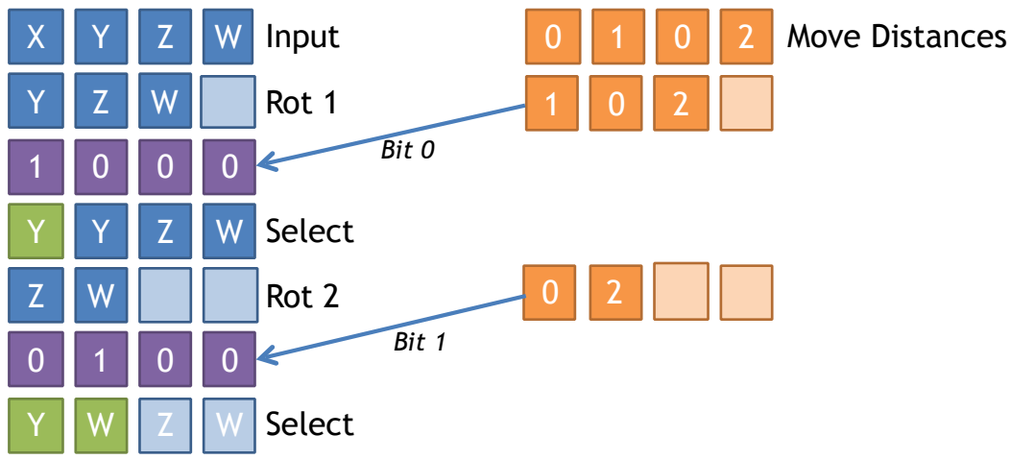
We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)

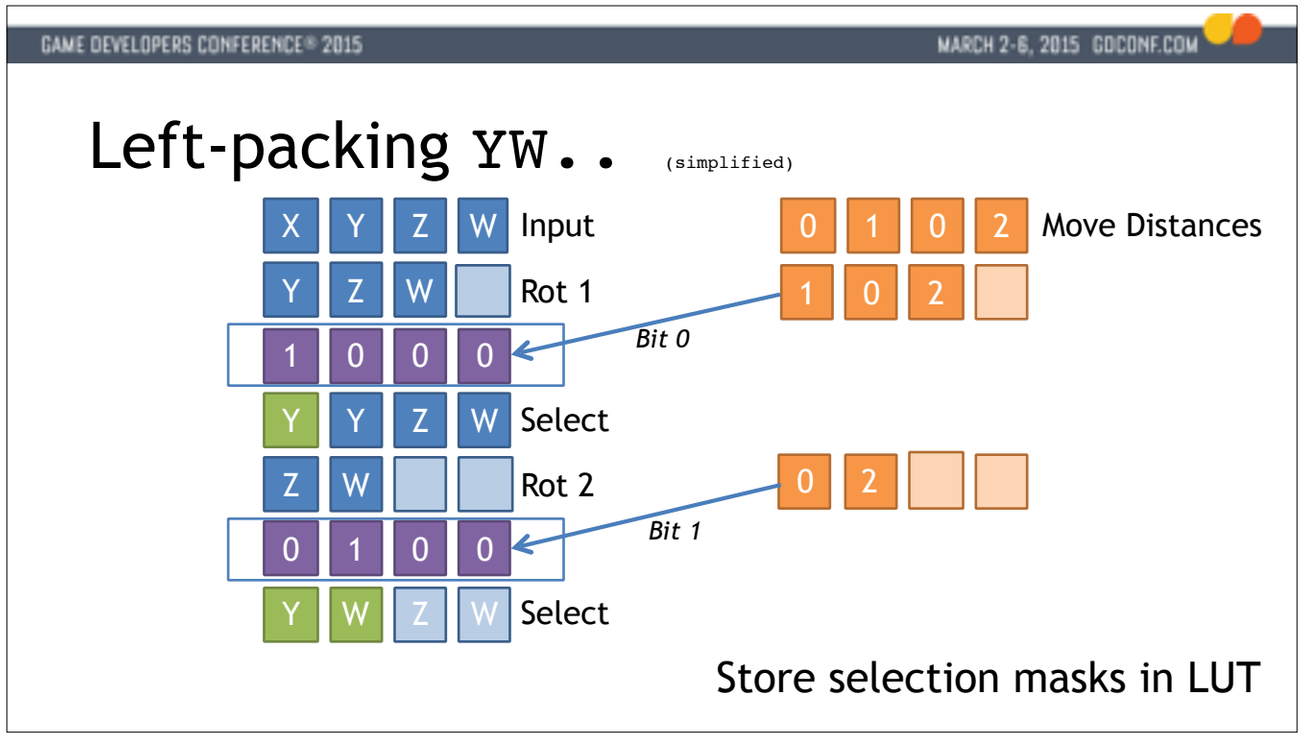


We can now do fixed shuffles and selects to perform our left packing.

Left-packing `YW..` (simplified)



We can now do fixed shuffles and selects to perform our left packing.



We can now do fixed shuffles and selects to perform our left packing.

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.

We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Grab mask of valid elements

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.

We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Load precomputed selection masks from LUT

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.

We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

First round of rotate+select

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.
We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Second round of rotate+select

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.
We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

This solution is slower than the PSHUFB based solution we saw earlier, but still pretty speedy.

We need twice the LUT space - 32 128-bit words for a total of 512 bytes (or 8 cache lines)

Dynamic Masks in SSE2

We'll need some float hackery. SSE2 doesn't really have anything to cover us here.

Dynamic Masks in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa

We'll need some float hackery. SSE2 doesn't really have anything to cover us here.

Dynamic Masks in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa
- That exponent sure looks like a shifter..
  - $2^n = 1 << n$

We'll need some float hackery. SSE2 doesn't really have anything to cover us here.

Dynamic Masks in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa
- That exponent sure looks like a shifter..
  - $2^n = 1 << n$
- Idea:
  - Craft special floats by populating exponent with biased ***n***
  - Convert to integer, then subtract 1

We'll need some float hackery. SSE2 doesn't really have anything to cover us here.

Overflow woes

- Conversion from float to int is signed

Overflow woes

- Conversion from float to int is signed
- When n >= 31, can't fit in signed integer
  - INT_MAX = `0x7fffffff`
  - Overflow is clamped to "integer indeterminate"

Overflow woes

- Conversion from float to int is signed
- When n >= 31, can't fit in signed integer
  - INT_MAX = `0x7fffffff`
  - Overflow is clamped to "integer indeterminate"
- Which happens to be.. `0x80000000`
  - Exactly what we need for n = 31
  - n > 31 will clamp to 31

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Add 127 to generate biased exponent

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Move exponent into place to make it pass as a float

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Convert the float to an int yielding 2^n as an integer

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Subtract one to generate mask

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Dynamic Masks in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return         _mm_sub_epi32(intv, c_1);
}
```

Slower than SSSE3 solution by 1-3 cycles depending on microarch.

Especially costly on Intel Core micro architecture due to int->float reinterpretation.

Best Practices: Branching

TARGET: 50:00

A mispredicted branch is around 10-30 cycles depending on HW.

Don’t want hard-to-predict branches in inner loops

The branch for the loop itself is fine, as it will be predicted correctly!

Branch should be predicted correctly 99+% to make sense

E.g. a handful of expensive things in a sea of data

Use _mm_movemask_X()+ if on SSE2

Consider: _mm_testz_si128() and friends on SSE4.1+

Best Practices: Branching

- Branch mispredictions are very costly

TARGET: 50:00
A mispredicted branch is around 10-30 cycles depending on HW.
Don’t want hard-to-predict branches in inner loops
The branch for the loop itself is fine, as it will be predicted correctly!
Branch should be predicted correctly 99+% to make sense
E.g. a handful of expensive things in a sea of data
Use _mm_movemask_X()+ if on SSE2
Consider: _mm_testz_si128() and friends on SSE4.1+

Best Practices: Branching

- Branch mispredictions are very costly
- Guideline: Avoid branches in general

TARGET: 50:00
A mispredicted branch is around 10-30 cycles depending on HW.
Don’t want hard-to-predict branches in inner loops
The branch for the loop itself is fine, as it will be predicted correctly!
Branch should be predicted correctly 99+% to make sense
E.g. a handful of expensive things in a sea of data
Use _mm_movemask_X()+ if on SSE2
Consider: _mm_testz_si128() and friends on SSE4.1+

Best Practices: Branching

- Branch mispredictions are very costly
- Guideline: Avoid branches in general
- Exception: Can be OK if *very* predictable

TARGET: 50:00
A mispredicted branch is around 10-30 cycles depending on HW.
Don’t want hard-to-predict branches in inner loops
The branch for the loop itself is fine, as it will be predicted correctly!
Branch should be predicted correctly 99+% to make sense
E.g. a handful of expensive things in a sea of data
Use _mm_movemask_X()+ if on SSE2
Consider: _mm_testz_si128() and friends on SSE4.1+

Alternatives to Branching

- GPU-style “compute both branches” + select

GPU-style select branching works fine for many smaller problems
Start here for small branches.
Yields best performance when possible

Run fast kernel to partition index data into multiple sets
Run optimized kernel on each subset
Prefetching can be useful unless most indices are visited

Alternatives to Branching

- GPU-style “compute both branches” + select
- Separate input data + routines when possible

GPU-style select branching works fine for many smaller problems
Start here for small branches.
Yields best performance when possible

Run fast kernel to partition index data into multiple sets
Run optimized kernel on each subset
Prefetching can be useful unless most indices are visited

Alternatives to Branching

- GPU-style “compute both branches” + select
- Separate input data + routines when possible
- Consider partitioned index sets

GPU-style select branching works fine for many smaller problems
Start here for small branches.
Yields best performance when possible

Run fast kernel to partition index data into multiple sets
Run optimized kernel on each subset
Prefetching can be useful unless most indices are visited

Best Practices: Prefetching

- 100% required on previous generation HW

Not a good idea to carry this forward blindly to x86

Can carry a heavy TLB miss cost chance on some H/W
The chip is already prefetching at the cache level for free

IF: you know they will be far enough apart/irregular
Prefetch instructions vary somewhat between AMD/Intel
Test carefully that you’re getting benefit on all H/W

Best Practices: Prefetching

- 100% required on previous generation HW
- Guideline: Don’t prefetch linear array accesses

Not a good idea to carry this forward blindly to x86

Can carry a heavy TLB miss cost chance on some H/W
The chip is already prefetching at the cache level for free

IF: you know they will be far enough apart/irregular
Prefetch instructions vary somewhat between AMD/Intel
Test carefully that you’re getting benefit on all H/W

Best Practices: Prefetching

- 100% required on previous generation HW
- Guideline: Don’t prefetch linear array accesses
- Guideline: *Maybe* prefetch upcoming ptrs/indices

Not a good idea to carry this forward blindly to x86

Can carry a heavy TLB miss cost chance on some H/W
The chip is already prefetching at the cache level for free

IF: you know they will be far enough apart/irregular
Prefetch instructions vary somewhat between AMD/Intel
Test carefully that you’re getting benefit on all H/W

Best Practices: Unrolling

- Common in VMX128/SPU style code

Made a lot of sense with in-order machines to hide latency
Also had lots of registers!

Only 16 (named) registers - H/W has many more internally
Out of order execution unrolls for you to some extent

E.g. unroll 2x 64-bit loop to get 128 bit loop, but no more
Can make exceptions for very small loops as needed

Best Practices: Unrolling

- Common in VMX128/SPU style code
- Generally a waste of time for SSE/AVX

Made a lot of sense with in-order machines to hide latency
Also had lots of registers!

Only 16 (named) registers - H/W has many more internally
Out of order execution unrolls for you to some extent

E.g. unroll 2x 64-bit loop to get 128 bit loop, but no more
Can make exceptions for very small loops as needed

Best Practices: Unrolling

- Common in VMX128/SPU style code
- Generally a waste of time for SSE/AVX
- Guideline: Unroll only up to full register width

Made a lot of sense with in-order machines to hide latency
Also had lots of registers!

Only 16 (named) registers - H/W has many more internally
Out of order execution unrolls for you to some extent

E.g. unroll 2x 64-bit loop to get 128 bit loop, but no more
Can make exceptions for very small loops as needed

Best Practices: Unrolling

- Common in VMX128/SPU style code
- Generally a waste of time for SSE/AVX
- Guideline: Unroll only up to full register width
- Exception: For *very* small loops.

Made a lot of sense with in-order machines to hide latency
Also had lots of registers!

Only 16 (named) registers - H/W has many more internally
Out of order execution unrolls for you to some extent

E.g. unroll 2x 64-bit loop to get 128 bit loop, but no more
Can make exceptions for very small loops as needed

Best Practices: Streaming loads+stores

- E.g. _mm_stream_ps, _mm_stream_load_si128

SSE 4.1 feature to do streaming reads.
Helps avoid cache trashing
Especially for kernels using large lookup tables

Different options for different architectures
_mm_mfence() always works but is slow
Streaming sidesteps strong x86 memory model
Subtle data races will happen if you don’t fence

Best Practices: Streaming loads+stores

- E.g. _mm_stream_ps, _mm_stream_load_si128
- Guideline: Use these to avoid trashing cache

SSE 4.1 feature to do streaming reads.
Helps avoid cache trashing
Especially for kernels using large lookup tables

Different options for different architectures
_mm_mfence() always works but is slow
Streaming sidesteps strong x86 memory model
Subtle data races will happen if you don’t fence

Best Practices: Streaming loads+stores

- E.g. _mm_stream_ps, _mm_stream_load_si128
- Guideline: Use these to avoid trashing cache
- Guideline: Get the routine right first

SSE 4.1 feature to do streaming reads.
Helps avoid cache trashing
Especially for kernels using large lookup tables

Different options for different architectures
_mm_mfence() always works but is slow
Streaming sidesteps strong x86 memory model
Subtle data races will happen if you don’t fence

Best Practices: Streaming loads+stores

- E.g. _mm_stream_ps, _mm_stream_load_si128
- Guideline: Use these to avoid trashing cache
- Guideline: Get the routine right first
- Don’t forget to fence! (e.g. _mm_sfence)

SSE 4.1 feature to do streaming reads.
Helps avoid cache trashing
Especially for kernels using large lookup tables

Different options for different architectures
_mm_mfence() always works but is slow
Streaming sidesteps strong x86 memory model
Subtle data races will happen if you don’t fence

Conclusion

Not a lot of best practices out there

Hopefully this talk gives you something to start with!

Conclusion

- CPU SIMD is not magic and has wide applicability

Not a lot of best practices out there
Hopefully this talk gives you something to start with!

Conclusion

- CPU SIMD is not magic and has wide applicability
- Program the CPU directly, not via abstractions

Not a lot of best practices out there
Hopefully this talk gives you something to start with!

Conclusion

- CPU SIMD is not magic and has wide applicability
- Program the CPU directly, not via abstractions
- Lots of performance left on the table

Not a lot of best practices out there
Hopefully this talk gives you something to start with!

Conclusion

- CPU SIMD is not magic and has wide applicability
- Program the CPU directly, not via abstractions
- Lots of performance left on the table
- Tricks allow you to use SIMD in more places

Not a lot of best practices out there
Hopefully this talk gives you something to start with!

SSE and AVX Resources

- ISPC
  - http://ispc.github.io
- Intel Instrinsics Guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide
  - Available as Dash DocSet for Mac OS X by yours truly
- Intel Architecture Code Analyzer
  - https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
- Agner Fog's instruction timings
  - http://www.agner.org/optimize/#manuals

Q & A

Twitter: @deplinenoise
Email: afredriksson@insomniacgames.com

Special thanks:
Fabian Giesen
Mike Day

Bonus Material

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines

So now that we know HOW to write some SIMD code, what's available in the x86 space?

It's been 16 years (!) since SSE was originally released in 1999. If we take a fresh look today, SSE 4.1 and later offer a pretty good programming environment. The main challenge for PC devs is that the consumer space is heavily fragmented when it comes to SSE feature level support.

One big limitation compared to other SIMD ISAs is we only get 16 registers. This is really easy to fill up and you do tend to use values from memory quite a bit to compensate for this. This is something to keep an eye on, as a large loop might end up spilling lots of registers to the stack and that can slow you down quite a bit.

Another big limitation is that we don't have two-register shuffles, so some algorithms that were optimal on Altivec or SPU will need to be redesigned to use the more limited shuffle forms available with SSE. We'll look at this in more detail later in the talk.

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines
- Limitation #1: Only 16 registers (x64)
  - Easy to overcommit and start stack spilling
  - Carefully check generated assembly from intrinsics

So now that we know HOW to write some SIMD code, what's available in the x86 space?

It's been 16 years (!) since SSE was originally released in 1999. If we take a fresh look today, SSE 4.1 and later offer a pretty good programming environment. The main challenge for PC devs is that the consumer space is heavily fragmented when it comes to SSE feature level support.

One big limitation compared to other SIMD ISAs is we only get 16 registers. This is really easy to fill up and you do tend to use values from memory quite a bit to compensate for this. This is something to keep an eye on, as a large loop might end up spilling lots of registers to the stack and that can slow you down quite a bit.

Another big limitation is that we don't have two-register shuffles, so some algorithms that were optimal on Altivec or SPU will need to be redesigned to use the more limited shuffle forms available with SSE. We'll look at this in more detail later in the talk.

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines
- Limitation #1: Only 16 registers (x64)
  - Easy to overcommit and start stack spilling
  - Carefully check generated assembly from intrinsics
- Limitation #2: No dynamic two-register shuffles
  - Challenge when porting Altivec/SPU style code

So now that we know HOW to write some SIMD code, what's available in the x86 space?

It's been 16 years (!) since SSE was originally released in 1999. If we take a fresh look today, SSE 4.1 and later offer a pretty good programming environment. The main challenge for PC devs is that the consumer space is heavily fragmented when it comes to SSE feature level support.

One big limitation compared to other SIMD ISAs is we only get 16 registers. This is really easy to fill up and you do tend to use values from memory quite a bit to compensate for this. This is something to keep an eye on, as a large loop might end up spilling lots of registers to the stack and that can slow you down quite a bit.

Another big limitation is that we don't have two-register shuffles, so some algorithms that were optimal on Altivec or SPU will need to be redesigned to use the more limited shuffle forms available with SSE. We'll look at this in more detail later in the talk.

SSE Goodies since SSE2

Technology	*Goodies*
SSSE3	***PSHUFB, Integer Abs***
SSE4.1	***32-bit low mul, Blend, Integer Min+Max, Insert + Extract, PTEST, PACKUSDW, …***
SSE4.2 *(POPCNT has its own CPUID flag)*	***POPCNT (only)***

I mentioned it's been 16 years since SSE came out. SSE 2 - which added integer instructions - is what we consider the baseline nowadays as it has near universal penetration.

SSSE3 is important (notice the extra S!) - this feature level adds PSHUFB which is a much more capable shuffle instruction that sees a lot of use in our code.

SSE4.1 is the big one - it's here you'll find a non-crippled integer multiply, the blend (select) instructions, integer min and max, and lots of other goodies

SSE4.2 is weird - it's really a bunch of crypto and string instructions that we certainly don't have any use for, but it came out with a bunch of unrelated scalar instructions for bit manipulation that we use a lot, such as population count.

SSE Fragmentation Nov 2014

Data kindly provided by Unity

Technology	Web Player	Unity Editor	Year Introduced
SSE2	100%	100%	2001
SSE3	100%	100%	2004
SSSE3	75%	93%	2006
SSE4.1	51%	83%	2007
SSE4.2	44%	77%	2008
AVX	23%	61%	2011
AVX2	4%	19%	2013

So what can you use? If you're a console dev, you can use all of it, and then some.
But if you're making PC games you'll have to carefully weigh your options.
Aras over at Unity provided me with this data in Nov 2014 - there are two trends I think are worth highlighting here:
1. People hang on to their old PCs for a long time
2. Developers have far better machines than the average consumer
If you decide to go for the SSE4.1 feature level you're leaving a large group of lower-end PCs behind.
This could be a real problem for a casual game. The only real solution to that is to provide multiple implementations of certain routines and switch between them; either at compile time or at runtime.
If you switch at compile time you can emulate most SSE 4.1 instructions in terms of SSE 2 and target a sort of common subset.
On the other hand if you're making a AAA title with high system demands you could possibly target SSE 4.1 exclusively.
AVX however is so far another story..

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!

So let's talk about AVX for a minute.
It's basically 256 bit wide ALU instead of 128 bits, but otherwise very similar to SSE.
If you're on an Intel CPU and you have AVX - good for you! It can really give you that 2x speed boost in some cases - I already mentioned the texture compression library we use that gets a nice speed boost from AVX. But a caveat is memory bandwidth - well tuned SSE code can be memory bound, in which case AVX will do exactly nothing as you're already using the full memory bandwidth.
The real limitation is of course the availability - AVX was only introduced with Sandy Bridge and availability isn't high enough that we can target it exclusively. We have plenty of workstations around the office that don't support AVX.
The other problem is AMD chips. All AMD chips I know of that support AVX split the ops internally into two 128 bit operations which drives up the latency of the instructions compared to SSE. We haven't found any compelling speedups at all on AMD chips.
So the bottom line for us is that we don't focus on AVX at all - that might change when support is more wide spread and/or AMD supports it better.

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space

So let's talk about AVX for a minute.
It's basically 256 bit wide ALU instead of 128 bits, but otherwise very similar to SSE.
If you're on an Intel CPU and you have AVX - good for you! It can really give you that 2x speed boost in some cases - I already mentioned the texture compression library we use that gets a nice speed boost from AVX. But a caveat is memory bandwidth - well tuned SSE code can be memory bound, in which case AVX will do exactly nothing as you're already using the full memory bandwidth.
The real limitation is of course the availability - AVX was only introduced with Sandy Bridge and availability isn't high enough that we can target it exclusively. We have plenty of workstations around the office that don't support AVX.
The other problem is AMD chips. All AMD chips I know of that support AVX split the ops internally into two 128 bit operations which drives up the latency of the instructions compared to SSE. We haven't found any compelling speedups at all on AMD chips.
So the bottom line for us is that we don't focus on AVX at all - that might change when support is more wide spread and/or AMD supports it better.

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space
- Crippled on AMD micro architectures
  - Splits to 2 x 128 bit ALU internally (high latency)

So let's talk about AVX for a minute.
It's basically 256 bit wide ALU instead of 128 bits, but otherwise very similar to SSE.
If you're on an Intel CPU and you have AVX - good for you! It can really give you that 2x speed boost in some cases - I already mentioned the texture compression library we use that gets a nice speed boost from AVX. But a caveat is memory bandwidth - well tuned SSE code can be memory bound, in which case AVX will do exactly nothing as you're already using the full memory bandwidth.
The real limitation is of course the availability - AVX was only introduced with Sandy Bridge and availability isn't high enough that we can target it exclusively. We have plenty of workstations around the office that don't support AVX.
The other problem is AMD chips. All AMD chips I know of that support AVX split the ops internally into two 128 bit operations which drives up the latency of the instructions compared to SSE. We haven't found any compelling speedups at all on AMD chips.
So the bottom line for us is that we don't focus on AVX at all - that might change when support is more wide spread and/or AMD supports it better.

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space
- Crippled on AMD micro architectures
  - Splits to 2 x 128 bit ALU internally (high latency)
- Not worth it for us, except for some PC tools

So let's talk about AVX for a minute.
It's basically 256 bit wide ALU instead of 128 bits, but otherwise very similar to SSE.
If you're on an Intel CPU and you have AVX - good for you! It can really give you that 2x speed boost in some cases - I already mentioned the texture compression library we use that gets a nice speed boost from AVX. But a caveat is memory bandwidth - well tuned SSE code can be memory bound, in which case AVX will do exactly nothing as you're already using the full memory bandwidth.
The real limitation is of course the availability - AVX was only introduced with Sandy Bridge and availability isn't high enough that we can target it exclusively. We have plenty of workstations around the office that don't support AVX.
The other problem is AMD chips. All AMD chips I know of that support AVX split the ops internally into two 128 bit operations which drives up the latency of the instructions compared to SSE. We haven't found any compelling speedups at all on AMD chips.
So the bottom line for us is that we don't focus on AVX at all - that might change when support is more wide spread and/or AMD supports it better.

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance

As console developers we target all the features our consoles have, because we know the that the HW is fixed. The consoles support the AVX VEX prefix encoding for SSE instructions - you should be using it. It's free performance. It enables three address form for most instructions while keeping them 128-bit wide.

At Insomniac Games we still need to target SSE2 for certain PC builds of our engine. One example are the dedicated servers for Sunset Overdrive that run on a cloud service that only offers SSE3 instructions.
After learning about this the hard way we decided to emulate the newer instructions we were using. It turns out that most of them can be emulated reasonably entirely in SIMD registers without having to drop out to scalar code.
So if you need to target more than one feature level I'd recommend establishing a header with intrinsic wrappers early and implement emulation fallbacks as necessary for older hardware. This worked well for us and we spent perhaps a few days to fix up the dedicated server configuration.

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance
- Still need SSE2/3 compatibility for PC builds
  - Tools (and games) running on older PCs
  - Dedicated cloud servers with ancient SSE support

As console developers we target all the features our consoles have, because we know the that the HW is fixed. The consoles support the AVX VEX prefix encoding for SSE instructions - you should be using it. It's free performance. It enables three address form for most instructions while keeping them 128-bit wide.

At Insomniac Games we still need to target SSE2 for certain PC builds of our engine. One example are the dedicated servers for Sunset Overdrive that run on a cloud service that only offers SSE3 instructions.
After learning about this the hard way we decided to emulate the newer instructions we were using. It turns out that most of them can be emulated reasonably entirely in SIMD registers without having to drop out to scalar code.
So if you need to target more than one feature level I'd recommend establishing a header with intrinsic wrappers early and implement emulation fallbacks as necessary for older hardware. This worked well for us and we spent perhaps a few days to fix up the dedicated server configuration.

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance
- Still need SSE2/3 compatibility for PC builds
  - Tools (and games) running on older PCs
  - Dedicated cloud servers with ancient SSE support
- Straightforward to emulate most SSE4+ insns
  - Establish wrappers early for cross-platform projects

As console developers we target all the features our consoles have, because we know the that the HW is fixed. The consoles support the AVX VEX prefix encoding for SSE instructions - you should be using it. It's free performance. It enables three address form for most instructions while keeping them 128-bit wide.

At Insomniac Games we still need to target SSE2 for certain PC builds of our engine. One example are the dedicated servers for Sunset Overdrive that run on a cloud service that only offers SSE3 instructions.
After learning about this the hard way we decided to emulate the newer instructions we were using. It turns out that most of them can be emulated reasonably entirely in SIMD registers without having to drop out to scalar code.
So if you need to target more than one feature level I'd recommend establishing a header with intrinsic wrappers early and implement emulation fallbacks as necessary for older hardware. This worked well for us and we spent perhaps a few days to fix up the dedicated server configuration.

Data Layout Recap

- Two basic choices
  - AOS - Array of Structures
  - SOA - Structure of Arrays
  - Hybrid layouts possible

I want to spend a few minutes talking about data layout issues, even if this is probably repetition for a lot of you. Data layout is the single most important issue to consider before starting to write SIMD code. SIMD doesn't do any good when the data layout is a bad fit as the program will tend be bound by cache misses and wasted memory traffic.

We have two basic choices - Array of Structures (AOS) and Structure of Arrays (SOA). We can also combine these into any number of hybrid layouts.

If you're not familiar with these terms, whenever you're writing a C++ struct or class, you're designing AOS data. All the attributes for a struct are laid out sequentially in memory. This is usually not ideal for SIMD programming, and much of time converting a system is spent on changing layout choices.

Let's look at these real quick so we're all on the same page.

Data Layout Recap

- Two basic choices
  - AOS - Array of Structures
  - SOA - Structure of Arrays
  - Hybrid layouts possible
- Most scalar code tends to be AOS
  - C++ structs and classes make that design choice implicitly
  - Clashes with desire to use SIMD instructions
  - This is probably 75% of the work to fix/compensate for

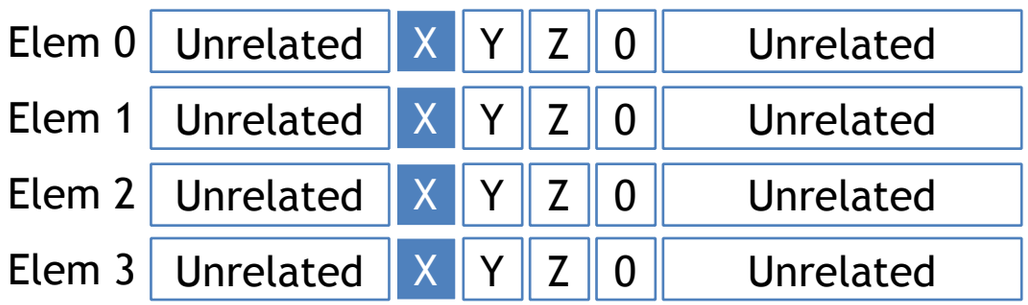
I want to spend a few minutes talking about data layout issues, even if this is probably repetition for a lot of you. Data layout is the single most important issue to consider before starting to write SIMD code. SIMD doesn't do any good when the data layout is a bad fit as the program will tend be bound by cache misses and wasted memory traffic.

We have two basic choices - Array of Structures (AOS) and Structure of Arrays (SOA). We can also combine these into any number of hybrid layouts.

If you're not familiar with these terms, whenever you're writing a C++ struct or class, you're designing AOS data. All the attributes for a struct are laid out sequentially in memory. This is usually not ideal for SIMD programming, and much of time converting a system is spent on changing layout choices.

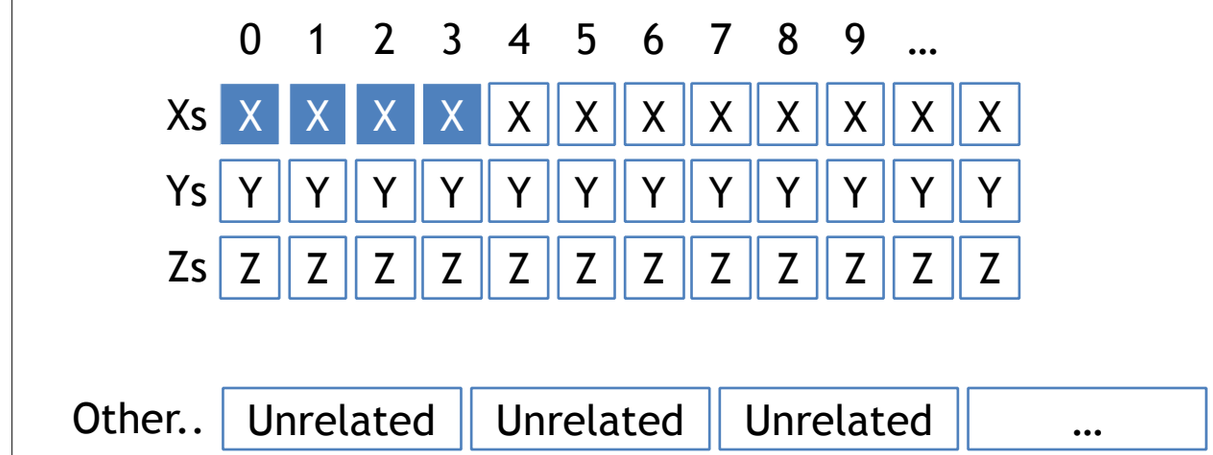
Let's look at these real quick so we're all on the same page.

AOS Data



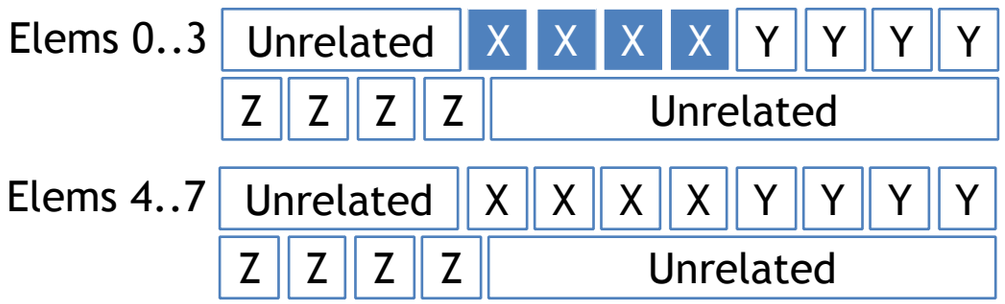
AOS data is laid out like this - if we imagine some array of structures, we need to visit distinct places in memory to gather up four X values. As I mentioned - this is what you get with a basic struct or class that groups attributes together.

SOA Data



In an SOA arrangement, all attributes of a certain type are stored together in their own array. This means that the identity of an object is split into multiple arrays. This can be confusing at first, but it has fantastic benefits for SIMD programming. As you can see, we can easily load 4 X, Y or Z with a single instruction. With AVX you can naturally extend the code to process 8 at a time.

Hybrid: Tiled storage (x4)



Finally in a hybrid scheme, we can make mini-SOA arrays of say 4 elements. Each such group has 4 Xs, 4 Ys and so on. This is a tradeoff between the two others layout choices. It can sometimes be a good tool in its own right, but it is often a sign of a compromise between scalar and SIMD efficiency.