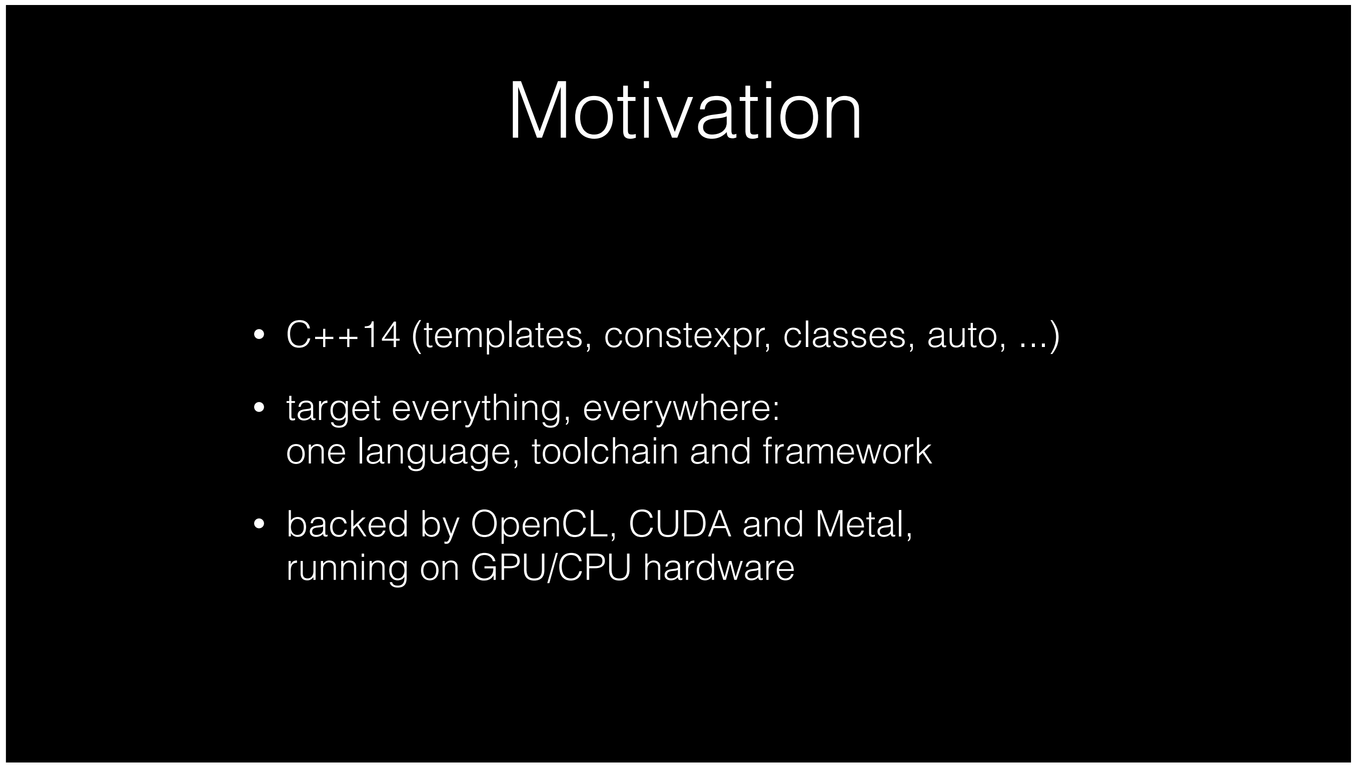
A Multi-platform C++ Language for Parallel Computing

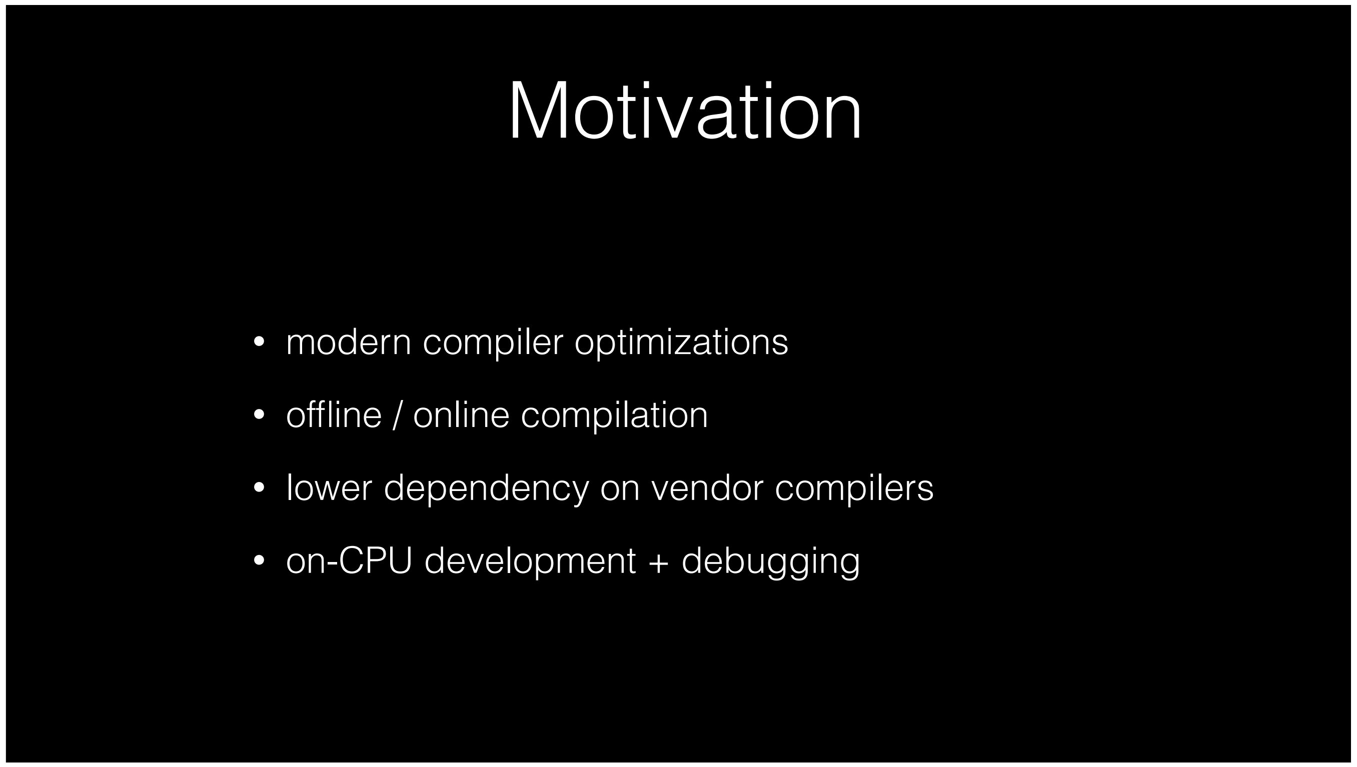
Florian Ziesche


Motivation

- C++14 (templates, constexpr, classes, auto, ...)
- target everything, everywhere:
  one language, toolchain and framework
- backed by OpenCL, CUDA and Metal,
  running on GPU/CPU hardware

* C++: GLSL/OpenGL and C99/OpenCL too primitive, have C++11 in Metal and CUDA now, but no common language between all of them (also: C++14 has much improved constexpr).

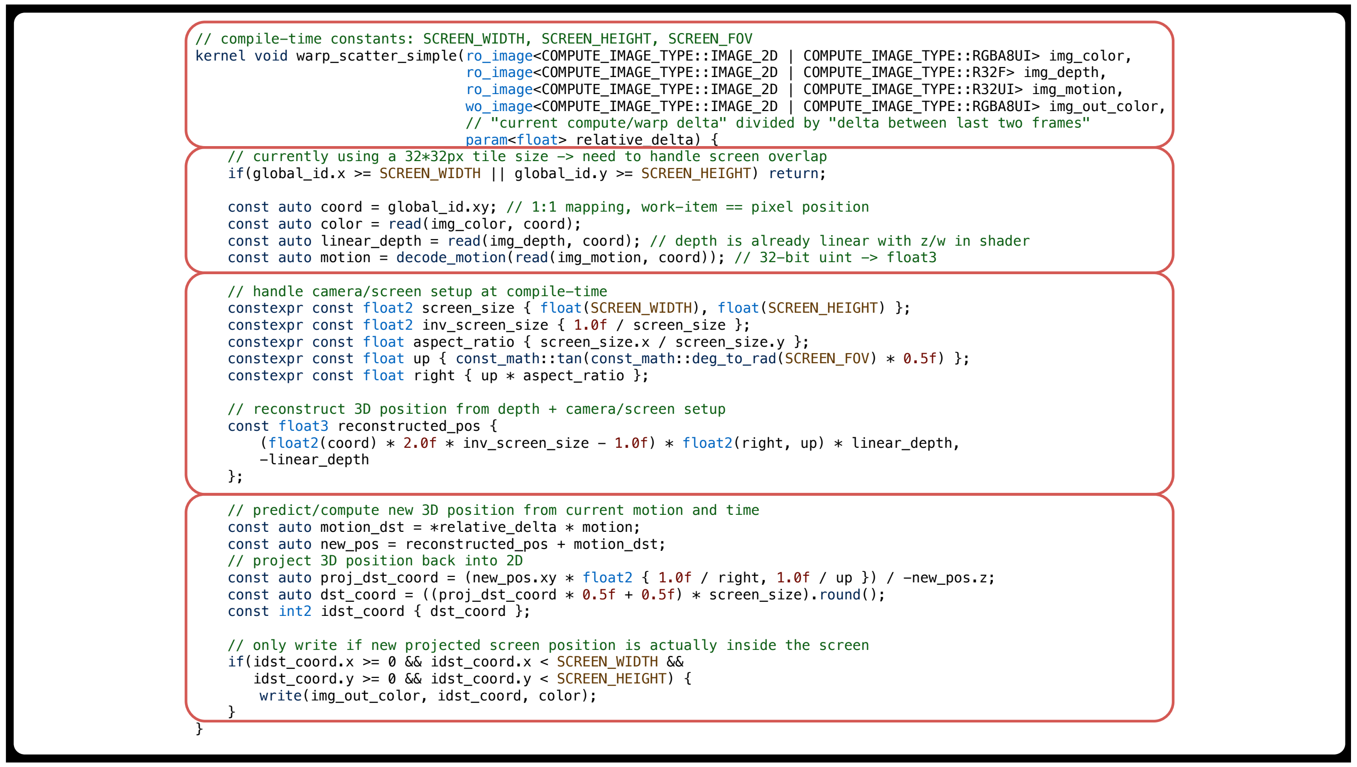
* targets: target everything from NVIDIA GPUs via CUDA, to AMD and Intel CPUs and GPUs via OpenCL/SPIR and Apple GPUs via Metal (also host execution for debugging/development purposes). This is achieved through a common device-side language (C++14 +/- extensions/restrictions) and library, and a common library/framework on the host. OS support for Linux, OS X, Windows and iOS.



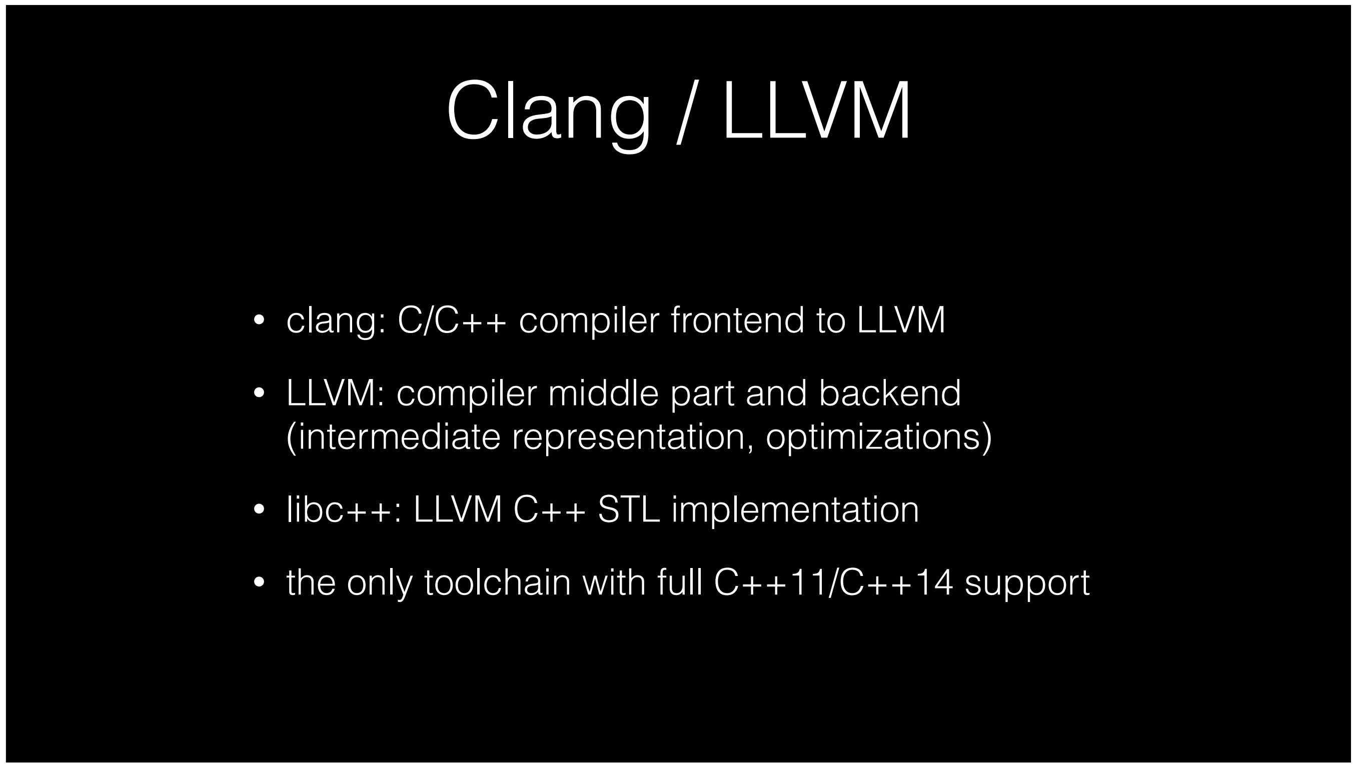
Motivation

- modern compiler optimizations
- offline / online compilation
- lower dependency on vendor compilers
- on-CPU development + debugging

* compilation: can offline/online compile to SPIR/PTX/AIR (full toolchain run, up to the point where the binary is handed over to the driver). bonus/in general: can compile multiple kernels in one source file at once; can also use pre-compiled headers; potentially also linking of multiple source files/objects
* dependence on vendor conformance is lessened: they only need to compile/understand LLVM or PTX code, which is already pretty close to assembly (things can still go wrong of course, but less so than when having to deal with different interpretations of high-level language constructs)
* can develop + debug code on the host CPU for convenience, less reboots ...



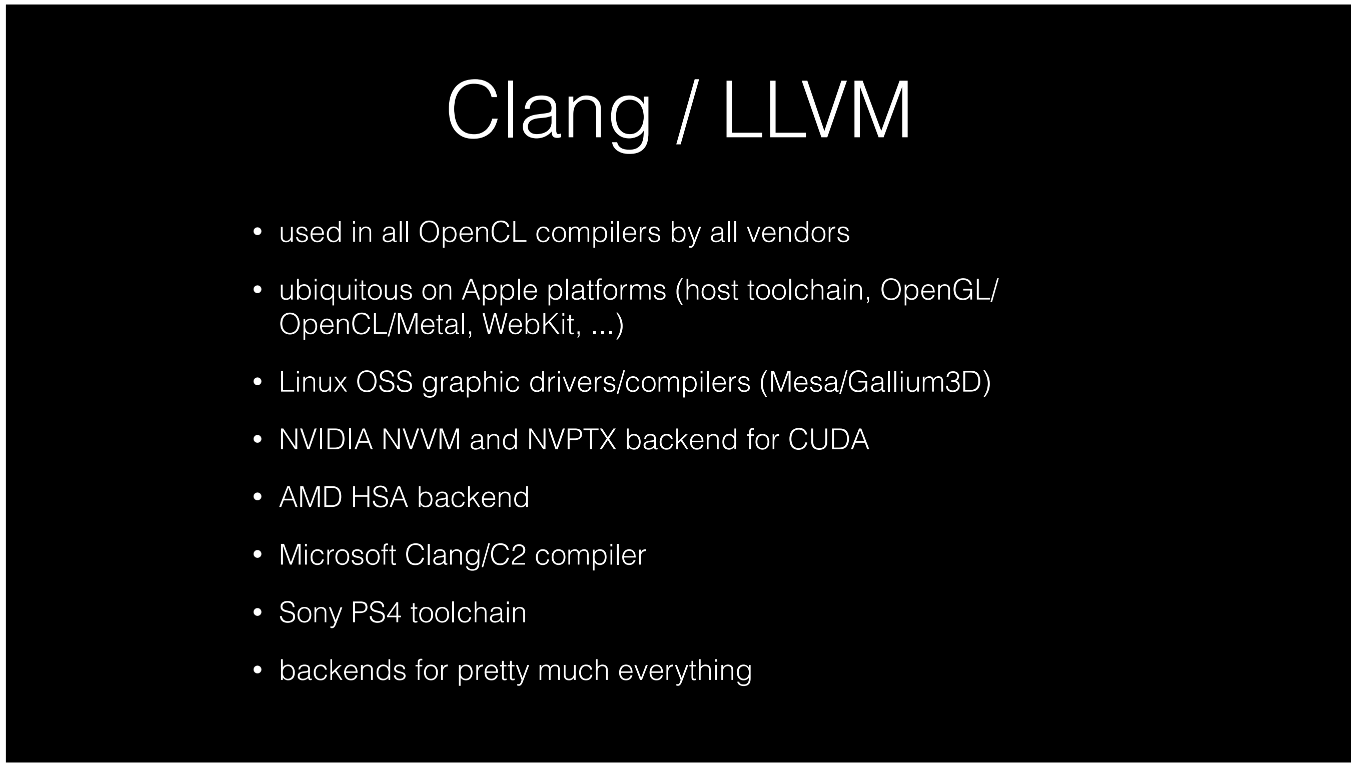
```
// compile-time constants: SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_FOV
kernel void warp_scatter_simple(ro_image<COMPUTE_IMAGE_TYPE::IMAGE_2D | COMPUTE_IMAGE_TYPE::RGBA8UI> img_color,
                                ro_image<COMPUTE_IMAGE_TYPE::IMAGE_2D | COMPUTE_IMAGE_TYPE::R32F> img_depth,
                                ro_image<COMPUTE_IMAGE_TYPE::IMAGE_2D | COMPUTE_IMAGE_TYPE::R32UI> img_motion,
                                wo_image<COMPUTE_IMAGE_TYPE::IMAGE_2D | COMPUTE_IMAGE_TYPE::RGBA8UI> img_out_color,
                                // "current compute/warp delta" divided by "delta between last two frames"
                                param<float> relative_delta) {
    // currently using a 32*32px tile size -> need to handle screen overlap
    if(global_id.x >= SCREEN_WIDTH || global_id.y >= SCREEN_HEIGHT) return;

    const auto coord = global_id.xy; // 1:1 mapping, work-item == pixel position
    const auto color = read(img_color, coord);
    const auto linear_depth = read(img_depth, coord); // depth is already linear with z/w in shader
    const auto motion = decode_motion(read(img_motion, coord)); // 32-bit uint -> float3

    // handle camera/screen setup at compile-time
    constexpr const float2 screen_size { float(SCREEN_WIDTH), float(SCREEN_HEIGHT) };
    constexpr const float2 inv_screen_size { 1.0f / screen_size };
    constexpr const float aspect_ratio { screen_size.x / screen_size.y };
    constexpr const float up { const_math::tan(const_math::deg_to_rad(SCREEN_FOV) * 0.5f) };
    constexpr const float right { up * aspect_ratio };

    // reconstruct 3D position from depth + camera/screen setup
    const float3 reconstructed_pos {
        (float2(coord) * 2.0f * inv_screen_size - 1.0f) * float2(right, up) * linear_depth,
        -linear_depth
    };

    // predict/compute new 3D position from current motion and time
    const auto motion_dst = *relative_delta * motion;
    const auto new_pos = reconstructed_pos + motion_dst;
    // project 3D position back into 2D
    const auto proj_dst_coord = (new_pos.xy * float2 { 1.0f / right, 1.0f / up }) / -new_pos.z;
    const auto dst_coord = ((proj_dst_coord * 0.5f + 0.5f) * screen_size).round();
    const int2 idst_coord { dst_coord };

    // only write if new projected screen position is actually inside the screen
    if(idst_coord.x >= 0 && idst_coord.x < SCREEN_WIDTH &&
       idst_coord.y >= 0 && idst_coord.y < SCREEN_HEIGHT) {
        write(img_out_color, idst_coord, color);
    }
}
```



Clang / LLVM

- clang: C/C++ compiler frontend to LLVM
- LLVM: compiler middle part and backend (intermediate representation, optimizations)
- libc++: LLVM C++ STL implementation
- the only toolchain with full C++11/C++14 support

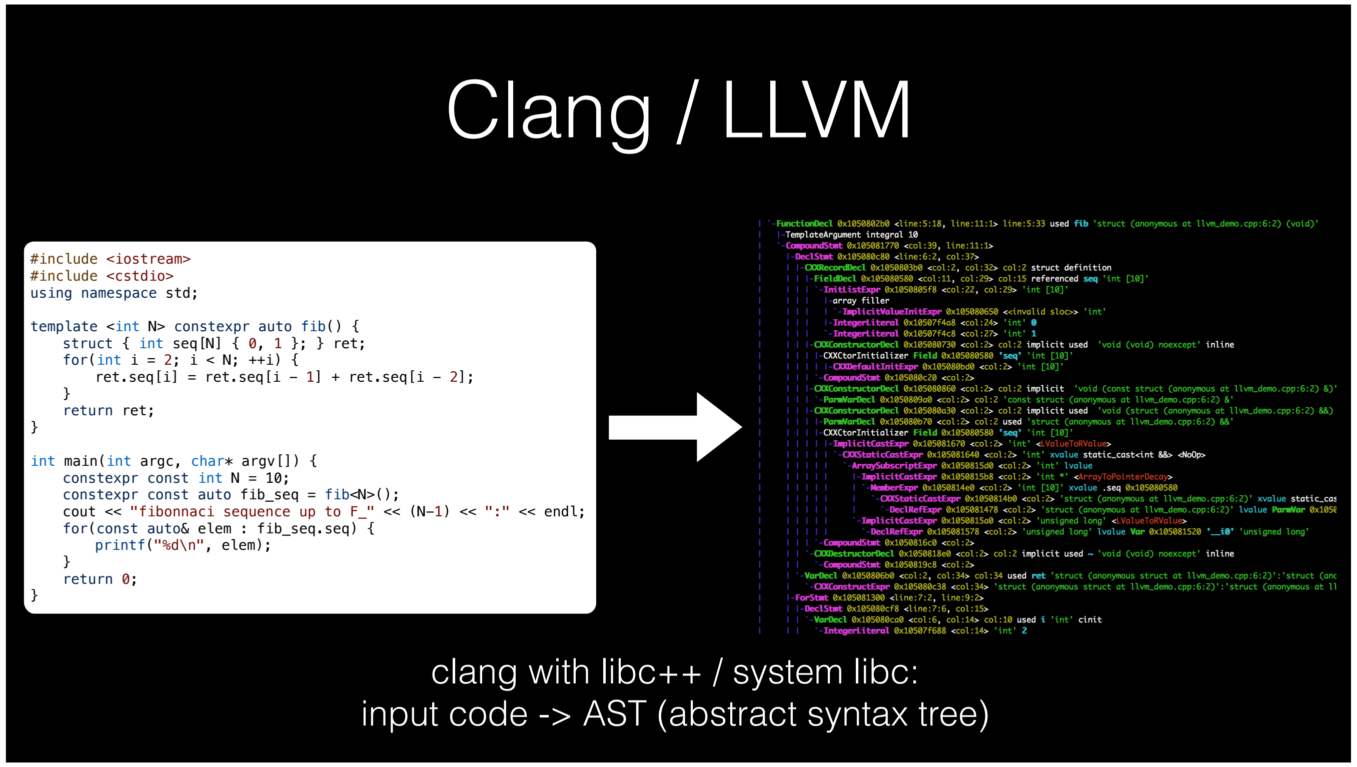
* LLVM both the name of the umbrella project of all toolchain parts, as well as the name of the compiler middle and backend part (optimization and codegen)
* LLVM is not an acronym for anything any more
* clang: lexer/parser/sema for C89 - C11 (with OpenCL C99), C++98 - C++1z, Objective-C
* LLVM: LLVM Intermediate Representation used as a single target for all frontends (pretty much SSA form + "other stuff"), runs all optimizations on this format, all backends transforming this format to their target assembly
* http://clang.llvm.org / http://llvm.org / http://libcxx.llvm.org



Clang / LLVM

- used in all OpenCL compilers by all vendors
- ubiquitous on Apple platforms (host toolchain, OpenGL/OpenCL/Metal, WebKit, ...)
- Linux OSS graphic drivers/compilers (Mesa/Gallium3D)
- NVIDIA NVVM and NVPTX backend for CUDA
- AMD HSA backend
- Microsoft Clang/C2 compiler
- Sony PS4 toolchain
- backends for pretty much everything

* used everywhere nowadays
* main contributors: Apple and Google
* also working on it: Intel, IBM, NVIDIA, AMD, ARM, Sony, Qualcomm, ...



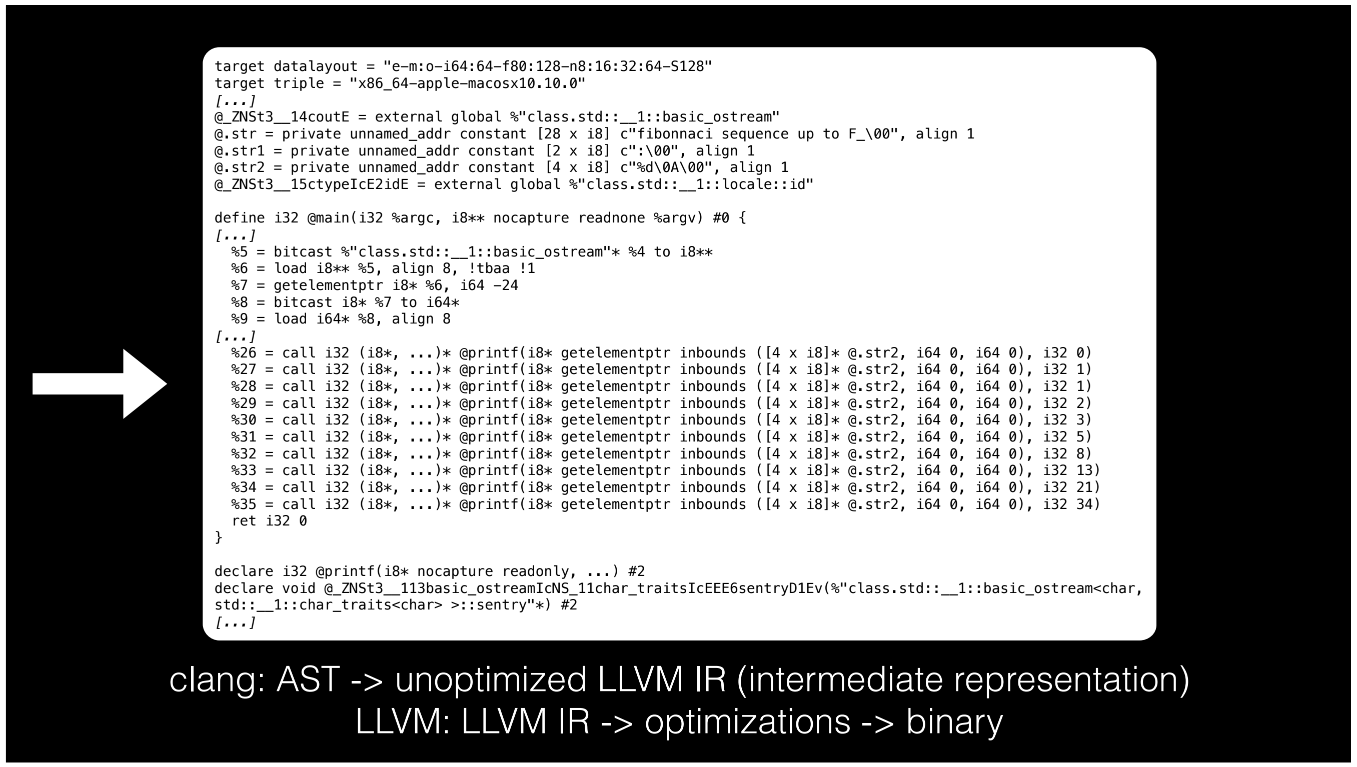
Clang / LLVM

```cpp
#include <iostream>
#include <cstdio>
using namespace std;

template <int N> constexpr auto fib() {
    struct { int seq[N] { 0, 1 }; } ret;
    for(int i = 2; i < N; ++i) {
        ret.seq[i] = ret.seq[i - 1] + ret.seq[i - 2];
    }
    return ret;
}

int main(int argc, char* argv[]) {
    constexpr const int N = 10;
    constexpr const auto fib_seq = fib<N>();
    cout << "fibonnaci sequence up to F_" << (N-1) << ":" << endl;
    for(const auto& elem : fib_seq.seq) {
        printf("%d\n", elem);
    }
    return 0;
}
```

```
| `-FunctionDecl 0x1050802b0 <line:5:18, line:11:1> line:5:33 used fib 'struct (anonymous at llvm_demo.cpp:6:2) (void)'
|   |-TemplateArgument integral 10
|   `-CompoundStmt 0x105081770 <col:39, line:11:1>
|     |-DeclStmt 0x105080c80 <line:6:2, col:37>
|     | |-CXXRecordDecl 0x1050803b0 <col:2, col:32> col:2 struct definition
|     | | |-FieldDecl 0x105080580 <col:11, col:29> col:15 referenced seq 'int [10]'
|     | | | `-InitListExpr 0x1050805f8 <col:22, col:29> 'int [10]'
|     | | |   |-array filler
|     | | |   | `-ImplicitValueInitExpr 0x105080650 <<invalid sloc>> 'int'
|     | | |   |-IntegerLiteral 0x10507f4a8 <col:24> 'int' 0
|     | | |   `-IntegerLiteral 0x10507f4c8 <col:27> 'int' 1
|     | | |-CXXConstructorDecl 0x105080730 <col:2> col:2 implicit used  'void (void) noexcept' inline
|     | | | |-CXXCtorInitializer Field 0x105080580 'seq' 'int [10]'
|     | | | | |-CXXDefaultInitExpr 0x105080bd0 <col:2> 'int [10]'
|     | | | `-CompoundStmt 0x105080c20 <col:2>
|     | | |-CXXConstructorDecl 0x105080860 <col:2> col:2 implicit  'void (const struct (anonymous at llvm_demo.cpp:6:2) &)'
|     | | | `-ParmVarDecl 0x1050809a0 <col:2> col:2 'const struct (anonymous at llvm_demo.cpp:6:2) &'
|     | | |-CXXConstructorDecl 0x105080a30 <col:2> col:2 implicit used  'void (struct (anonymous at llvm_demo.cpp:6:2) &&)
|     | | | |-ParmVarDecl 0x105080b70 <col:2> col:2 used 'struct (anonymous at llvm_demo.cpp:6:2) &&'
|     | | | |-CXXCtorInitializer Field 0x105080580 'seq' 'int [10]'
|     | | | | |-ImplicitCastExpr 0x105081670 <col:2> 'int' <LValueToRValue>
|     | | | | | `-CXXStaticCastExpr 0x105081640 <col:2> 'int' xvalue static_cast<int &&> <NoOp>
|     | | | | |   `-ArraySubscriptExpr 0x1050815d0 <col:2> 'int' lvalue
|     | | | | |     |-ImplicitCastExpr 0x1050815b8 <col:2> 'int *' <ArrayToPointerDecay>
|     | | | | |     | `-MemberExpr 0x1050814e0 <col:2> 'int [10]' xvalue .seq 0x105080580
|     | | | | |     |   `-CXXStaticCastExpr 0x1050814b0 <col:2> 'struct (anonymous at llvm_demo.cpp:6:2)' xvalue static_cas
|     | | | | |     |     `-DeclRefExpr 0x105081478 <col:2> 'struct (anonymous at llvm_demo.cpp:6:2)' lvalue ParmVar 0x1050
|     | | | | |     `-ImplicitCastExpr 0x1050815a0 <col:2> 'unsigned long' <LValueToRValue>
|     | | | | |       `-DeclRefExpr 0x105081578 <col:2> 'unsigned long' lvalue Var 0x105081520 '__i0' 'unsigned long'
|     | | | `-CompoundStmt 0x1050816c0 <col:2>
|     | | `-CXXDestructorDecl 0x1050818e0 <col:2> col:2 implicit used ~ 'void (void) noexcept' inline
|     | |   `-CompoundStmt 0x1050819c8 <col:2>
|     | `-VarDecl 0x1050806b0 <col:2, col:34> col:34 used ret 'struct (anonymous struct at llvm_demo.cpp:6:2)':'struct (anc
|     |   `-CXXConstructExpr 0x105080c38 <col:34> 'struct (anonymous struct at llvm_demo.cpp:6:2)':'struct (anonymous at ll
|     |-ForStmt 0x105081300 <line:7:2, line:9:2>
|     | |-DeclStmt 0x105080cf8 <line:7:6, col:15>
|     | | `-VarDecl 0x105080ca0 <col:6, col:14> col:10 used i 'int' cinit
|     | |   `-IntegerLiteral 0x10507f688 <col:14> 'int' 2
```

clang with libc++ / system libc:
input code -> AST (abstract syntax tree)

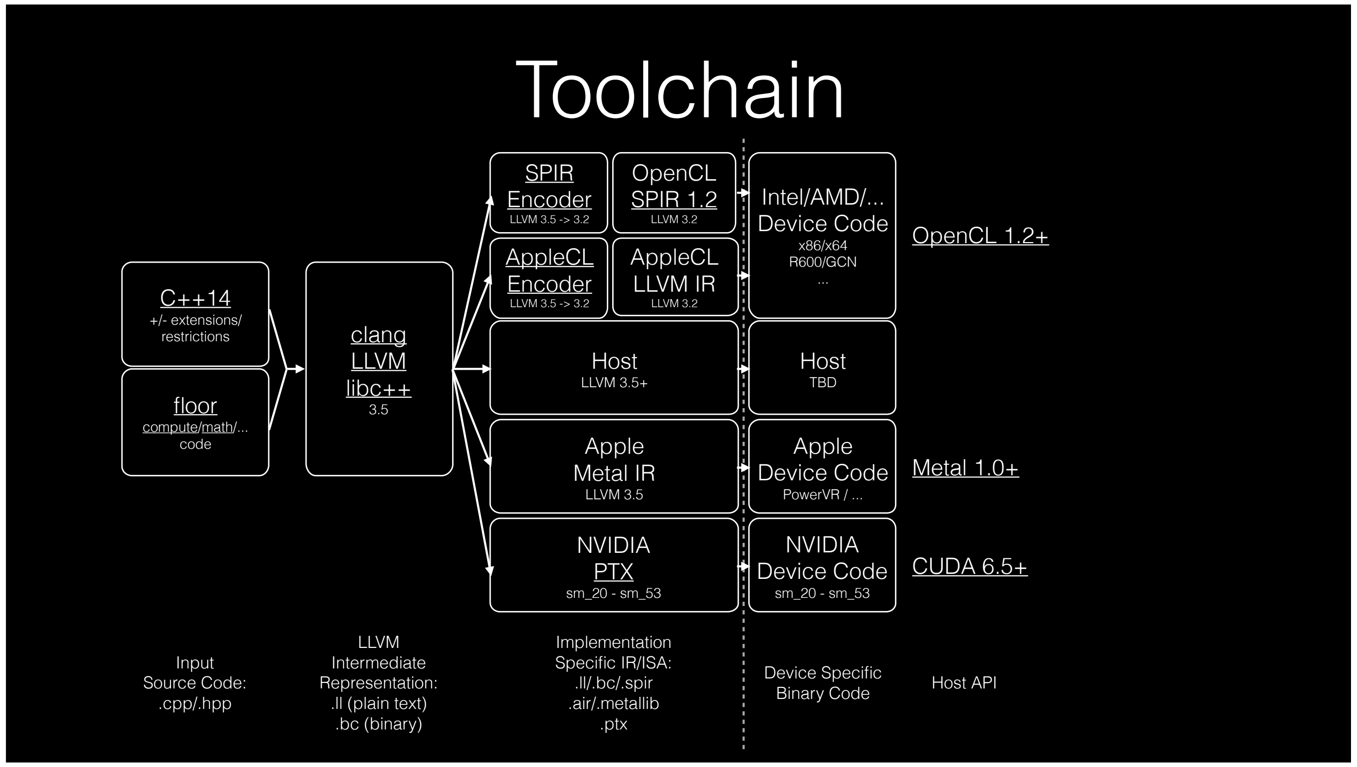
* clang: transform C++ code to internal AST representation (note: AST on the right just a tiny cut-out)
* completely unoptimized at this point!
* C++ STL headers via libc++, C headers via the systems libc implementation



```
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.10.0"
[...]
@_ZNSt3__14coutE = external global %"class.std::__1::basic_ostream"
@.str = private unnamed_addr constant [28 x i8] c"fibonnaci sequence up to F_\00", align 1
@.str1 = private unnamed_addr constant [2 x i8] c":\00", align 1
@.str2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@_ZNSt3__15ctypeIcE2idE = external global %"class.std::__1::locale::id"

define i32 @main(i32 %argc, i8** nocapture readnone %argv) #0 {
[...]
  %5 = bitcast %"class.std::__1::basic_ostream"* %4 to i8**
  %6 = load i8** %5, align 8, !tbaa !1
  %7 = getelementptr i8* %6, i64 -24
  %8 = bitcast i8* %7 to i64*
  %9 = load i64* %8, align 8
[...]
  %26 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 0)
  %27 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 1)
  %28 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 1)
  %29 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 2)
  %30 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 3)
  %31 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 5)
  %32 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 8)
  %33 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 13)
  %34 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 21)
  %35 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str2, i64 0, i64 0), i32 34)
  ret i32 0
}

declare i32 @printf(i8* nocapture readonly, ...) #2
declare void @_ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryD1Ev(%"class.std::__1::basic_ostream<char,
std::__1::char_traits<char> >::sentry"*) #2
[...]
```

clang: AST -> unoptimized LLVM IR (intermediate representation)
LLVM: LLVM IR -> optimizations -> binary

* note: trimmed down output
* clang: transform internal AST to LLVM IR (still unoptimized)
* LLVM: transform IR, apply optimizations
* LLVM backend: transform IR to binary
* IR: "kind of" assembly, but higher level and properly defined, also quite understandable/straightforward
* contains functions, global variables, target information, misc metadata and attributes



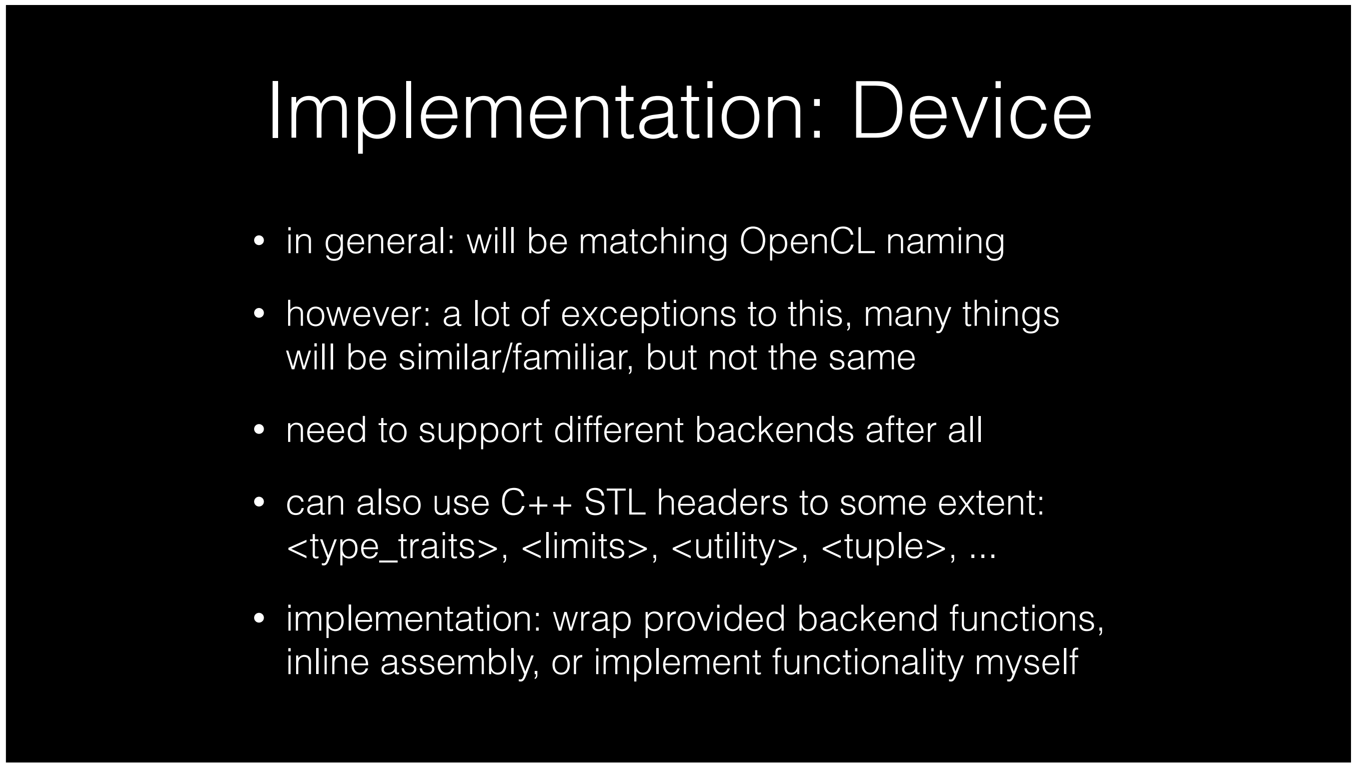
*   what I'll be doing, or already did:

* extend Clang/LLVM to compile to OpenCL/SPIR (Standard Portable Intermediate Representation) in C++ mode, fix some CUDA issues, write special LLVM metadata to support SPIR/AppleCL/Metal (kernel and kernel arg information, e.g. sizes, address spaces), extend with support for OpenCL depth + MSAA images

* write device side library/code/wrapper for backend functionality

* write general math and vector library (for things like float3, int2, ...)

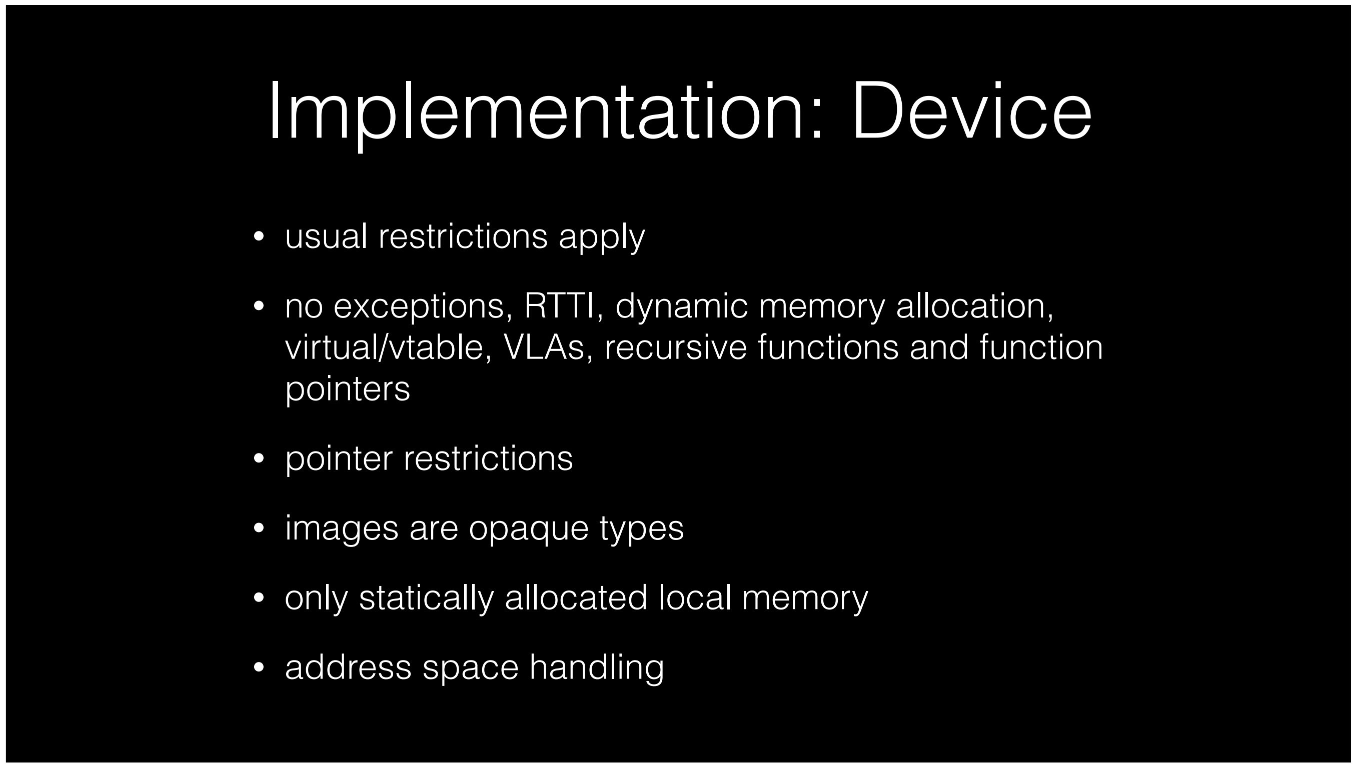
* also: compile-time math is awesome



Implementation: Device

- in general: will be matching OpenCL naming
- however: a lot of exceptions to this, many things will be similar/familiar, but not the same
- need to support different backends after all
- can also use C++ STL headers to some extent: <type_traits>, <limits>, <utility>, <tuple>, ...
- implementation: wrap provided backend functions, inline assembly, or implement functionality myself

* usually more C++ style
* additionally supported STL headers: <functional>, <algorithm>, <array> (although replaced by const_array), <memory>, <iterator>, <iosfwd>, <atomic> (memory and atomic only partially functional, also no I/O streams ...)
* note that these don't work as-is, but need patching, because of things not being supported on compute devices
* OpenCL library easy to support, just need to wrap everything
* Metal kind of easy to support, uses ASM labels everywhere
* CUDA much harder / low-level, need to implement a lot of things myself (image, math, ...)



Implementation: Device

- usual restrictions apply
- no exceptions, RTTI, dynamic memory allocation, virtual/vtable, VLAs, recursive functions and function pointers
- pointer restrictions
- images are opaque types
- only statically allocated local memory
- address space handling

* restrictions imposed by graphics hardware (and their backends)

* either make no sense, or just impossible to implement (just yet)

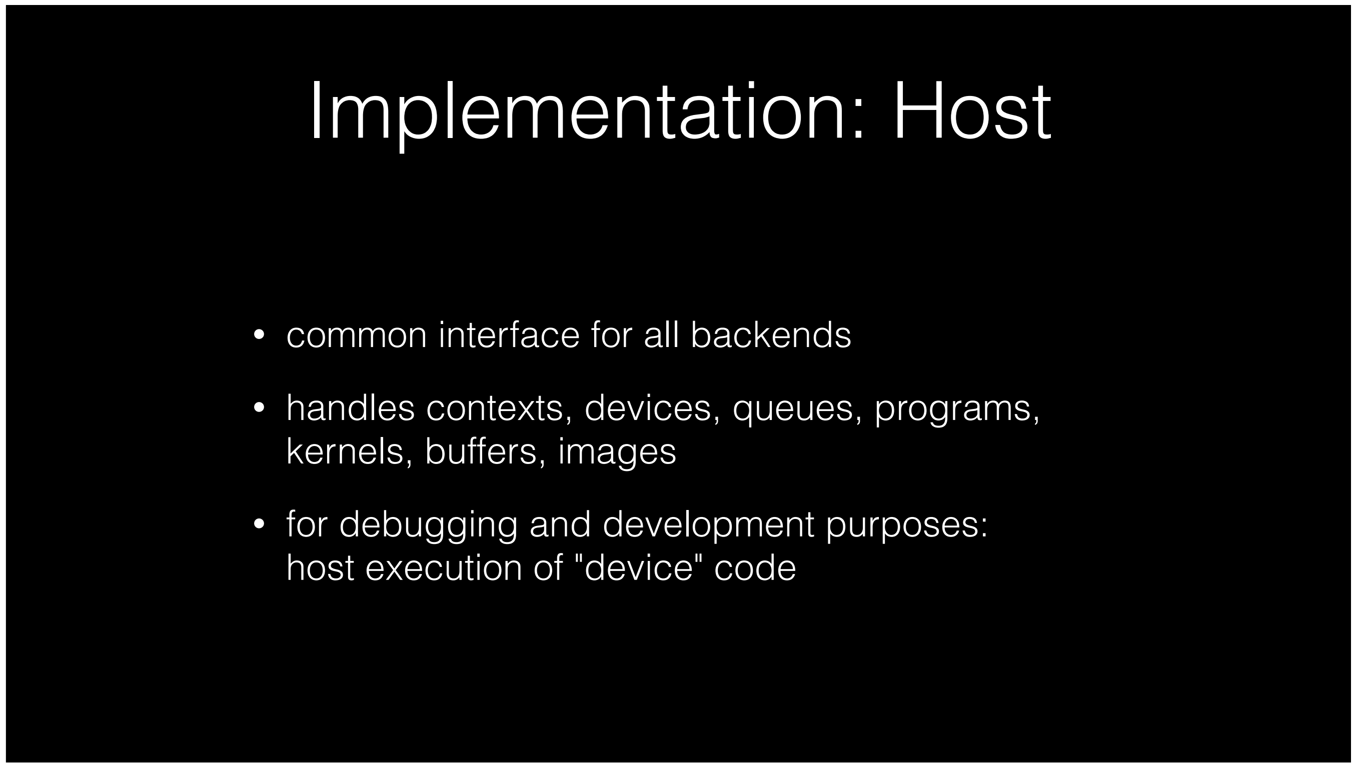
* no pointers to pointers in kernel arguments (i.e. host), or: host doesn't know about the address of device-side memory and can't nest device-side memory

* no pointers to images, no objects containing images, images can only be defined as kernel arguments

* for C++: references are pointers too

* local memory size must be known at compile-time for now (all backends do support dynamically sized local memory, but in very different ways - couldn't find a good way to consolidate this yet, but will do this at a later point), better for optimization anyways (static load offsets, loop unrolling)!

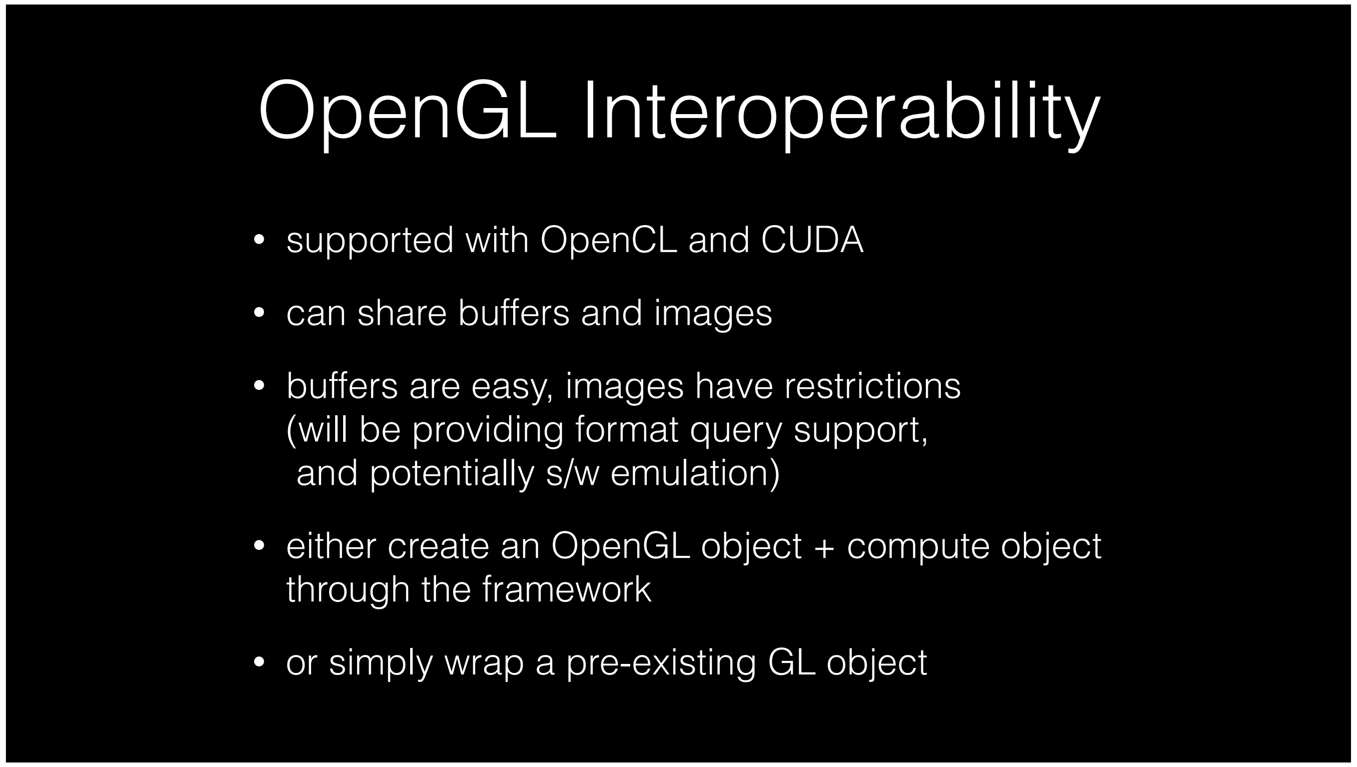
* address spaces must be handled with care ("this" knows no address space), use the containers I provide and all will be well (working on this right now, will be a lot less restrictive and more as you would expect, i.e. calling a member function of a class object in global memory will load the object into private address space first, call the function, then store the result back if non-const, with LLVM taking care of redundancy)



Implementation: Host

- common interface for all backends
- handles contexts, devices, queues, programs, kernels, buffers, images
- for debugging and development purposes: host execution of "device" code

* no need to handle backend or API specific things, unless you want to
* not implemented yet, but will do so soon(tm): host execution of device code, both single-threaded, as well as multi-threaded. not intended for "fast execution" (use Intels CPU implementation for this), but rather for debugging and development
* could also run clang/LLVM sanitizers through this (especially ThreadSanitizer to find e.g. race conditions and AddressSanitizer to find memory use issues, e.g. out-of-bound access)



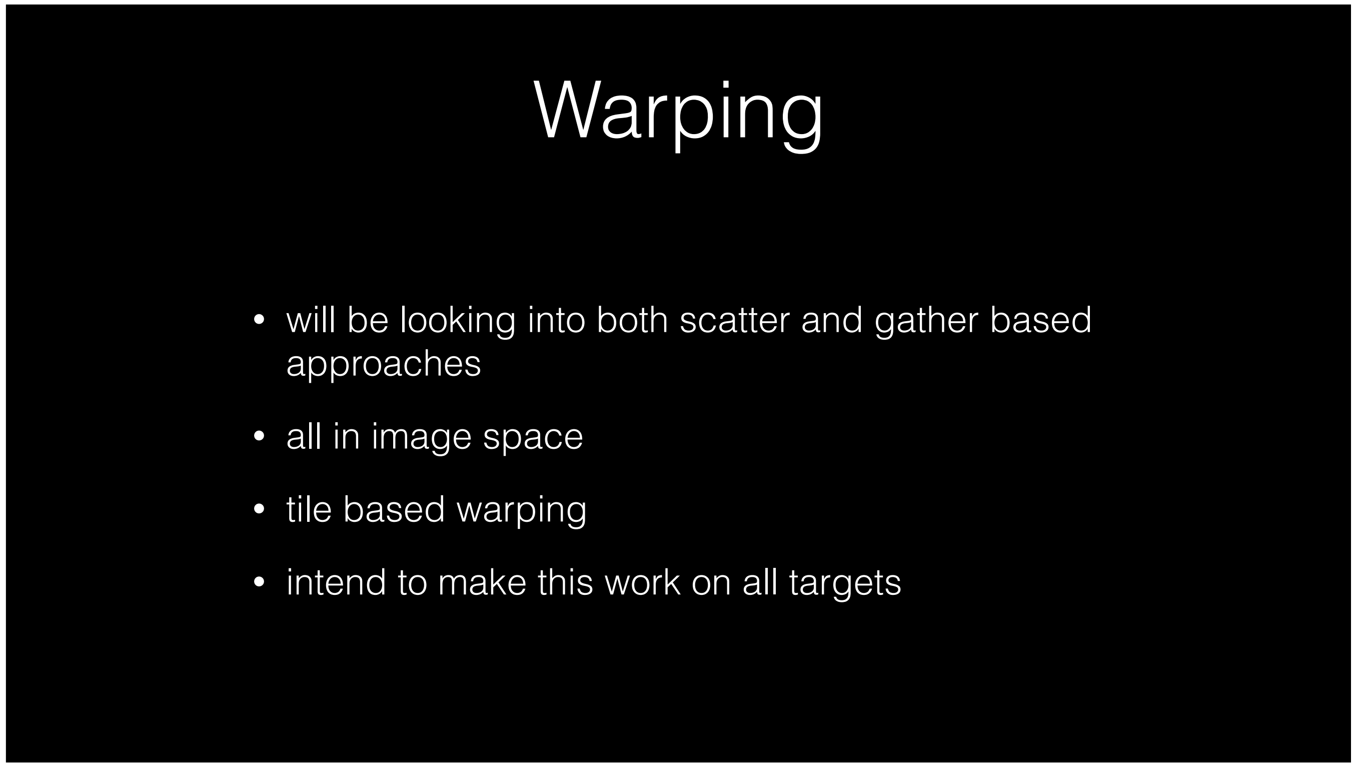
OpenGL Interoperability

- supported with OpenCL and CUDA
- can share buffers and images
- buffers are easy, images have restrictions
  (will be providing format query support,
  and potentially s/w emulation)
- either create an OpenGL object + compute object through the framework
- or simply wrap a pre-existing GL object

* buffers are just memory, so they just work
* images are a lot of work, immense amount of formats and format combinations
* OpenCL provides ways to query format support, for CUDA this is kind of obvious (for the most part), Metal supports the same formats for compute and graphics
* generally: 1, 2 or 4 channel R/RG/RGBA formats with 8-bit, 16-bit or 32-bit per channel usually work w/o problems, also 1D, 2D, 3D (arrays, cube maps and MSAA not just yet). Depth is an issue on CUDA, need to write or copy to R32F. In OpenCL, depth and MSAA need an extension (but supported by Intel, AMD and Apple).

Host Example

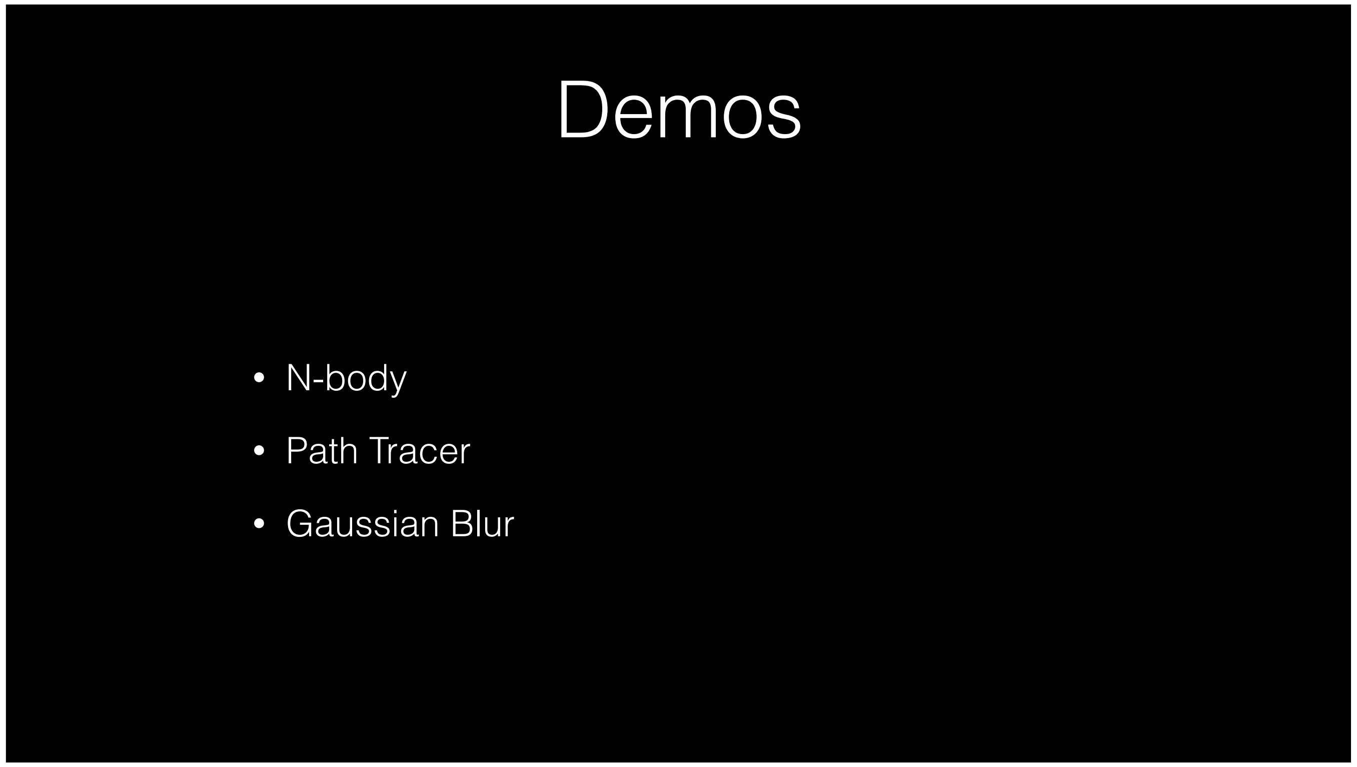
```
// get the default context, or: make_shared<opencl_compute>(), or cuda_compute, metal_compute
auto ctx = floor::get_compute_context();
// get the "fastest device" (units * clock speed), also: FASTEST_GPU/FASTEST_CPU, GPU0, GPU1, ...
auto dev = ctx->get_device(compute_device::TYPE::FASTEST);
// create a device queue (aka command queue, stream, ...)
auto dev_queue = ctx->create_queue(dev);

// compile program (all kernel functions in program), additional options are directly passed to the compiler
auto prog = ctx->add_program_file("program.cpp", "-DADDITIONAL_OPTIONS -Weverything -Isome/folder/");
// retrieve a specific kernel from the program, also: get_kernel_fuzzy
auto kernel = prog->get_kernel("some_kernel");

// wrap a pre-existing OpenGL image (texture target must be set, all else is queried)
auto img = ctx->wrap_image(dev, opengl_tex, GL_TEXTURE_2D); // also: create_image
// create a buffer on a specific device (can also mirror a host buffer/array/vector)
auto buffer = ctx->create_buffer(dev, sizeof(float4) * 1024 * 1024); // also: wrap_buffer

// schedule kernel for execution (2D, can also do 1D/3D), args are passed as variadic template
dev_queue->execute(kernel,
                   size2 { 1024, 1024 }, // global size, #work-items
                   size2 { 32, 32 },     // local/work-group size
                   img, buffer           // args...
);
```



Warping

- will be looking into both scatter and gather based approaches
- all in image space
- tile based warping
- intend to make this work on all targets

* show current demo if enough time?
* current progress uses scatter based approach, as shown in example code before (not depth correct though)
* image space: direct read/write access to every pixel
* tile based: use this for cooperative things, e.g. snap/scale/blur, hole filling, resampling, ...; lots of possibilities
* works on CUDA and OpenCL right now, will be interesting to see if Metal/mobile h/w is fast enough

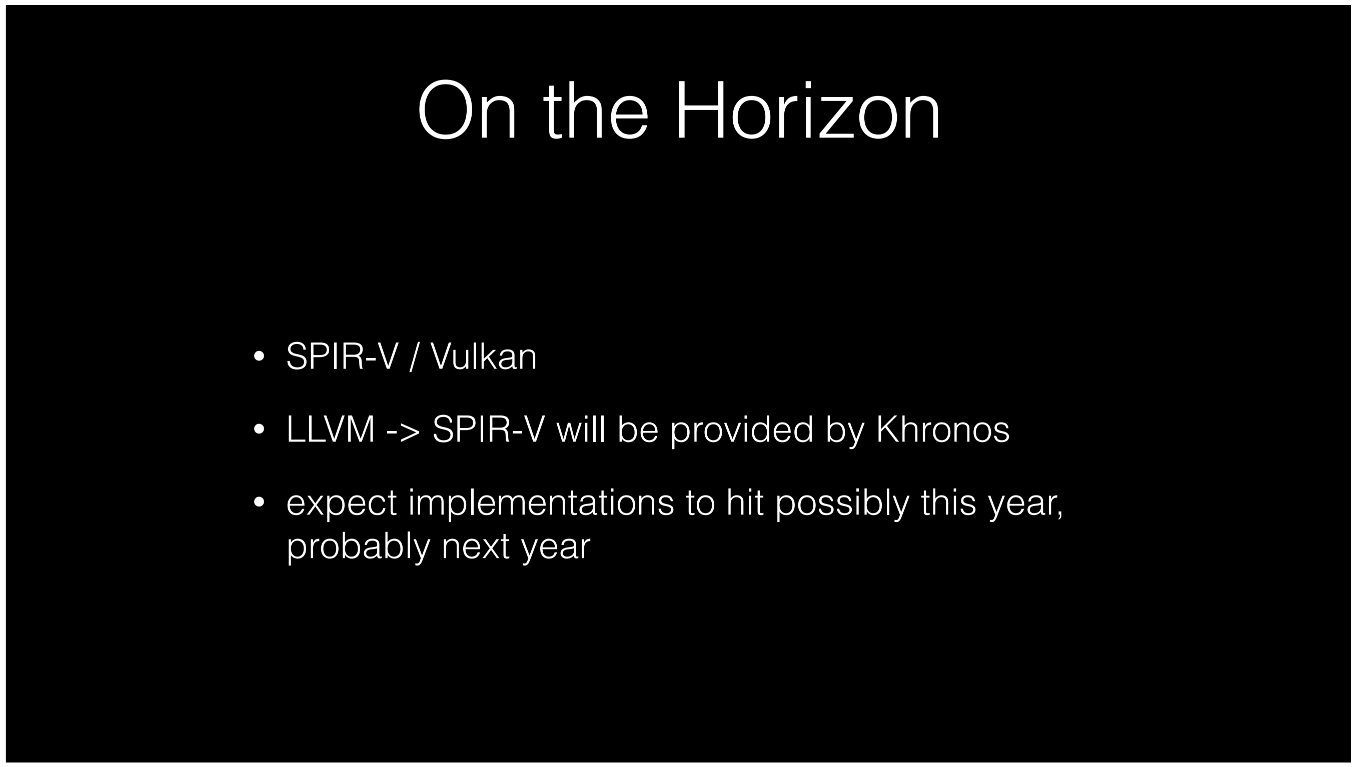


Demos

- N-body
- Path Tracer
- Gaussian Blur

* code for these: https://github.com/a2flo/floor_examples

* N-body YouTube video: https://www.youtube.com/watch?v=DoLe1c-eokI



On the Horizon

- SPIR-V / Vulkan
- LLVM -> SPIR-V will be provided by Khronos
- expect implementations to hit possibly this year, probably next year

* SPIR-V successor to SPIR, new binary format, completely different to LLVM, but same goals
* LLVM to SPIR-V conversion/targeting tools will be provided by Khronos, and there is really no other way for them to support C++ (already on the way to be merged into LLVM: http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-May/085538.html)
* might actually be able to write vertex/fragment/... shaders through LLVM with this (technically, this should already work with Metal, but I haven't looked into this yet)
* "just another backend", will of course implement support for this once there are working drivers and conversion tools
* hopefully not pulling a Longs Peak, really hoping for betas this year and a preliminary spec

The End

- Thanks for listening!
- Code available at:
  https://github.com/a2flo/floor
  https://github.com/a2flo/floor_examples