Programming with `ispc`

Troels Henriksen

DIKU
University of Copenhagen

Agenda

SIMD on CPU

Auto-vectorisation

Programming in ISPC

ISPC Performance

SIMD on CPU

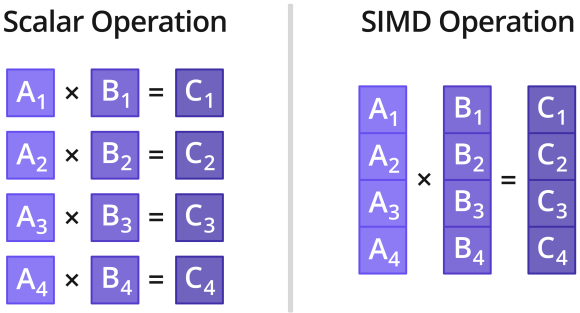
Auto-vectorisation

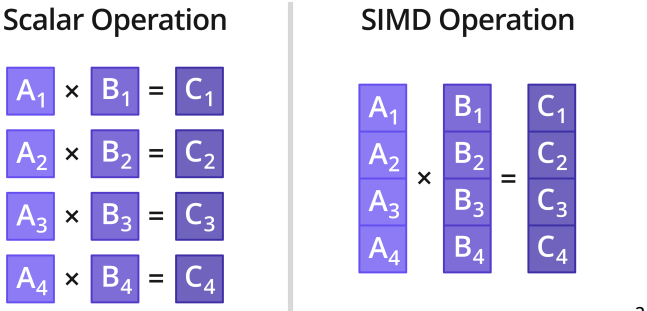
Programming in ISPC

ISPC Performance

What is SIMD?

- **Single Instruction Multiple Data**.
- Fixed-size *vector registers* that contain multiple values.
- *Vector instructions* that operate on vector registers.
  - A form of explicit ILP where instruction is the same.
  - If vector register contains four `floats`, then single SIMD instruction can replace four scalar instructions.
- Usually an extension to the scalar instruction set.





x86 example

The registers `xmm0`–`xmm15` are 128 bits each, and operated on with instructions like ADDSS, SUBSS, MULSS.

- Instead of `add eax ebx` use e.g. `addss xmm0 xmm1`.
- Follows all the normal assembly rules.

[2] `https://medium.com/wasmer/webassembly-and-simd-13badb9bf1a8`

When is SIMD useful?

- In our ontology, SIMD is *first-order flat data parallelism* on *vectors of static size*.
- When program is not bottlenecked by IO or memory traffic.
  - Although bulk memory reads can sometimes be vectorised.

SIMD is cheap

- It's already part of the CPU and uses the same memory.
- Using an SIMD instruction has zero latency.
- Compare to GPUs, which have huge overhead.

- **SIMD is not multithreading**
  - Both techniques can be used simultaneously.
  - Will only focus on SIMD for this lecture.
- Speedup limited to SIMD vector size (in practice usually 4 or 8).
  - Often much less.

Applying vectorisation

- In principle, we can use SIMD instructions whenever independent operations with same operator occur

```
float x = a + b;
float y = b + c;
float z = d + e;
float v = e + f;
```

- Compilers *do* try to do this…
- … but vectorisation mostly shines when doing bulk data-parallel operations

```
for (int i = 0; i < n; i++) {
  xs[i] *= 2;
}
```

Simple vectorisation

```
for (int i = 0; i < n; i++) {
  xs[i] *= 2;
}
```

Parallel?

Simple vectorisation

```
for (int i = 0; i < n; i++) {
  xs[i] *= 2;
}
```

Parallel?

Conceptually vectorise by *strip-mining* the loop into chunks corresponding to the vector size (e.g. 4):

```
for (int j = 0; j < n; j += 4) {
  for (int i = j; i < j+4; i++) {
    xs[i] *= 2;
  }
}
```

- Inner loop can be replaced with vector instructions.
- Separate loop for any remaining < 4 iterations.

```
for (int j = 0; j < n; j += 4)
  for (int i = j; i < j+4; i++)
    xs[i] *= 2;
```

```
# eax: j
# ebx: i*4
# ecx: n
# rdi: xs
loop:
  movdqu  xmm0, xmmword ptr [rdi + ebx]
  paddd   xmm0, xmm0
  movdqu  xmmword ptr [rdi + ebx], xmm0
  add     eax, 4
  add     ebx, 16
  cmp     eax, ecx
  jl      .loop
```

Vectorising functions with straight-line code is easy

Suppose to want to vectorise this single function to operate on a vector of values:

```
int f(int x, int y) {
  return (x + y) / 2;
}
```

Scalar version handles one x/y pair at a time.

```
add edi, esi
shr exi, 31
```

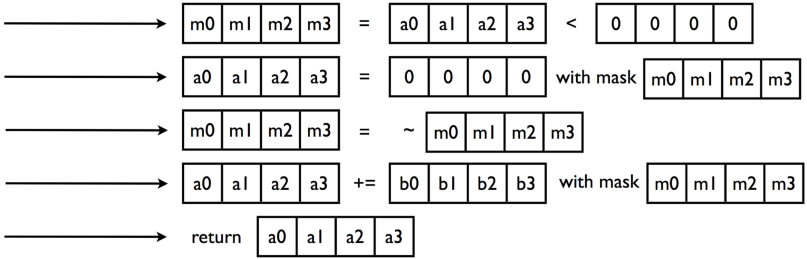
Vector version handles four x/y pairs at a time.

```
paddd   xmm1, xmm0
psrad   xmm1, 1
```

What about control flow?

Vectorising scalar function with control flow[3]

```
int f(int a, int b) {
  if (a < 0)
    a = 0;
  else
    a += b;
  return a;
}
```



- **Masking**, similar to NumPy.
- Control flow turned into data flow (arithmetic).
- Doing this by hand gets very tiresome and is error-prone.
- Vector capabilities differ a lot between architectures.

[3] Example from `ispc` paper.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

2007 **SSE4**: 54 instructions, particularly masking.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

2007 **SSE4**: 54 instructions, particularly masking.

2011 **AVX**: 256-bit registers (YMM0–YMM15). 86 floating-point instructions.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

2007 **SSE4**: 54 instructions, particularly masking.

2011 **AVX**: 256-bit registers (YMM0–YMM15). 86 floating-point instructions.

2013 **AVX2**: 137 new instructions, now also with integers.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

2007 **SSE4**: 54 instructions, particularly masking.

2011 **AVX**: 256-bit registers (YMM0–YMM15). 86 floating-point instructions.

2013 **AVX2**: 137 new instructions, now also with integers.

2016 **AVX-512**: 512-bit registers (ZMM0–ZMM31), and a *zoo* of layers, not expected to be in all processors.

SIMD extensions added to x86 over time

1997 **MMX**: New 64-bit registers (MM0—MM7), 57 new integer instructions.

1999 **SSE**: New 128-bit registers (XMM0—XMM7) representing four 32-bit floats each. 70 instructions.

2000 **SSE2**: Adds another 144 instructions, particularly for integer operations. AMD64 eventually adds more registers (XMM8—XMM15).

2004 **SSE3**: 13 new instructions, in particular for "horizontal" operations that work within a vector.

2006 **SSSE3**: 16 new instructions.

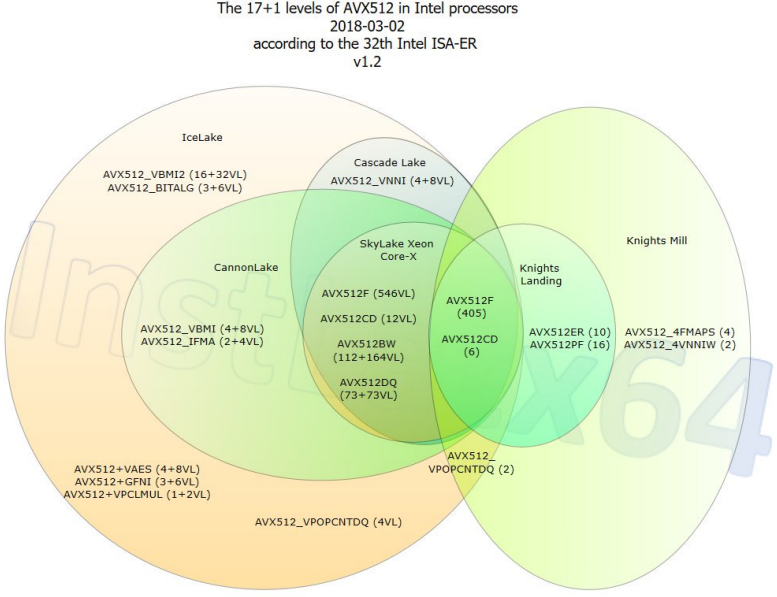
2007 **SSE4**: 54 instructions, particularly masking.

2011 **AVX**: 256-bit registers (YMM0–YMM15). 86 floating-point instructions.

2013 **AVX2**: 137 new instructions, now also with integers.

2016 **AVX-512**: 512-bit registers (ZMM0–ZMM31), and a *zoo* of layers, not expected to be in all processors.

2023 **AVX10**: 512 bit registers no longer guaranteed, some new instructions.



https://twitter.com/InstLatX64/status/969560033922035713

SIMD on CPU

Auto-vectorisation

Programming in ISPC

ISPC Performance

Auto-vectorisation

Write ordinary C code and hope that a sufficiently smart compiler can figure out what you mean and make use of whatever instruction set is supported on your computer.

Good: you don't have to learn anything new!

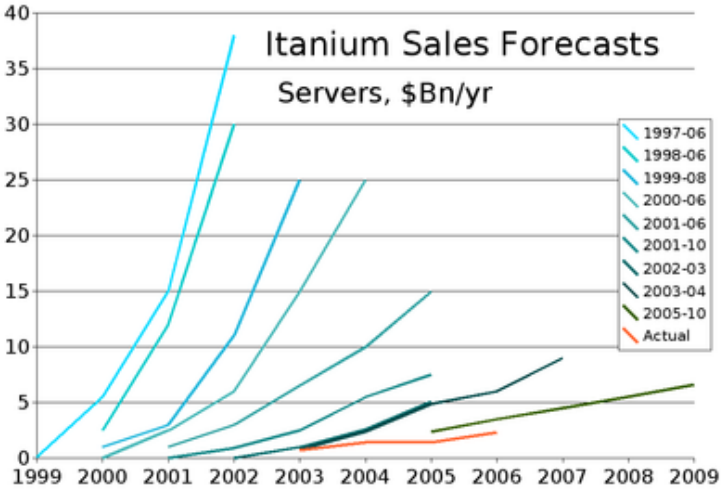
Auto-vectorisation

Write ordinary C code and hope that a sufficiently smart compiler can figure out what you mean and make use of whatever instruction set is supported on your computer.

Good: you don't have to learn anything new!

Bad: does it work?

What happened last time we depended on Sufficiently Smart Compilers?



Auto-vectorisation, example A

```
void A(int n, int *xs) {
  for (int i = 0;
       i < n;
       i++) {
    xs[i] *= 2;
  }
}

$ gcc -O3 A.c -S
```

Auto-vectorisation, example A

```
void A(int n, int *xs) {
  for (int i = 0;
       i < n;
       i++) {
    xs[i] *= 2;
  }
}

$ gcc -O3 A.c -S
```

```
.L4:
  movdqu  xmm0, XMMWORD PTR [rax]
  add     rax, 16
  pslld   xmm0, 1
  movups  XMMWORD PTR [rax-16], xmm0
  cmp     rax, rcx
  jne     .L4
```

Auto-vectorisation, example A

```
void A(int n, int *xs) {
  for (int i = 0;
       i < n;
       i++) {
    xs[i] *= 2;
  }
}
```

```
$ gcc -O3 A.c -S
```

```
.L4:
  movdqu  xmm0, XMMWORD PTR [rax]
  add     rax, 16
  pslld   xmm0, 1
  movups  XMMWORD PTR [rax-16], xmm0
  cmp     rax, rcx
  jne     .L4
```

Vectorised!

Auto-vectorisation, example B

```
int B(int n, int *xs) {
  int res = 0;
  for (int i = 0;
       i < n;
       i++) {
    res += xs[i];
  }
  return res;
}
```

Auto-vectorisation, example B

```
int B(int n, int *xs) {
  int res = 0;
  for (int i = 0;
       i < n;
       i++) {
    res += xs[i];
  }
  return res;
}
```

```
.L4:
  movdqu  xmm2, XMMWORD PTR [rax]
  add     rax, 16
  paddd   xmm0, xmm2
  cmp     rdx, rax
  jne     .L4
  movdqa  xmm1, xmm0
  mov     edx, ecx
  psrldq  xmm1, 8
  and     edx, -4
  paddd   xmm0, xmm1
  movdqa  xmm1, xmm0
  psrldq  xmm1, 4
  paddd   xmm0, xmm1
  movd    eax, xmm0
  test    cl, 3
  je      .L11
```

Auto-vectorisation, example B

```
int B(int n, int *xs) {
  int res = 0;
  for (int i = 0;
       i < n;
       i++) {
    res += xs[i];
  }
  return res;
}
```

Tricky, but **vectorised!**

```
.L4:
  movdqu  xmm2, XMMWORD PTR [rax]
  add     rax, 16
  paddd   xmm0, xmm2
  cmp     rdx, rax
  jne     .L4
  movdqa  xmm1, xmm0
  mov     edx, ecx
  psrldq  xmm1, 8
  and     edx, -4
  paddd   xmm0, xmm1
  movdqa  xmm1, xmm0
  psrldq  xmm1, 4
  paddd   xmm0, xmm1
  movd    eax, xmm0
  test    cl, 3
  je      .L11
```

Auto-vectorisation, example C

```
void C(int n,
       int *ys,
       int *xs) {
  int acc = 0;
  for (int i = 0;
       i < n;
       i++) {
    acc += xs[i];
    ys[i] = acc;
  }
}
```

Auto-vectorisation, example C

```
void C(int n,
       int *ys,
       int *xs) {
  int acc = 0;
  for (int i = 0;
       i < n;
       i++) {
    acc += xs[i];
    ys[i] = acc;
  }
}
```

```
.L3:
  add     ecx, DWORD PTR [rdx+rax]
  mov     DWORD PTR [rsi+rax], ecx
  add     rax, 4
  cmp     rdi, rax
  jne     .L3
```

Auto-vectorisation, example C

```
void C(int n,
       int *ys,
       int *xs) {
  int acc = 0;
  for (int i = 0;
       i < n;
       i++) {
    acc += xs[i];
    ys[i] = acc;
  }
}
```

```
.L3:
  add     ecx, DWORD PTR [rdx+rax]
  mov     DWORD PTR [rsi+rax], ecx
  add     rax, 4
  cmp     rdi, rax
  jne     .L3
```

Not vectorised.

Vectorising `powi()`

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a;
    return r;
}
```

Vectorising `powi()`

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a;
    return r;
}
```

Two ways of vectorising

1. Make a single call `powi(a,b)` faster by trying to do multiple iterations of the `while` loop in parallel (probably infeasible).

Vectorising `powi()`

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a;
    return r;
}
```

Two ways of vectorising

1. Make a single call `powi(a,b)` faster by trying to do multiple iterations of the `while` loop in parallel (probably infeasible).
2. Make multiple *instances* of `powi(a,b)` faster:

```
for (int i = 0; i < n; i++) {
  cs[i] = powi(as[i], bs[i]);
}
```

- (1) much harder, (2) often what we want.
- **How would a compiler know?**

Vectorising `powi()` with option (2)

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a; // AVX2 assembly for this loop below
    return r;
}
```

Vectorising `powi()` with option (2)

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a; // AVX2 assembly for this loop below
    return r;
}
```

```
.LBB1_3:
  vpaddd     ymm8, ymm1, ymm5        # Decrement b.
  vblendvps  ymm1, ymm1, ymm8, ymm7  # Only active lanes.
  vmulps     ymm7, ymm3, ymm0        # r * a.
  vblendvps  ymm3, ymm3, ymm7, ymm6  # Only active lanes.
  vpcmpeqd   ymm8, ymm1, ymm4        # Compare b with zero.
  vmovaps    ymm7, ymm6              # Compute which lanes
  vpandn     ymm6, ymm8, ymm6        # still active.
  vpand      ymm8, ymm6, ymm2        # AND with current.
  vmovmskps  eax, ymm8               # Any lanes active?
  test       eax, eax                # ...
  jne        .LBB1_3                 # Redo if so.
```

A C compiler will not do this.

Motivation for `ispc`

- Around 2010, Matt Pharr from Intel's graphics group had too much time and too much trouble with extant auto-vectorisers.
- **Motivating problem: graphics**
  - Often embarassingly parallel:

```
for (int j = 0; j < height; j++) {
  for (int i = 0; i < width; i++) {
    // compute pixel i,j
  }
}
```

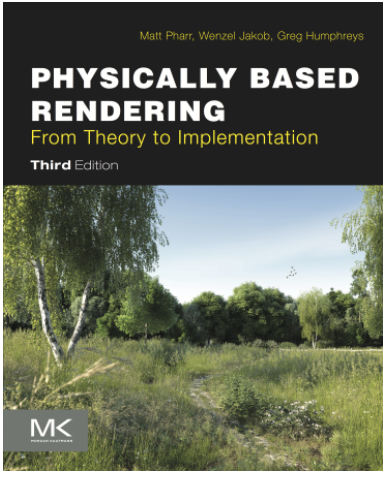
  - Often lots of control flow in the loop body.
  - If we could have one vector lane compute one pixel, then we could compute 4/8/16 pixels simultaneously.
  - Even today, auto-vectorisers choke on anything non-trivial (ask Cosmin about stencils).

Quotes from Matt Pharr

1. *They'd inform the graphics folks that they'd improved their auto-vectorizer in response to our requests and that it did everything we had asked for.*
2. *We'd try it and find that though it was better, boy was it easy to write code that wasn't actually compiled to vector code—it'd fail unpredictably.*
3. *We'd give them failing cases, a few months would would pass and they'd inform us that the latest version solved the problem.*

They tried to patch things up at first but eventually they threw up their hands and came up with `#pragma simd`*, which would disable the "is it safe to vectorize this" checks in the auto-vectorizer and vectorize the following loop no matter what. (Once a* `#pragma` *is proposed to solve a hard problem, you know things aren't in a good place.)*

As an aside, Matt Pharr is likely the only programming language designer to win an Academy Award



Winner of a 2014 Scientific and Technical Academy Award

Auto-vectorisation is not a programming model

- A compiler can easily tell you *what* fails to vectorise.
- Not so clear *why*.
- You need to become an expert in the compiler and tickle your code in just the right way.
- **This is a horrible way to program**.

And then they release a new compiler with subtly different behaviour…

> *With a proper programming model, then the programmer learns the model (which is hopefully fairly clean), one or more compilers implement it, the generated code is predictable (no performance cliffs), and everyone's happy.*

Single Program Multiple Data (SPMD)

- Essentially the model also used by CUDA.
- Multiple *instances* of program is running.
- Group of running instances is called a *gang*.

```
void inc(int xs[], int ys[], int result[]) {
  result[programIndex] =
    (xs[programIndex] + ys[programIndex]) / 2;
}
```

- In principle, one instance per array element.
- On GPU, *many* instances are useful to allow latency hiding.
- On CPU, rarely more than twice the SIMD width.

SIMD on CPU

Auto-vectorisation

Programming in ISPC

ISPC Performance

Intel SPMD Program Compiler

> `ispc` *is a compiler for a variant of the C programming language, with extensions for "single program, multiple data" (SPMD) programming.*

`ispc is`

- a variant of C—much scalar C is also valid `ispc`.
- a *low-level parallel language*.
- *not guaranteed to be deterministic*.
- a compiler with only few *compiler optimisations*.

Intel SPMD Program Compiler

> `ispc` *is a compiler for a variant of the C programming language, with extensions for "single program, multiple data" (SPMD) programming.*

`ispc is`

- a variant of C—much scalar C is also valid `ispc`.
- a *low-level parallel language*.
- *not guaranteed to be deterministic*.
- a compiler with only few *compiler optimisations*.

What makes it relevant for our course is that it is a good example of using *the right programming model*.

Disclaimer

I am not an `ispc` expert.

I am merely an enthusiast, and I think its design is a masterpiece of simplicity, worthy of study.

Most of what I know is from

1. *ispc: A SPMD Compiler for High-Performance CPU Programming* (paper)
2. *The story of* `ispc` (series of blog posts by Matt Pharr)[4]
3. The User's Guide[5]

[4] `https://pharr.org/matt/blog/2018/04/18/ispc-origins.html`
[5] `https://ispc.github.io/ispc.html`

ispc SPMD gang abstraction using SIMD instructions

SPMD implementation in ispc

- Call to ispc function spawns gang of program instances.
- All instances done when function returns.
- Gang size is SIMD width or a small multiple.
- Lockstep execution one ”line” at a time.
- Each instance gets a copy of each non-uniform variable.

```
export void f(uniform int n, uniform float as[]) {
  for (int i = programIndex;
       i < n;
       i += programCount) {
    float a = as[i];
    // All instances finish previous line
    // before continuing
    as[i] = a * 2;
  }
}
```

Abstraction versus implementation

- **Model is SPMD**
  - Running a gang spawns `programCount` logical execution flows.
  - Each has distinct value of `programIndex`.
  - (PMPH students: sound familiar?).
  - We program against this **abstraction**.
- **Implementation is SIMD**
  - Variables mapped to slots in SIMD registers.
  - SIMD instructions carry out work of gang.
  - Control flow implemented by masking or arithmetic.
- **Abstraction is intentionally leaky**
  - `uniform` values and cross-instance communication (later) let us mess up.
  - `ispc` concerned with real-world performance, not purity.
  - An expert programmer can often predict the SIMD instructions generated by compiler.

In principle, `ispc` could also generate GPU code! **Good idea?**

Previous example in ispc

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a;
    return r;
}

export void f(uniform int n,
              uniform float as[],
              uniform int bs[],
              uniform float cs[]) {
  for (int i = programIndex;
       i < n;
       i += programCount) {
    cs[i] = powi(as[i], bs[i]);
  }
}
```

Previous example in ispc (better)

```
float powi(float a, int b) {
    float r = 1;
    while (b--) r *= a;
    return r;
}

export void f(uniform int n,
              uniform float as[],
              uniform int bs[],
              uniform float cs[]) {
  foreach (i = 0 ... n) {
    cs[i] = powi(as[i], bs[i]);
  }
}
```

To inspect generated assembly

```
$ ispc powi.isc --target=avx2 --emit-asm -o powi.s
```

To inspect generated assembly

```
$ ispc powi.isc --target=avx2 --emit-asm -o powi.s
```

To generate object file and header file

```
$ ispc powi.isc --target=avx2 -o powi.o -h powi.h
```

Then from some C program we can say

```
#include "powi.h"

...

float *as = malloc(1024*sizeof(float));
int *bs = malloc(1024*sizeof(int));
float *cs = malloc(1024*sizeof(float));
... initialize as, bs, cs ...
f(1024, as, bs, cs);
```

Mandelbrot in C

```
static inline int mandel(float c_re, float c_im, int count) {
  float z_re = c_re, z_im = c_im;
  int i;
  for (i = 0; i < count; ++i) {
    if (z_re * z_re + z_im * z_im > 4.)
      break;
    float new_re = z_re*z_re - z_im*z_im;
    float new_im = 2.f * z_re * z_im;
    z_re = c_re + new_re;
    z_im = c_im + new_im;
  }
  return i;
}
```

Also valid `ispc`!

Mandelbrot in C

```
void mandelbrot_c(float x0, float y0,
                  float x1, float y1,
                  int width, int height,
                  int maxIterations,
                  int output[]) {
  float dx = (x1 - x0) / width, dy = (y1 - y0) / height;
  for (int j = 0; j < height; j++) {
    for (int i = 0; i < width; i++) {
      float x = x0 + i * dx, y = y0 + j * dy;
      int index = j * width + i;
      output[index] = mandel(x, y, maxIterations);
    }
  }
}
```

Mandelbrot in ispc

```
export void mandelbrot_ispc
    (uniform float x0, uniform float y0,
     uniform float x1, uniform float y1,
     uniform int width, uniform int height,
     uniform int maxIterations,
     uniform int output[]) {
  float dx = (x1 - x0) / width, dy = (y1 - y0) / height;
  for (uniform int j = 0; j < height; j++) {
    foreach (i = 0 ... width) {
      float x = x0 + i * dx, y = y0 + j * dy;
      int index = j * width + i;
      output[index] = mandel(x, y, maxIterations);
    }
  }
}
```

Benefits of `uniform`

- A `uniform` variable has same value in all program instances:
  - \+ Can be stored in non-vector register.
  - \+ Takes less space in memory if spilled.
  - \+ CPU can co-issue with vector instructions.
  - \+ Dereferencing `uniform` pointer is load, not vector gather.
  - \+ `uniform` branching avoids masking.
- Unadorned types are `variant` by default:
  - Each program instance has its own copy.
  - `uniform` converts automatically to `variant`.

```
float foo[10];         // 10 floats per program instance
uniform float bar[10]; // 10 floats total
```

uniform and control flow

```
float a = ...;
uniform int b = 0;
if (a == 0) {
    ++b;
    // b is 1
}
else {
    b = 10;
    // b is 10
}
```

- Whether b is 1 or 10 depends on whether any of the values of a in the executing gang were 0.
- Be careful!

SIMD on CPU

Auto-vectorisation

Programming in ISPC

ISPC Performance

Summation in ispc

```
export uniform int sum_ispc(uniform int input[],
                            uniform int n) {
  uniform int sum = 0;
  foreach (i = 0 ... n) {
    sum += reduce_add(input[i]);
  }

  return sum;
}
```

- `reduce_add()` takes varying value and returns `uniform` sum of that value across all of the active program instances.
- Is this correct?
- Is this optimal?

Summation in ispc — better

```
export uniform int sum_ispc(uniform int input[],
                            uniform int n) {
  int sum = 0;
  foreach (i = 0 ... n ) {
    sum += input[i];
  }

  return reduce_add(sum);
}
```

Writing `reduce_add()` ourselves

```
uniform int own_reduce_add(int x) {
  uniform int sums[programCount];
  sums[programIndex] = x;
  if (programIndex == 0) {
    for (uniform int i = 1; i < programCount; i++) {
      sums[0] += sums[i];
    }
  }
  return sums[0];
}
```

Builtin one is probably smarter.

Other functions for cross-instance communication

```
// Returns 'x' provided by program instance 'i'.
uniform int32 extract(int32 x, uniform int i);
```

Other functions for cross-instance communication

```
// Returns 'x' provided by program instance 'i'.
uniform int32 extract(int32 x, uniform int i);

// Returns 'x' provided by program
// instance 'offset' steps away.
int32 rotate(int32 value, uniform int offset)
```

Other functions for cross-instance communication

```
// Returns 'x' provided by program instance 'i'.
uniform int32 extract(int32 x, uniform int i);

// Returns 'x' provided by program
// instance 'offset' steps away.
int32 rotate(int32 value, uniform int offset)

// Exclusive scan of provided 'v's.
int32 exclusive_scan_add(int32 v)
```

1D stencil

```
static inline void relax_c(float *output,
                           float* input,
                           int n) {
  for (int i = 0; i < n; i++) {
    float l, c, r;
    l=input[i == 0 ? n-1 : i-1];
    c=input[i];
    r=input[(i+1) % n];
    output[i] = (l + c + r)/3;
  }
}
```

Runtime for n=1000000: 140μs

Try it with ispc

```
export void relax_ispc_naive(uniform float output[],
                             uniform float input[],
                             uniform int n) {
  foreach (i = 0 ... n) {
    float l, c, r;
    l=input[i == 0 ? n-1 : i-1];
    c=input[i];
    r=input[(i+1) % n];
    output[i] = (l + c + r)/3;
  }
}
```

Try it with ispc

```
export void relax_ispc_naive(uniform float output[],
                             uniform float input[],
                             uniform int n) {
  foreach (i = 0 ... n) {
    float l, c, r;
    l=input[i == 0 ? n-1 : i-1];
    c=input[i];
    r=input[(i+1) % n];
    output[i] = (l + c + r)/3;
  }
}
```

Runtime for n=1000000: 403μs — **what happened?**

ispc compiler output

```
relax.ispc:6:13: Performance Warning: Modulus
        operator with varying types is very inefficient.
    r=input[(i+1) % n];
            ^^^^^^^^^

relax.ispc:4:7: Performance Warning: Gather required
        to load value.
    l=input[i == 0 ? n-1 : i-1];
      ^^^^^^^^^^^^^^^^^^^^^^^^

relax.ispc:6:7: Performance Warning: Gather required
        to load value.
    r=input[(i+1) % n];
      ^^^^^^^^^^^^^^^^
```

Forces **scalarisation**—but at least the compiler will tell us!

Better way of writing it

```
export void relax_ispc(uniform float output[],
                       uniform float input[],
                       uniform int n) {
  if (programIndex == 0) {
    output[0] =(input[n-1]+input[0]+input[1])/3;
    output[n-1] = (input[n-2]+input[n-1]+input[0])/3;
  }
  foreach (i = 1 ... n-2) {
    float l, c, r;
    c=input[i];
    l=input[i-1];
    r=input[i+1];
    output[i] = (l + c + r)/3;
  }
}
```

Runtime for n=1000000: $32\mu s$ — **much better!**

Better way of writing it

```
export void relax_ispc(uniform float output[],
                       uniform float input[],
                       uniform int n) {
  if (programIndex == 0) {
    output[0] =(input[n-1]+input[0]+input[1])/3;
    output[n-1] = (input[n-2]+input[n-1]+input[0])/3;
  }
  foreach (i = 1 ... n-2) {
    float l, c, r;
    c=input[i];
    l=input[i-1];
    r=input[i+1];
    output[i] = (l + c + r)/3;
  }
}
```

Runtime for n=1000000: 32μs — **much better!**
If we do the same thing to the C implementation... 38μs.

Memory accesses

Uniform
```
vs[i]; // 'uniform' i

movss  xmm0, dword ptr [rdi + 4*rax]
shufps xmm0, xmm0, 0
```

Regular
```
vs[programIndex];

movups  (%rdi), %xmm0
```

Irregular
```
vs[is[programIndex]];

movss    xmm0, dword ptr [rdi + rsi]
insertps xmm0, dword ptr [rdi + rcx], 16
insertps xmm0, dword ptr [rdi + r8],  32
insertps xmm0, dword ptr [rdi + rax], 48
```

`ispc` can always fall back to scalar code.

The GPU perspective

- GPUs have the notion of *coalesced memory*.
- In the same clock cycle, neighbouring threads should access neighbouring memory locations.
- Allows a single memory bus transaction to service multiple threads.

`ispc` is similar, but not identical.

+ GPU coalesced memory is *dynamic*—as long as the indexes are OK at runtime, all is well.
- With `ispc`, the indexing must be *statically* regular.

- This is because (most) CPUs do not actually have real gather instructions.
- GPUs are *throughput-oriented*, CPUs often more about *latency*.

Implementing filtering in ispc

```
int filter(float *output, float *input, int n) {
  int j = 0;
  for (int i = 0; i < n; i++) {
    float v = input[i];
    if (v >= 0) {
      output[j++] = v;
    }
  }
  return j;
}
```

- n=100000: 461μs
- Looks sequential, but we know better
- How do we filter in parallel?

Filtering in Futhark

```
def filter (as: []f32): []f32 =
  let keep = map (\a -> if a >= 0 then 1 else 0) as
  let offsets1 = scan (+) 0 keep
  let num_to_keep = reduce (+) 0 keep
  in scatter (replicate num_to_keep 0)
             (map2 (\i k ->
                      if k == 1
                      then i-1
                      else -1)
                   offsets1 keep)
             as
```

- `scan` then `scatter`
- Is that a good fit for `ispc`?

Filtering in Futhark

```
def filter (as: []f32): []f32 =
  let keep = map (\a -> if a >= 0 then 1 else 0) as
  let offsets1 = scan (+) 0 keep
  let num_to_keep = reduce (+) 0 keep
  in scatter (replicate num_to_keep 0)
             (map2 (\i k ->
                      if k == 1
                      then i-1
                      else -1)
                   offsets1 keep)
             as
```

- `scan` then `scatter`
- Is that a good fit for `ispc`?
- **No**, because we do not need to *maximise* parallelism
- `n=100000`: 51.30μs on `gpu04-diku-apl`

Sliding-window SIMD

Idea: slide a `programCount`-sized window sequentially over the input.

```
uniform int m = 0;
foreach (i = 0 ... n) {
  float j = input[i];
  int keep = j >= 0;
  int offset = exclusive_scan_add(keep);
  output[m + offset] = j;
  m += reduce_add(keep);
}
return m;
```

Correct?

Sliding-window SIMD

Idea: slide a `programCount`-sized window sequentially over the input.

```
uniform int m = 0;
foreach (i = 0 ... n) {
  float j = input[i];
  int keep = j >= 0;
  int offset = exclusive_scan_add(keep);
  output[m + offset] = j;
  m += reduce_add(keep);
}
return m;
```

Correct?

No, might write garbage.

```
uniform int m = 0;
foreach (i = 0 ... n) {
  float j = input[i];
  int keep = j >= 0;
  int offset = exclusive_scan_add(keep);
  if (keep) {
    output[m + offset] = j;
  }
  m += reduce_add(keep);
}
return m;
```

- n=100000: 407μs
- What is the likely culprit?

```
uniform int m = 0;
foreach (i = 0 ... n) {
  float j = input[i];
  int keep = j >= 0;
  int offset = exclusive_scan_add(keep);
  if (keep) {
    output[m + offset] = j;
  }
  m += reduce_add(keep);
}
return m;
```

- n=100000: 407μs
- What is the likely culprit?
- That **predicated memory write**!
- Optimisation: turn control flow into data.

```
uniform int m = 0;
foreach (i = 0 ... n) {
  float j = input[i];
  int keep = j >= 0;
  int offset = exclusive_scan_add(keep);
  if (!keep) {
    offset = programCount-1;
  }
  output[m + offset] = j;
  m += reduce_add(keep);
}
return m;
```

- n=100000: 137μs
- **Why is this valid?**

Summary

- Almost all CPUs support some kind of SIMD instructions.
- Utilising them can result in nontrivial speedup.
- Manual vectorisation sucks.
- Auto-vectorisation sucks.
- The SPMD programming model seems promising.

For the weekly assignment

- Several tasks about implementing things in `ispc`.
- See how fast you can make them go compared to C.
- 100 $\times$ is when you write Futhark programs; don't expect that much.
- On Hendrix, use `module load ispc`, but you can also use your own computer.