EFFECTIVE SIMD FOR MODERN ARCHITECTURES

With Manu of Brisbane

What is SIMD?

SIMD, or ‘Scottish Index of Multiple Deprivation’…

…erm, thanks google!

Rather, Single Instruction Multiple Data, is a technology present in many modern CPU architectures which offers parallel operations on short arrays of data.

That sounds cool!

Note: I was googling if ‘Multiple’ or ‘Many’

What is SIMD?

In practise, say we have iterative code like this:

```
int data[100];

for(int i = 0; i < data.length; ++i)
{
  data[i] += 10;
}
```

SIMD architecture allows us to effectively perform:

```
for(int i = 0; i < data.length; i += 4)
{
  data[i + 0] += 10; // these operations are performed simultaneously!
  data[i + 1] += 10;
  data[i + 2] += 10;
  data[i + 3] += 10;
}
```

Effectively quadrupling our throughput! *

(*) Nothing is ever so simple!

What is SIMD?

Using SIMD hardware is a low-level optimisation, and best applies to tight work loops.

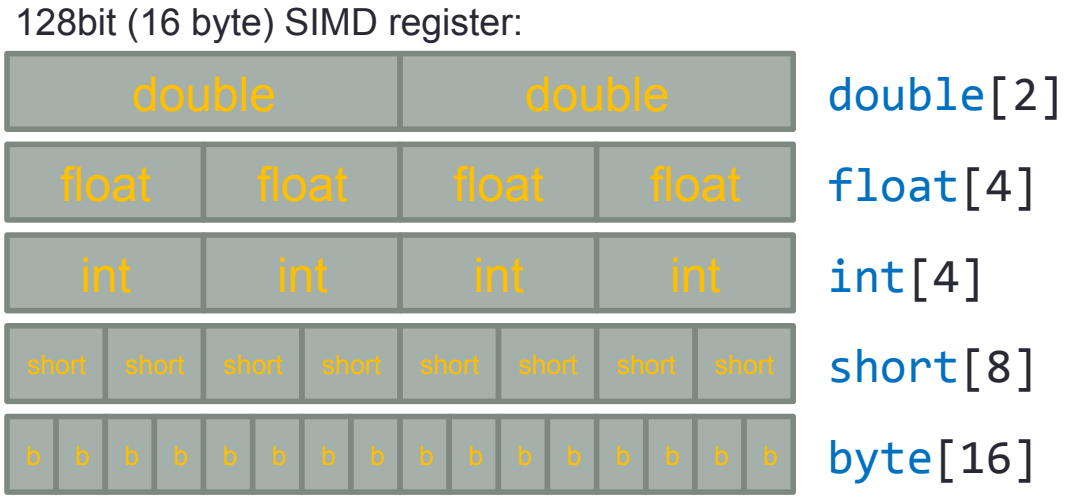
SIMD may improve performance in a few important ways:

- Reduce instructions issued
- Reduce loop Iterations
- Increase data bandwidth
- Reduce program code size
- Specialised instructions to perform complex tasks

How do we use it?

Usually accessed via a special set of instructions, and specialised multi-element data registers.

SIMD registers are often able to be reinterpreted as various different types, for instance:



* Interesting to note; SIMD registers may contain float *or* integer data

How do we use it?

In D, we access this hardware via the `__vector()` intrinsic:

```
import core.simd;

__vector(float[4]) myVector;
```

`core.simd` also provides a suite of helpful aliases:

```
alias __vector(double[2])  double2;
alias __vector(int[4])     int4;
alias __vector(byte[16])   byte16;
// etc
```

It's important to note that these are intrinsic types that map to the hardware directly.

`type[width]` pairs which are not supported by the target architecture produce compile errors.

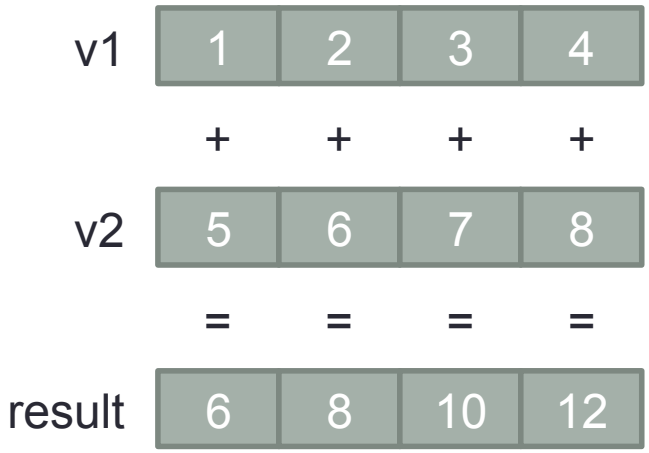
How do we use it?

Standard set of operators can be used with SIMD types.

Perform operations element-wise:

```
import core.simd;

float4 v1 = [1,2,3,4];
float4 v2 = [5,6,7,8];
float4 result = v1 + v2;
```



In this way, they work just like regular D array operations.

How do we use it?

Let's consider our previous example:

```
int data[100];

foreach(i; 0 .. data.length)
{
  data[i] += 10;
}
```

Applying SIMD logic, we get:

```
int4 data[100 / int4.length];
int4 tens = [10, 10, 10, 10]; // we need a vector with 10's

foreach(i; 0 .. data.length)
{
  data[i] += tens;
}
```

- Only 25 iterations
- Improved bandwidth

How do we use it?

SIMD can greatly accelerate iterative tasks; scalar workloads processed in parallel.

But another popular use is to consider the elements as components of a 4d vector.

Though functionally identical, this perspective applies better to linear algebra.

How do we use it?

Linear algebra is popular in various fields of computing:

- Image processing
- Geometry processing/rendering
- Simulation/physics

I'll focus on this perspective for the majority of this talk.

Limitations

So we understand the basic technology…

To make good efficient use of the hardware, we need to understand some restrictions and limitations.

FAQ: Why require the `__vector()` intrinsic? Why not work with standard D arrays?

A: SIMD hardware is not as flexible as regular arrays.

`__vector()` enforces rules associated with the hardware.

Reserves the right to produce compile errors for unsupported `type[width]` combinations.

Limitations

Limitations are non-uniform across different architectures.

But a set of best-practises can be applied that will:

- Get good performance from all architectures
- Won't hinder any particular architecture
- Maximise portability

Let's look at some of these…

Limitations

1. Test all target architectures.

Not a best-practise as such, but a requirement:

Programmers will need to test their code on all target architectures!

Typical approach is to provide a fall-back routine using regular D code and only enable SIMD for tested hardware.

Discussions about a ‘portable’ SIMD library ongoing…

Limitations

2. Data alignment to register width.

The `__vector()` intrinsic will force alignment.

- Take care when laying out structures and memory.
- Avoid wasteful padding.

Let's consider an example…

Limitations

2. Data alignment to register width.

```
struct Dude
{
  float4   position;
  int      health;
  float    velocity;
  void[8]  __padding;
}
```

Looks innocent, one might imagine it is 4 + 16 + 4 = 24 bytes.

But, due to alignment requirements, the struct is internally rewritten.

Turns out the struct is actually 4 + 12 + 16 + 4 + 12 = 48 bytes!

We can improve the situation by reordering the structure.

16 + 4 + 4 + 8 = 32 bytes, better, but there's still 8 bytes waste…

These 8 bytes can't be eliminated, so you might as well use them!

- Put some other useful data, or cached values in that space?

Limitations

3. SIMD vectors don't interact with scalar types.

It seems natural to express scalar values with scalar types, but SIMD types only interact efficiently with other SIMD types.

Don’t:

```
float  scale = 2;
float4 v;

v = v * scale; // *** vector/float multiply! ***
```

Do:

```
float4 scale = [ 2, 2, 2, 2 ];
float4 v;

v = v * scale;
```

Move conversions as far outside loops as possible.

Limitations

4. Individual elements are not randomly accessible; must be digested in parallel!

Consider:

```
float4 v;
v.array[1] = 0; // don't do this unless you know what you're doing!
```

The array indexing syntax is removed from `__vector()`'s.

Perform operations on all elements in parallel

- You may need to get quite creative!

If random access is absolutely required, use the `.array` property

- Expect serious performance penalties!

`std.simd`

So now we have a pretty good understanding of SIMD.

How can we easily put this all to use, while considering the limitations presented?

Enter `std.simd`, the goals of which are to:

- Provide a complete set of higher-level functions
- Map efficiently to the underlying architecture
- API encourages best practises
- Fill any gaps in hardware support
  - But only if it can be done efficiently!

std.simd

Let's have a quick look at the sort of functions available…

Basic maths:

- `abs, neg, add, sub`
- `mul, min, max`

Operators are available, but these functions can increase portability.

Bitwise operations:

- `comp, or, xor, and`
- `nor, nand, andNot`

Note the `nor`, `nand`, `andNot` functions. These can take advantage of a free compliment on some architectures!

std.simd

Compound maths:

- `madd, msub, nmadd, nmsub`

Many architectures offer fused multiply-accumulate instructions, which can greatly improve efficiency. Use these wherever applicable!

Complex maths:

- `rcp, div, sqrt, rsqrt`

`sin`/`cos`/`pow` will likely appear in the future.

Linear algebra:

- `dot, cross, magnitude, normalise`

Typical suite of linear algebra functions.

std.simd

Fast estimates:

- `rcpEst, divEst, sqrtEst, rsqrtEst`
- `magEst, normEst`

Attempt to be as fast as possible. Precision may be reduced, and may differ slightly between architectures.

Element permutation:

- `permute/swizzle, interleave, broadcast`
- `unaligned load`

Functions that rearrange elements within vectors.

Shifts:

Comprehensive set of functions to shift/rotate elements.

std.simd

Type conversion:

- Type conversion/casting
- Data packing/unpacking

Comparisons:

- Full suite of comparisons

Can produce bit-masks, or boolean ‘any’/‘all’ logic.

Branchless selection:

Branchless element-wise selection based on comparison, or predicate.

And much more!

std.simd

std.simd makes great effort to provide a uniform API, but if the target architecture can't perform an operation efficiently, it will produce a compile error.

This is deliberate; programmers are better to stop and reconsider the problem than un-knowingly use a slow emulation of a function they require.

Users may write vector libraries above std.simd, filling in remaining gaps with compromises they are comfortable with in their project.

Getting the best performance

Using these functions alone will not yield the best results.

Implementing good SIMD code will often just result in a different bottleneck holding you up.

Only by eliminating *all* bottlenecks will you truly unleash your code.

Let's look at some common associated performance hazards you should also be aware of…

Getting the best performance

1. Minimise memory access.

Memory is slow, particularly on embedded architectures.

- Consider the allocation of your variables in registers
- Pre-load outside loops wherever possible
- Never cast between register types (int/float/simd)
  - These casts usually transfer registers via memory

Getting the best performance

2. The ABI supports vector arguments, use them.

Always pass vectors by value.

This way, they will be passed in a register and avoid accessing the stack. (more unnecessary memory access!)

People are often in the habit of passing their non-SIMD vector classes by reference. Don’t transfer that habit.

Getting the best performance

3. Use ‘leaf’ functions where possible.

‘Leaf functions’ are functions that don’t allocate a stack frame at all. (again, eliminate memory access)

There are some criteria to qualify as a leaf function:

- Can't call any sub-functions (unless they inline)
- Can't receive any non-primitive arguments by value
- May not return a structure by value
- Don’t cast between register types (int/float/simd)
- Use few enough locals that fit in the processors registers

This results in good gains, especially within hot loops!

Getting the best performance

4. Only cast SIMD types

If you need to cast float<->int, do it in SIMD.

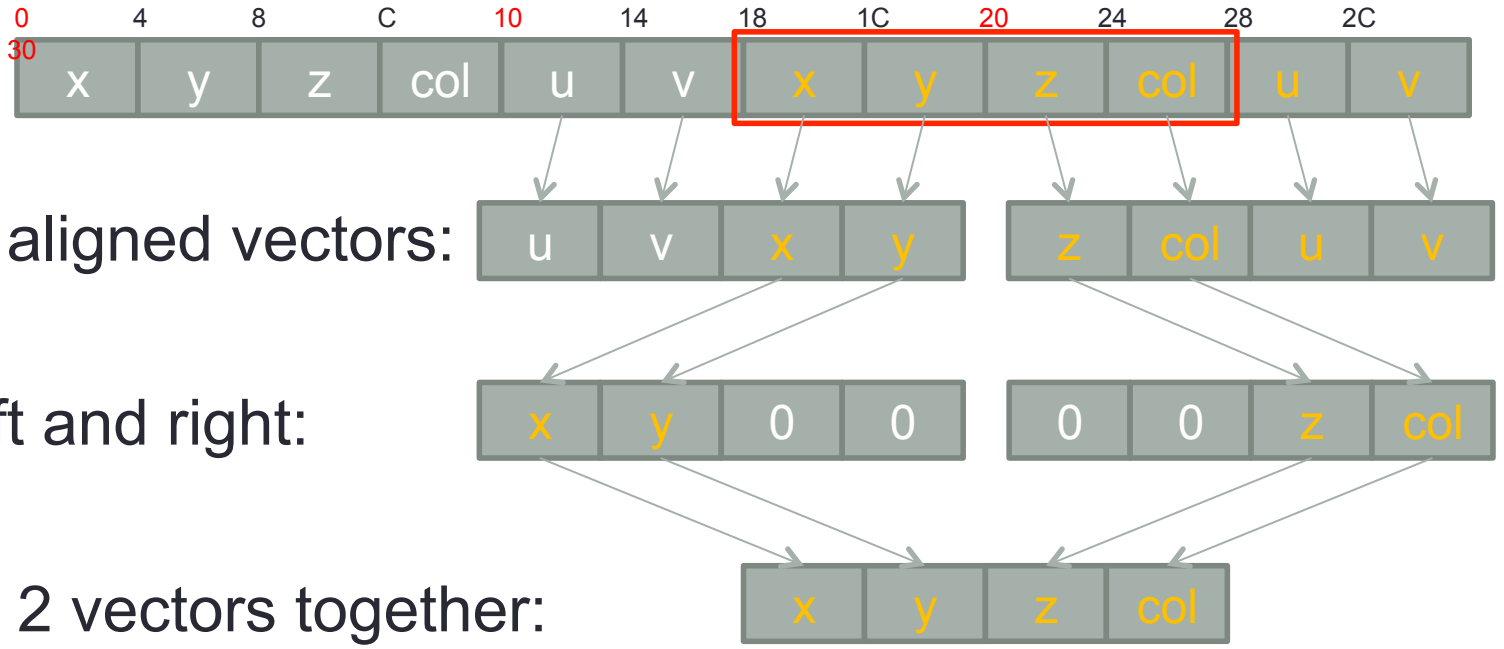
Typically, a cast requires swapping register types, which passes the value through memory.

SIMD registers can hold both int and float data, so they can cast without accessing memory!

Getting the best performance

5. Avoid unaligned loads

Unaligned loads are a slow process…



Getting the best performance

6.a. Pipelining

SIMD operations tend to have higher latency, and increase dependence on past results vs normal code.

Unrolling loops a little can help fill the pipeline with work.

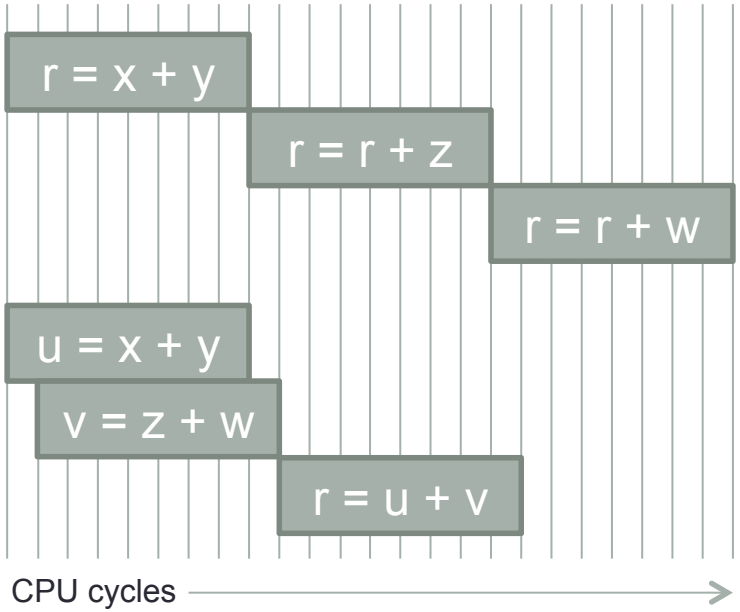
Getting the best performance

6.b. Pipelining

Reduce operation dependencies.

```
r = x + y + z + w;
```

```
r = (x + y) + (z + w);
```



The optimiser will help where it can, but don’t rely on it!

Getting the best performance

7. Bandwidth

Aggressively optimised code will almost always become bandwidth limited.

Tips to improve memory bandwidth:

- Pack your data where possible
  - Preferably into 16-byte blocks (single, aligned load)
  - Unpacking is usually faster than fetching memory
- Be aware of the d-cache
  - Access memory linearly
  - Keep random-access memory close (in pools?)
- Experiment with prefetching?

In summary

To recap, what have we learned?

- Test all target architectures, provide standard D fallback code
- Pay careful attention to data alignment
- Avoid accessing individual elements
- Avoid casting between int/float/simd
- Use compound operations wherever able (madd, nor, andNot, etc)
- Avoid memory (args by value, leaf functions, don't cast)
- Avoid unpredictable 'if's, prefer select-style logic
- Minimise data structures (multiples of 16 bytes)
- Pay attention to pipeline; unroll loops, concurrent tasks where possible

Congratulations!

You are now an expert at SIMD programming.

Questions?