

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

**A Multi-platform C++ Language
for Parallel Computing**

submitted by

Florian Ziesche

submitted

December 15, 2015

Reviewers

Dr.-Ing. Tobias Ritschel
Prof. Dr.-Ing. Philipp Slusallek

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 15. Dezember 2015

(Ort/Place, Datum/Date)

(Unterschrift/Signature)

Contents

1. Abstract	1
2. Basics	2
2.1. LLVM and Clang	2
2.1.1. What is LLVM?	4
2.1.2. Why use Clang/LLVM for this project?	6
2.1.3. Clang/LLVM Compilation Process Overview	7
2.1.4. Compute Toolchain	8
2.2. Compute	10
2.2.1. Platform	10
2.2.2. Execution	11
2.2.3. Memory Hierarchy	13
2.3. Platforms	15
2.3.1. CUDA	15
2.3.2. OpenCL	15
2.3.3. Metal	16
2.3.4. Host-Compute	16
2.3.5. GPU/CPU Hardware	17
3. Implementation	18
3.1. Compiler	19
3.1.1. Address spaces in C++ Compute	19
3.1.2. Language Restrictions	24
3.1.3. Language Extensions & Modifications	27
3.1.4. Platform-specific Information & Idiosyncrasies	28
3.1.4.1. CUDA	29
3.1.4.2. OpenCL	30
3.1.4.3. Metal	31
3.1.4.4. Host-Compute	32
3.2. Host	36
3.2.1. Library: context	36
3.2.2. Library: device	40
3.2.3. Library: queue	41
3.2.4. Library: program	43
3.2.5. Library: kernel	43
3.2.6. Library: memory	44
3.2.7. Library: buffer	45
3.2.8. Library: image	48

3.2.9. Online/Offline Compilation	49
3.2.10. Debugging & Development	50
3.2.11. Graphics Interoperability	52
3.3. Device	53
3.3.1. Basic Math Library	53
3.3.2. Compute-specific Functionality	60
3.3.3. Image Functionality	68
3.3.4. Misc Functionality	74
3.3.5. STL	75
4. Application	77
4.1. Image-space Warping	77
4.1.1. Scatter approach	77
4.1.2. Gather approach	79
4.1.3. Library	81
4.1.4. Implementation	83
4.1.5. Example	87
4.2. Demos & Examples	88
4.2.1. Path Tracer	88
4.2.2. N-body Simulation	89
4.2.3. Image Gaussian Blur	92
5. Conclusion & Future	93
5.1. new Compute & Graphics APIs	94
5.2. new language and hardware functionality	95
A. Gaussian blur compile-time coefficient computation	97
B. Source Code Release	100

1. Abstract

While there are quite a few ways of programming GPUs nowadays, one usually has to make the choice between using a low-level language like C or domain specific language like GLSL that work almost everywhere, or using a high-level language like C++ which is then restricted to vendor specific hardware and implementations (CUDA and Metal). Furthermore, vendor-agnostic languages like GLSL or OpenCL C run into the issue that different vendors will interpret language specifications differently, thus again creating incompatibilities. For these reasons, this thesis and accompanying projects implement and show how to provide a unified C++14 Compute language and support library, targeting the major Compute platforms CUDA, OpenCL and Metal, running on Linux, OS X/iOS and Windows, on AMD, Apple, Intel and NVIDIA GPU and CPU hardware. To aid in development and debugging, a full Compute implementation fully running on the host CPU inside a user's application is also provided. To accomplish all of this, this project makes use of the LLVM compiler infrastructure which is now either the basis or one of the possible ways to program CUDA, OpenCL and Metal. Proving that this is in fact working, several applications were written and will be demonstrated. In particular, an Image-space Warping library implementing both scatter-based and gather-based approaches will be shown.

2. Basics

The following three sections provide a general introduction to Compute and supported platforms, how these general concepts map to specific hardware and a general introduction to the LLVM compiler infrastructure and why it is used for this project.

2.1. LLVM and Clang

Compute and Graphics languages and libraries have long been in the hands of GPU vendors, sometimes arriving at universally usable standards like OpenGL/GLSL or OpenCL/C, at other times creating languages that only work for an individual vendor or platform like CUDA or Metal. What all of these have in common however, is that the library and toolchain to compile code is always provided by the vendor, even and especially those parts which aren't specific to vendor hardware. Imagine Intel or AMD compilers and libraries were the only way to program on x86 - quite absurd from today's standpoint, but similar is still true for GPUs. This leads to the situation where language and library design are of arguable quality and usability, and are usually lagging behind the state-of-the-art. To the point of CUDA only getting C++11 support 3 years after [NVI15a] this was possible on CPU platforms, and an OpenCL C++ standard still only being in a provisional state [OWG15b] with no time frame known yet of when implementations will become available. On the Graphics side, things don't look much better, with GLSL not fundamentally changing over its lifespan of almost 12 years, continuing to be a very limited domain specific language. Another problem occurs with the open standards GLSL and OpenCL C, where vendors interpret these standards differently, resulting in differences and incompatibilities in library and language. Even worse, these differences will only become visible when compiling code individually for each vendor.

To solve this situation, this thesis and projects implement a unified C++14 compute language and support library, compiling to a low-level code representation that makes it possible to confine vendors to what they're particularly good at: optimizing for their hardware. On both accounts, this is where the LLVM compiler infrastructure comes in [LLV15o]. Over the past few years, LLVM has either become the basis or one of multiple ways to program for Compute platforms.

Starting with OpenCL on OS X back in 2008 [Wik15j], almost all OpenCL implementations since then have been using Clang [LLV15c] and LLVM as their toolchain of choice, including vendor implementations ranging from AMD, to Intel, to NVIDIA, to open source implementations like POCL [poc15] and Beignet

[Bei15]. While using the same toolchain, there wasn't a standardized way of making use of that fact, i.e. by allowing developers to compile to LLVM's intermediate representation (IR) format and let vendors deal with the final compilation to device code. Only since the finalization of the Khronos Standard Portable Intermediate Representation (SPIR) 1.2 specification in early 2014, being based on LLVM's 3.2 intermediate representation, and first implementations from Intel and AMD arriving the same year, has it been possible to officially program OpenCL device code using an external (and open source) toolchain. Even though being based on Clang/LLVM since its beginning, OpenCL on OS X can currently not be programmed via SPIR, but has to use Apple's proprietary AppleCL format, which is however very close to SPIR 1.2, as it is also based on LLVM 3.2 at this time. I made the necessary modifications to the compiler for this project, in part also using and modifying SPIR tools provided by Khronos [KG15b], so that this platform can be targeted as well.

On the CUDA [NVI15c] side, while there has been an unofficial LLVM backend since LLVM 2.9 [LLV11], proper support was only added with LLVM 3.2 in 2012 when a new backend under the name of NVPTX was directly contributed by and has since been maintained by NVIDIA [Hol12]. Using this backend, it is possible to compile any LLVM IR input to NVIDIA's Parallel Thread Execution (PTX) format [NVI15e]. This is conceptually quite similar to LLVM IR, supporting virtual registers, functions and a higher-level abstraction of hardware instructions, but is already very specific to NVIDIA hardware. PTX code can then be compiled for individual NVIDIA GPUs either through the CUDA driver API or through offline tools provided by the CUDA SDK. For this project, I'm solely making use of PTX and the driver functionality to compile it for a GPU. For this reason, the CUDA implementation of this project is much more low-level than the other two GPU targets, since I need to completely provide my own library implementation and directly call PTX instructions instead of some high-level construct as is done for OpenCL or Metal.

Metal [App15b] was only introduced last year (2014), at that time only for iOS and PowerVR hardware, with an update this year that also added OS X support and brought AMD, Intel and NVIDIA GPU support with it [Ars15]. Different to CUDA and OpenCL, Metal is not only intended for Compute, but the same language and framework are also intended for use with Graphics. Although this project currently does not target Graphics APIs, this fact should make a future progression towards that more viable (Chapter 5.1.). Metal is also based on LLVM, specifically LLVM 3.5 just like this project, but no official documentation or open source toolchain exists for it.

2.1.1. What is LLVM?

Originally starting out in 2000 "as a research infrastructure to investigate dynamic compilation techniques for static and dynamic programming languages" [Wik15i], one of its main developers, Chris Lattner, was hired by Apple in 2005, where LLVM has since then been developed further and become the backbone of many Apple system libraries and drivers (among them OpenGL, OpenCL, Metal, Quartz and CoreGraphics for software rasterization). While no longer being an acronym for "Low-Level Virtual Machine" [LLV15o], LLVM is now both the name of the umbrella project that provides this compiler infrastructure as well as the name of the compiler middle and backend part, handling optimization and hardware code generation. Initially, LLVM was only usable as a backend to GCC [GCC15] (named LLVM-GCC, later DragonEgg [LLV15f]), but starting in 2007 [Wik15c] a new project called Clang ("C-language") [LLV15c] was created that was to complete LLVM as a compiler frontend for any kind of C-based language (including C itself, C++ and Objective-C), and thus finally being able to replace GCC completely. C and Objective-C support in Clang reached production quality in 2009, full self-hosting of Clang/LLVM occurred in 2010 [LLV10] (considered complete C++98 support), and it later became the first C++ compiler to fully implement C++11 [LLV13] and C++14 [Sta13] (early and late 2013 resp.). Clang/LLVM is now a very mature compiler that has completely replaced GCC-based toolchains on OS X, iOS and FreeBSD, and is widely supported on Unix-related systems, with steady progress towards incorporating MSVC/Windows functionality as well [LLV15m]. Main contributors to LLVM projects are now Apple and Google [LP14], with other contributions stemming from Intel, AMD, ARM, IBM, NVIDIA, Qualcomm, Sony and many others.

Other projects in the LLVM family include libc++ [LLV15h] (providing a full C++11/14 STL implementation), LLDB [LLV15n] (the LLVM debugger, replacing GDB) and compiler-rt [LLV15d] (providing the run-time library support for certain Clang/LLVM tools like sanitizers, which will be shown in Chapter 3.2.10. later). Aside from the already mentioned Compute implementations, other more recent users of Clang/LLVM include Sony using it in their PS4 toolchain [Rob13], AMD using LLVM to implement their AMDGPU [AMD15b] and HSA backends (for Graphics and Compute purposes), WebKit where LLVM is used to JIT-compile JavaScript [LLV14a], and Microsoft using it in their new Clang/C2 compiler [Mic15c] (Clang frontend, Microsoft own backend).

Coming to the intermediate code representation used throughout LLVM: LLVM IR, which "is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all'

high-level languages cleanly” [LLV15j]. This basically puts it somewhere between a high-level language allowing proper types, functions and functions calls, and an assembly language, only providing basic instructions, putting everything in registers (although virtual, typed and unlimited) and limited flow control (no loops, just branching). Any compiler frontend that uses LLVM as its backend will use or has to output this type of intermediate representation, i.e. translate its input source language code into LLVM IR. The IR is then independent from any input language and can be used interchangeably. Internally, LLVM is structured to operate on LLVM IR in so-called ”passes” [LLV15p] that make it possible to work on IR on different levels. With this, it is possible to operate on IR with surgical precision on single specific instructions, or at ever more broader levels from the intra-function level up to the whole program module. Any type of optimization, analysis or general transformation of IR is handled through passes. Note also that a pass can depend on another pass (e.g. an optimization pass requiring a specific analysis pass first). This separation in passes and varying degrees of interaction with IR generally make LLVM a very modular and flexible design. Concerning data representation, LLVM IR can exist both as an in-memory representation when used with any LLVM-related library, and as a file format, either as a human-readable or as a binary representation. It is also possible to annotate IR with additional information which is called ”metadata” in LLVM. This is for example used to insert debug information, or by OpenCL and Metal to store kernel and kernel arguments information.

The LLVM and Clang projects are highly modular, with them being able to be used both as a monolithic compiler binary, but also allow their single components to be used as separate libraries, e.g. `libClangLex/libClangParse` to lex and parse C/C++, or `libLLVMIRReader/libLLVMTransformUtils` to load and process LLVM IR. These libraries are also used to create helper tools that are used to process LLVM IR in various ways, e.g. `opt` to perform specific optimization passes on IR, `llc` to compile IR to a target binary format, `llvm-as` and `llvm-dis` to convert IR between its human-readable text version and binary format, and `llvm-ar` to even create archives/libraries from multiple IR files. Similarly, the tools from the Khronos SPIR-Tools project [KG15b] are used to process SPIR files, in particular `spir-encoder` to convert LLVM 3.5 IR to LLVM 3.2 IR and make sure that SPIR specifics are set, and `spir-verifier` that can be used to verify if a SPIR file conforms to the specification.

2.1.2. Why use Clang/LLVM for this project?

At this point it should already be clear that LLVM is the only option to even realize a project like this, as there are simply no other ways of achieving this without building a proper compiler¹, and because LLVM established itself as the foundation of almost all Compute implementations (and many other areas). This is not undeserved, as Clang/LLVM are well-designed projects that are easily extendable and usable as a building block to create a new (or C/C++ based) language toolchain. And while it would technically be possible to use or create a different frontend compiler that produces LLVM IR, no other C++ LLVM-frontend besides Clang exists, and there wouldn't be much point to it as Clang does an excellent job.

To summarize why Clang/LLVM are necessary and how and why this project uses them:

- Clang/LLVM provide a state-of-the-art compiler building framework (well-designed and modern C++)
- they enable this project to implement a unified approach that allows to target everything, everywhere, with one language, library and toolchain
- high-level vendor independence and no incompatibilities on the compiler, language and library side
- less conservative: future development can directly progress with Clang/LLVM development and new C++ standards, without needing to wait on Khronos to standardize things first and wait again for vendors to finally implement
- retaining the possibility to perform hardware-specific low-level programming that isn't covered by a generic standard (only possible for NVIDIA and Host-Compute at this point, might be possible on AMD in the future as well when also targeting the AMDGPU LLVM backend that is currently in development [AMD15b], uncertain about Intel and PowerVR)
- wide range of LLVM optimizations: quite hard to beat when done by each vendor individually, vendors should rather concentrate on hardware-specific optimizations based on a fully optimized IR that LLVM provides
- whole range of Clang/LLVM tooling and niceties: proper C++14 support, useful warning and error messages, various code sanitizers, various ways to operate on and use LLVM IR, ...
- all of this is available and usable now

¹on that note: I have tinkered around with OpenCL/CUDA source code compatibility in the past, but this is only feasible by sticking to C and comes with further language limitations and problems - all in all: not useful and not even remotely a stable solution

2.1.3. Clang/LLVM Compilation Process Overview

This should give a quick overview of what happens during the compilation of C++ code.

Starting off with Clang as it is the frontend: it reads the input source code (from a file or from stdin), lexes the C++ code (as per [ISO14, 2.2]), performs pre-processing, then continues with parsing the now tokenized C++ code, meanwhile also performing semantic checking and constructing an abstract syntax tree (AST [Wik15a]). The AST is Clang's final representation of the code, containing all information of the input code and providing the general structure of the code [LLV15g]. Note that this representation is fully unoptimized and in fact a direct representation of the input code. Note also that an AST is not a necessity at this point (compilers have been build before without ASTs), but it is at least very helpful to have an AST representation when compiling C++, as the compiler needs to be able to run constexpr code at compile-time, properly handle things like two-phase lookup and it generally being a good idea to handle C++'s complexity (proving this point, even Microsoft had to finally move to an internal AST representation to build a proper standard-conforming compiler [Mic15d]). In a final step, the AST representation is converted to LLVM IR which is then handed off to LLVM.

Onto LLVM and LLVM IR, which is at this point independent from the used frontend. LLVM will perform and run any selected optimization, analysis and transformation passes [LLV15k]. Among these passes are of course function inlining, DCE (dead code elimination), redundancy elimination (GVN, EarlyCSE, ...), loop unrolling, vectorization, and many others. This is also the point where some of the platform specific passes will be run (will be explained in Chapter 3.1.4.). At the end, depending on how LLVM and the compiler are used, this will either output LLVM IR again or run target specific code generation to produce a target binary. Note that for this project LLVM IR is always the output of the first step. The alternative here is to use the `llc` binary, which is a standalone frontend of the LLVM code generation part (as used for PTX).

2.1.4. Compute Toolchain

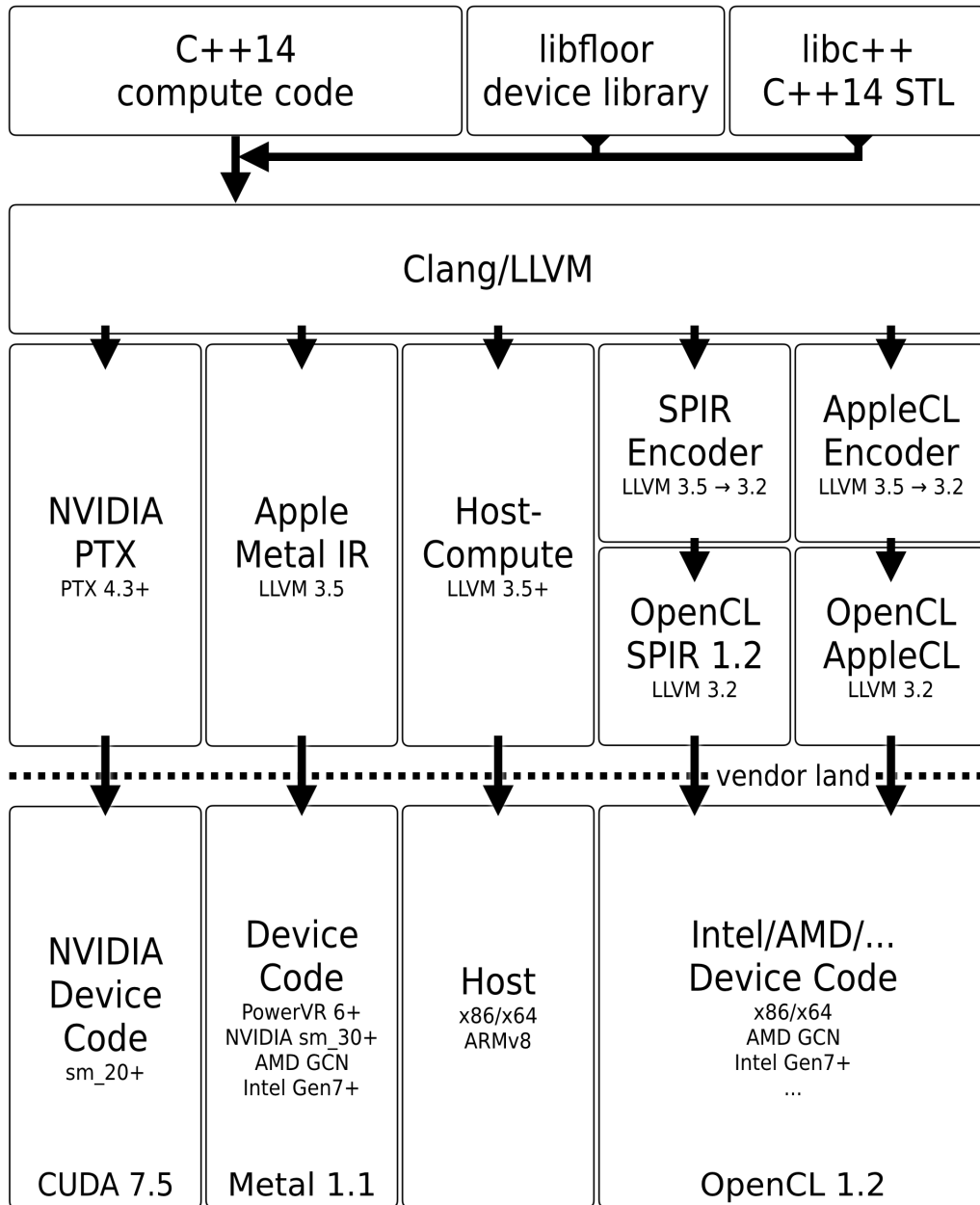


Figure 1: Compute Toolchain

The compilation process is very similar to how other compute implementations are handling it, but probably quite different to what one might be used to with OpenGL/GLSL. To compile a compute program, the user either has to specify the source code in text format in memory or by specifying a source file name (in which case it will be used directly). This is then handed off to an external binary (`clang`, the Clang/LLVM compiler binary), which in a first step will always create

an LLVM IR file. Depending on the target backend this is then further processed to varying degrees. When targeting CUDA, the `llc` binary is called (the LLVM system compiler) to transform the input LLVM IR to CUDA PTX code using LLVM's NVPTX backend (with some additional NVIDIA specific optimizations). When targeting OpenCL/SPIR, the LLVM IR is essentially in a finished state already, but has to be converted down from LLVM IR version 3.5 to 3.2 using the `spir-encoder` [KG15b], since 3.2 is the version required by SPIR 1.2. Similar is the case for OpenCL/AppleCL. Optionally it is also possible to verify the validity of the final SPIR binary using the `spir-verifier` tool. In case of Metal/AIR, only some minor textual replacements have to be performed, as both this project and Apple's Metal implementation are currently based on the same LLVM 3.5 version. Each of the compiled (intermediate) binaries is then read back to memory and then passed to backend specific functions that will produce the final binary for the respective compute device.

Which modifications that needed to be made to Clang (language extensions and platform support) and which modifications and additional passes were necessary in LLVM will be explained in Chapter 3. Implementation.

2.2. Compute

Concerning terminology in here, I will be sticking to OpenCL terminology [OWG12] as I think it is the most sensible one, aiming to represent a wide variety of hardware.

2.2.1. Platform

All Compute platforms essentially follow the same model, with the host at the top managing all compute devices below. The only difference here being OpenCL which can be comprised of multiple vendor platform implementations, whereas CUDA and Metal are only one platform and make no such distinction. The host, in form of the user application and platform libraries, is responsible for almost everything, managing memory on devices, scheduling execution on devices, compiling compute programs and providing necessary synchronization and communication with and between devices. Put differently, the sole responsibility of devices is to execute compute programs while the host directs the input and output of compute programs and handles the logical order of how programs are to be executed.

Compute devices come in various forms nowadays, from conventional CPUs, to specialized hardware like GPUs, DSPs or FPGAs. For the purpose of this thesis and project, I will however concentrate only on the major CPU and GPU vendors that currently permit programmability through external means (as already explained in the previous section). Namely, these are AMD CPUs and GPUs, Apple CPUs and (PowerVR) GPUs, Intel CPUs and GPUs, and NVIDIA GPUs. Note that in case of OpenCL CPU devices or when using Host-Compute, the compute device can correspond to the same hardware as the host, but the logical distinction between device and host remains the same regardless.

A compute device is always made up of one or more so-called compute units, equivalent to individual CPU cores, a streaming multiprocessor (NVIDIA [NVI15b]), also compute unit (AMD [AMD12]), unified scalable cluster (PowerVR [Pow14]), or execution unit (Intel [Int15c]). These are then again divided into smaller pieces of hardware called processing elements, where the actual program execution will happen, and correspond to individual SIMD lanes or ALUs. These are almost always physically grouped into larger SIMD-units of varying sizes that execute the same instruction in parallel, of course addressing different memory and program state. A compute unit can be made up of one or more such SIMD-units. Note that this hierarchy is not entirely necessary from a logical standpoint, but can probably be attributed to hardware production and design, allowing single compute units to be disabled if faulty and providing easier design scalability.

In this project, the combined host and platform part is abstracted in the so-called

compute context, providing a generic interface for all backend compute platforms and is further detailed in Chapter 3.2. Host. This makes it unnecessary for the user to know the implementation specifics and host APIs of each platform.

2.2.2. Execution

Execution on devices happens in so-called *kernel* functions, with these kernel functions being organized in compute *programs*. Kernels are specially marked functions that basically act as entry points into the program. It is generally intended to have multiple kernels in a single source file and share common functionality between them. This is also a good idea, since compiling N kernels in 1 program is faster and wastes less resources than compiling N kernels in N programs.

When specifying work, a kernel is essentially mirroring the innermost part of a single, double-nested or triple-nested for-loop, one kernel element execution corresponding to one innermost loop iteration. Another way of looking at this is that one kernel execution corresponds to either one element in an 1D array, one pixel in a 2D image, or one voxel in a 3D volume. In general, a kernel execution must be specified as a 1D, 2D or 3D range². This range is called the global work size and each individual element in it, equivalent to one kernel execution, is called a work-item. While single work-items could be executed like this, they are however always combined in so-called work-groups ranging from 1 to usually 512 or 1024 work-items at the most (this is hardware specific). The work-group size is also specified as a 1D, 2D or 3D range and called the local work size, and always evenly divides the global work size. Each of these work-groups is then processed on a single compute unit, continuously, until all work-groups have completed their work.

The reason for work-groups is both a hardware and a software one. It would be quite demanding and inefficient if only single work-items, all with their individual separate state, would need to be executed, leading to a lot more duplicated hardware than there is now. Furthermore, if all work-items would run on their own, it would make any kind of synchronization very costly as work-items would need to wait on others (also see Chapter 3.1.4.4. Host-Compute). This is not at all the case right now, as not all work-items are executed or running at once (therefore however making global synchronization across all work-items impossible). Instead, work-groups provide a middle ground solution, executing multiple work-items in lockstep on the SIMD-units of the compute unit. This allows single ALU/processing-element hardware to be much simpler and specialized control hardware to only exist on a compute unit level, and special function hardware that

²albeit that higher dimensions can be mapped in software, but hardware support is limited to 3D at the most

is rarely used to be separated and exist at a lower number than there are processing elements in the compute unit (e.g. NVIDIA GPUs [NVI15b]). This also enables synchronization between work-items in a work-group to occur at a much lower cost, as all work-items running on a SIMD-unit will already be synchronized (as they simultaneously executed the same code). Full work-group synchronization then only needs to wait on the individual SIMD-units of the compute unit. In general, work-groups and synchronization functions make it possible to efficiently implement cooperative computation using all work-items in such a work-group.

Note that there are various practically synonymous terms for *work-item*: thread, fiber, processing element, ALU, SIMD-lane. The only difference being that the latter three describe the hardware element, and the other two and work-item itself describing the logical part that is executed on that hardware piece.

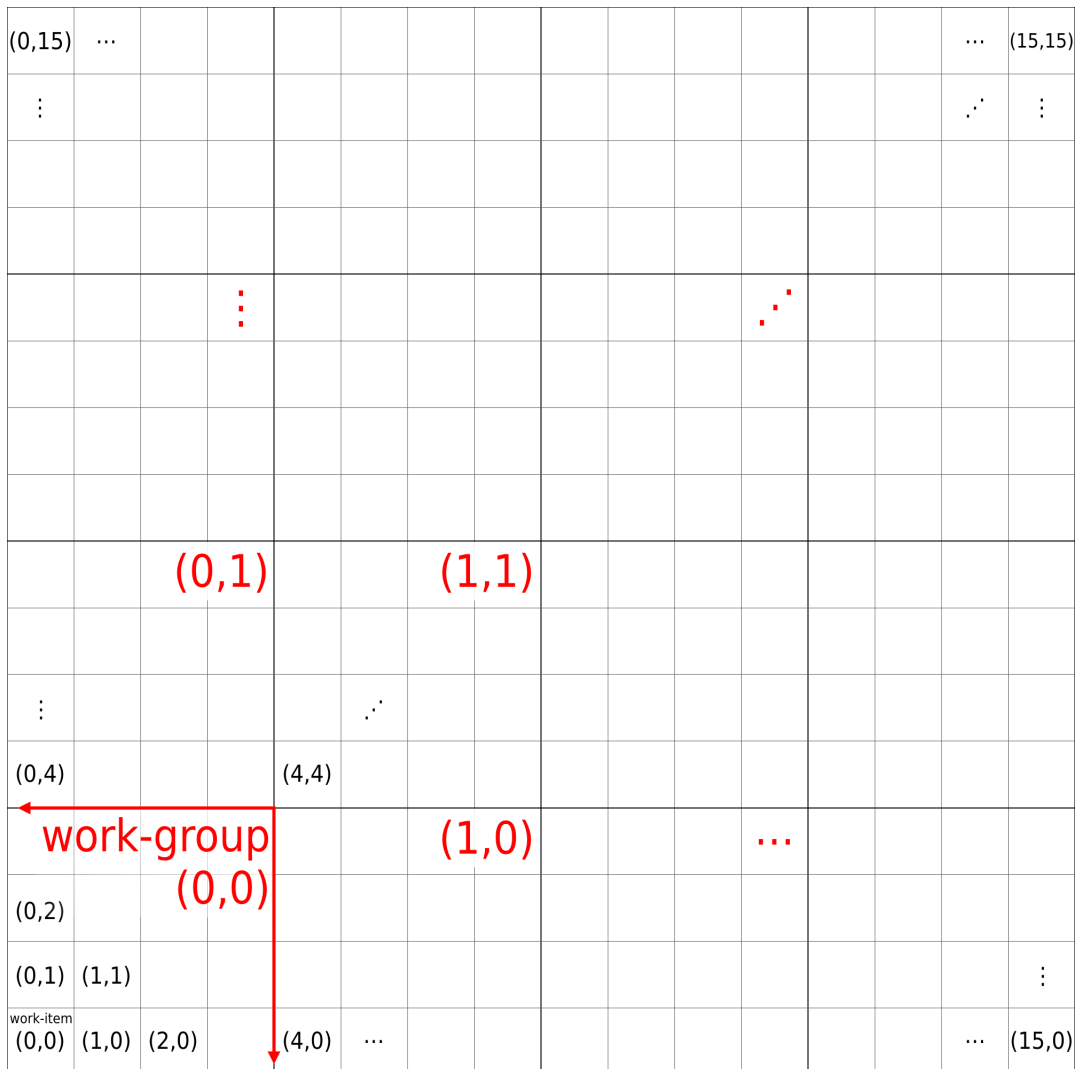


Figure 2: execution of a 2D kernel, work-item global IDs in black, work-group IDs in red, global size of (16,16) and local size of (4,4)

2.2.3. Memory Hierarchy

Instead of just a single type of memory and address space, compute devices and APIs expose 4 different kinds of memory. Namely global, local, constant and private memory, each potentially corresponding to distinct hardware. Global memory is represented by the device RAM (usually GDDR/DDR) that is read/write accessible by all work-items as well as the host. While its memory size is the largest on the device, bandwidth and latency are relatively speaking the worst among all the other types of memory, bandwidth ranging from $\approx 25\text{GiB/s}$ on very low-end hardware to $\approx 500\text{GiB/s}$ on high-end hardware [Wik15h], with latency being in the range of 200 - 400 cycles [NVI15b]. Local memory is a small amount of memory typically in the range of 16KiB - 64KiB and physically located on each compute unit (sometimes discrete hardware, sometimes equivalent to L1 cache). As such, read/write access can only happen by the work-items inside a work-group, and host access is not possible. Available bandwidth is very high and latency is low. For example, on NVIDIA hardware this is 128 or 256 bytes per clock cycle *per* compute unit (Maxwell and Kepler resp.), resulting in 128 or 256 GB/s per compute unit at a clock speed of 1GHz (and although it doesn't quite compare, this would be akin to 3 TB/s overall on a high-end device). Note that local memory is usually nonexistent on CPUs and is thus implemented via global memory. Constant memory is also a very small amount of memory, usually in the range of 64KiB - 128KiB and acts, as the name might already suggest, as constant read-only memory. This is potentially faster to access than global memory and possibly cached separately (hardware-dependent again, otherwise implemented via global memory). The host has read-write access to this (similar to global memory), while a kernel only ever has read-only access. Note that this stores both the static constant global variables of a compute program (these are transferred there before a kernel is executed) and the run-time variable constant memory buffers (managed by the user). Lastly, there is private memory which is the smallest amount of memory, but also the fastest, either comparable to or identical to registers. Each work-item has its own separate amount of private memory to which only itself has access to, with access always being read-write. Like local memory, private memory is inaccessible from the host.

All global and constant memory allocation happen on the host at run-time (Chapter 3.2.1. Host context API). Private memory (registers) and local memory allocation happen statically at compile-time. Note that dynamic local memory allocation on the host is currently unimplemented due to irreconcilable differences between CUDA and OpenCL (that being said, static allocation is to be preferred anyhow, since it allows the optimizer to make safe assumptions about the memory size, e.g. when unrolling loops - also, static offset addressing is usually faster than dynamic

addressing, which might require costly integer arithmetic at run-time).

It is also possible to map host memory to device global memory. If this is specially stored in RAM and accessed e.g. through the PCIe bus, or if it is actually copied to global device memory by the driver, is up to the implementation and hardware (integrated GPUs will probably make use of RAM directly, while discrete GPUs will either go through PCIe or cache it in global memory).

2.3. Platforms

2.3.1. CUDA

CUDA has been around the longest, being initially released back in 2007 [NVI07], and is at this time still the most efficient way to use NVIDIA hardware. Note that *CUDA* can generally describe either the Compute platform for NVIDIA hardware, the CUDA C/C++ language, or the CUDA run-time or driver API. For this project, the term CUDA stands in for the Compute platform and CUDA driver API and any code that is executed through it.

All hardware that is currently actively supported by NVIDIA is supported in this project as well. This includes Fermi (`sm_20` - `sm_21`), Kepler (`sm_30` - `sm_37`) and Maxwell (`sm_50` - `sm_53`) hardware. Image functionality is however restricted to Kepler and higher (`sm_30+`), due to reasons explained in Chapter 3.3.3. Future hardware should generally be compatible as well.

This project (and any code making use of it) does not use or depend on the CUDA SDK, i.e. it is completely standalone as it only requires the driver API functions which are queried at run-time from the system drivers. Thus, the only requirement are the NVIDIA graphics drivers (Windows/Linux) or the separate CUDA drivers (OS X), all at CUDA version 7.5 or higher. The only exception here is the offline compiler (Chapter 3.2.9.), which can optionally make use of the `ptxas` and `cuobjdump` tools from the CUDA SDK to offline-compile PTX code to CUBIN device code and allow the dumping of device specific assembly (useful for debugging and development purposes when checking and verifying what the compiler is actually doing).

2.3.2. OpenCL

OpenCL [OWG12] is special in the way that it is an open standard that is implemented by a wide range of vendors. At the current time, the only vendors that have implemented the SPIR 1.2 specification [KG14b] are however only Intel and AMD. As this project very much builds on SPIR 1.2, only Intel and AMD hardware can be supported currently. In case another vendor does implement SPIR 1.2, it should principally be supported as well. As previously mentioned, OpenCL on OS X is supported as well by emitting AppleCL LLVM IR instead, which is conceptually very similar to SPIR 1.2 (I do however consider this experimental, and it has technically been superseded by Metal on OS X).

A specialty of OpenCL is that different vendor implementations can co-exist in the operating system. This is handled through ICD libraries [OWG15c] that can

be selected at run-time by setting the specific platform index when creating an OpenCL context.

Hardware support with SPIR includes Intel and AMD CPUs (with at least SSE2), Intel GPUs (Gen7 or higher) and AMD GPUs (tested with GCN, but might work on older VLIW hardware as well). OS X additionally supports NVIDIA GPUs (although CUDA or Metal are to be preferred in general).

2.3.3. Metal

Metal [App15b] is the most recent Compute platform implementation, and unlike CUDA and OpenCL is also (or mainly) intended as a Graphics API and language. Metal is restricted to Apple software (OS X and iOS), with implementations for AMD, Intel and NVIDIA GPUs on OS X, and PowerVR on iOS. This translates to a minimum hardware requirement of NVIDIA Kepler, AMD GCN, Intel Gen7, and PowerVR Series 6.

This project is now fully based on Metal 1.1, which is the updated version of Metal that was released with iOS 9.0 and OS X 10.11.

2.3.4. Host-Compute

Host-Compute is a software Compute implementation created for this project that is entirely run on the host CPU, and is feature-wise on par with the other Compute platforms listed above. This is mainly intended as a helpful tool for developing and debugging Compute code, but can also act as a fallback Compute platform. As stated, this runs completely on the CPU in that any Compute code is compiled together with the users application code, i.e. there is no separate compilation process, and everything is treated just like any other host code. This has the large benefit that the same debugging and development tools available to other CPU host code can be used. In particular, this includes debugger support, Clang/LLVM sanitizers, profiling tools, and niceties like error and warning messages already being emitted together with other errors and warnings when compiling application code (and not at run-time later on). Additionally, Host-Compute can also interoperate with OpenGL, just like CUDA and OpenCL (Chapter 3.2.11.).

A detailed description of how Host-Compute is implemented can be found in Chapter 3.1.4.4.

2.3.5. GPU/CPU Hardware

Interestingly, over the few past years, all GPU and CPU hardware have converged to basically the same hardware and execution model. They all now essentially perform scalar operations, but across a SIMD-unit, which is in contrast to earlier GPU hardware that could execute special vector instructions (e.g. 4-component vector instructions). The same goes for CPUs as well: instead of writing specialized vector code that makes use of SIMD intrinsics, CPU software (at least with Intel's OpenCL CPU implementation [Int15a]) can now behave like GPU hardware in that parallelization automatically happens across work-items instead of only inside a work-item. This of course still uses the same vector instructions, but requires special compilation and preferably no user code interference (which would normally be the case for host CPU software). The reasons for this are quite obvious: a) there is basically no limit to SIMD-width, hardware architecture can now scale on (and between) #CPU-cores and SIMD-width, and b) this always makes full use of the hardware, instead of only using it partially when not using special vector instructions (e.g. when only using `float3`'s on a 4-wide vector architecture, this would waste 25% of performance, even worse when only using scalars or `float2`'s - these are now simply scalar instructions executed successively on an N-wide SIMD architecture).

Specifically, the current hardware SIMD-widths are: 32 for NVIDIA GPUs [NVI15b], 64 for AMD GPUs [AMD12], variable 8, 16 or 32 for Intel GPUs [Int15c], 4, 8 or 16 for x86 CPUs (with SSE, AVX/AVX2 or AVX-512 resp.), 2 or 4 for ARM CPUs (with NEON). As mentioned, for GPUs these are then grouped together to a larger group to form a compute unit (how many varies greatly with every GPU generation and vendor), for CPUs this directly corresponds to one compute unit (core).

3. Implementation

One of the major design goals of this projects is to keep the language and interfaces as standard C++ as possible, keeping additional compute-specific syntax and semantics to a minimum and generally providing standard C++ functionality, both at the language side as well as the library side as far as this is feasible. I think that this is a good idea for several reasons: a) it makes it easier to port or even directly use pre-existing C++ code, b) it makes it possible to compile Compute code on the host using an unmodified Clang/LLVM toolchain, and c) it makes it easier for anyone not knowing Compute specifics yet to learn. Note that behind the scenes there are of course a lot of non-standard things going on (albeit mostly Clang and GNU extensions), as these are necessary to implement any of this at all (Chapter 3.1.3.).

The implementation is divided into three logical parts. The compiler, needlessly to say responsible for the language part and compilation of device code, but also handling device and backend specifics that can't be dealt with on the library side. The host library, responsible for the user-side managing of computes programs, memory objects and devices, as well as providing a unified interface that speaks to CUDA, OpenCL and Metal APIs. And the device library which is responsible for Compute specific functionality on the device, alongside the more generic support facilities and C++ STL, and handles the mapping of backend specific types and built-in functionality, bridging the resp. compiler parts. The following sections are divided into these three implementation parts, explaining how and why things were implemented and how the user API looks like on the host and the device. Note that naturally certain functionality requires the cooperation of all three of these parts, which is why there are sometimes mixed explanations concerning all parts. Of course, deciding on which side to actually implement something is also a challenge, although I'm generally favoring the library side and only resort to the compiler side when it's too cumbersome or otherwise impossible to solve, as the library side is much easier to extend, exchange and reason about.

As mentioned in the introduction, the compiler is based on Clang/LLVM, with `libc++` providing the STL implementation. The host and device library are named *libfloor*. Note that all of these projects are available as open source, as detailed in Appendix B.

3.1. Compiler

This Compute compiler is based on Clang and LLVM version 3.5.2, with a full patch located at `etc/llvm35/350_clang_llvm.patch` (in the libfloor repository/folder) that can be applied on top of the Clang/LLVM release source code. This includes any changes that were made for this project, some bug fixes and functionality that were backported from Clang/LLVM 3.6 and 3.7, and some missing OpenCL-related pieces from the Khronos SPIR repository.

3.1.1. Address spaces in C++ Compute

One of the major challenges of C and C++ Compute is the use of address spaces. In ordinary C/C++ all memory is part of the same contiguous, or at least made to look like contiguous, address space, hence making all pointers plain and simple values that can be used interchangeably. With the addition of address spaces to the language, simple things start to get complicated. Pointers can no longer be used interchangeably, i.e. a pointer to global memory may not be used where a pointer to local memory is expected, as pointers could contain the same value, but point to different types of physical memory. In general, no pointer is no longer a simple value, but needs to "somehow" know to which address space it belongs. As a consequence, a language that only needed to know an address value to access memory, now needs to differentiate between that address value and the address space that is attached to it.

CUDA was probably the first to encounter this problem and NVIDIA's solution to this has been to require that all declarations of memory have an attached address space. All pointers are treated as being in a sort of generic address space, which is resolved through inference in a compiler pass by figuring out where a pointer originally came from [LLV14b][GH13a]. This will also duplicate code where necessary, as different instructions have to or should be used for different address spaces. In addition to this method, generic addressing was introduced with Fermi hardware, which allows loads and stores to any address space through generic load/store instructions instead of requiring address space specific load/store instructions (achieved by having a flat address space with a "window" for each memory type [NVI15e, 8.7.8.6.]). The upside to this is that no language modification other than tagging the initial memory declaration for local, private and constant memory is necessary, and the language tends to work just "as is". The major downsides to this approach however, are that 1) address space inference is costly and complicated, and there are cases where it can fail, and 2) generic loads/stores are expensive and can lead to huge performance losses, especially when accessing local or private memory. Nevertheless, address space inference is also the way chosen

by the NVPTX backend in LLVM (and the CUDA C++ path in Clang), which in addition to my address space specific containers (Chapter 3.3.2.) should work out well most of the time. Put differently, I'm not going to change a well working system that is optimized to be used like that, meets my requirements and that I can help along a little to do the right thing.

OpenCL took a different approach to solving this problem, as did Metal later on as it simply copied OpenCL's system (and unless specifically stated otherwise, everything mentioned in context with OpenCL in here also applies to Metal). In OpenCL, all pointers and memory declarations are required to have a specific address space attached to them, thus making it trivially obvious to the user (and the compiler!) to which address space a pointer belongs. In the language, this is achieved by making the address space part of the pointer type and prohibiting pointer assignments and casts between different address spaces. Conclusively, there are no pointers or chunks of memory that don't have an associated address space or isn't known at compile-time. The upside here is that everything is well-defined and it's quite easy for everyone involved to figure out where something points to or how memory needs to be accessed. On the other hand, the "it is part of the type" bit is a major drawback, making it impossible to write a simple function that just takes a pointer and doesn't care about the address space. This is less of a problem in C++, where it can be counteracted by using a template function, but for its originally intended language, C, this isn't an option.

```
template <typename T> void foo(T* generic_pointer); // any address space & type
template <typename T> void bar(global T* global_pointer); // fixed AS, any type
void baz(global float* global_float_pointer); // fixed address space & type
```

Listing 1: three functions taking pointers with decreasing genericity

As a result, OpenCL 2.0 [OWG15a] introduced a special "generic" address space which allows any source address space pointer to be used in functions that takes generic pointers, and hence bringing some alleviation to programming with address spaces to OpenCL C. Note that "generic" here is again dual-purpose (as it is for CUDA), primarily intended a software feature, but falling back to hardware functionality if the actually used address space can't be determined at compile-time. However, none of the "generic" parts apply to Metal, and neither can I make use of it in OpenCL, as I am targeting SPIR 1.2, which is in turn only building on OpenCL 1.2, therefore making rigid address spaces the only option at this point.

The use of address spaces brings further problems. Specifically, the interaction between address spaces and C++ objects (classes, structs and unions) and their non-static member functions. The problem lies with the implicit `this` pointer of

C++ objects that is automatically inserted by the C++ compiler for each member function call and simply doesn't know an address space. In a naive implementation where we would simply attach an address space to a class pointer, this would immediately lead to a type mismatch, as the object has an address space class type, but the class definition, with its `this` pointer, does not. More concretely, the generation of member functions only depends on cv-qualifiers (`const` and `volatile` [ISO14, 9.3.1.4]) and ref-qualifiers (`&` and `&&` [ISO14, 9.3.1.5]), which in turn also give the `this` pointer a very concrete semantic meaning. While it might seem prudent to simply extend this behavior to address spaces (thus expanding on the address part of [ISO14, 9.3.2.1]), I strongly disagree with such an extension, as it adds new semantic meaning to `this`, i.e. allowing code to differentiate between address spaces. I don't think this is a good or obvious enough reason (like `const` is), potentially confusing users with code behaving differently depending in which address space an object is located. It also adds the risk of new errors and is a burden on maintainability, as most code will simply be duplicated for each address space. Furthermore, it is unrealistic to think that hardware can actually handle address space memory differently, or as general as adding address space semantics might make it look out to be. Hardware is limited to mere load and store instructions from anything other than private address space (with very few, very specific exceptions like atomic functions), always needing to load data to private address space first (registers), then operating on that data and possibly storing data back to the address it came from [NVI15e, 8.1].

So how to solve this issue? Maybe more importantly, how to solve this in a way so that it "just works", without requiring large-scale modifications in possibly pre-existing user code or driving users mad with superfluous and repetitive syntax. After all, to keep the language as standard C++ as possible is one of the major requirements. This also affects any pre-existing source code, like the STL, and it would be very nice if changes can be kept down to absolutely necessary parts that are solely imposed by hardware restrictions. This problem and solutions to it have obviously come up before. There are essentially two sides to where and how this problem can be solved, with different solutions on either side (and further consequences). As detailed above, one option is to fix it on the language side by introducing new syntax and semantics. The other, possibly preferable option is to fix it entirely on the compiler side without needing to touch the language itself. The former has been shown to work before in AMD's initial exploration of bringing C++ to OpenCL [GH13b], as well as its current incarnation in Metal C++ [App15c]. Again, I don't think this is a good solution, leading to excessive code duplication and introducing more problems than it solves. Another quite telling reason might be that the current OpenCL C++ standard [OWG15b] doesn't do this either, but instead follows the road that CUDA has set before. To investi-

gate how it can be solved by only touching compiler internals, let's look again at what we actually need: a way to access the data of objects located in different address spaces in a generic way, i.e. partially templating the `this` pointer type. Technically, this could be solved by treating every member function as a template function that is instantiated for the respective address space on use, but this would incur a language change again and meddling with template semantics is even worse of an idea. Additionally, it would still duplicate the complete function for each address space, although now this would only happen inside the compiler, but still not ideal. My solution to this might seem trivial: simply do what the hardware and all software solutions to this are actually doing. Load the object data from the respective address space to private address space (which is the default and unmarked address space when using OpenCL or Metal), essentially creating a temporary object, call the member function with that temporary instead (which can now simply continue to be a single function without an attached address space) and, if modified [LLV15], store the temporary object back to the original address space pointer. To accomplish this, recall that C++ code with Clang/LLVM is transformed to a low-level representation, LLVM IR, where in this case, all member functions are simple functions with the first argument being the `this` pointer to the object. In a first step, all places where `this` pointers are inserted and member function are being called are modified by removing the address space check (so that no type mismatch errors occur any more) and instead of stripping the address space from each object pointer (which would happen normally), keep using the original address space of the object, consequently calling the member function with the first argument being a pointer that has the used address space still attached. Note that at this point the LLVM IR is invalid (we moved the type mismatch from C++ to LLVM which also has a type system). This is why, at the very first opportunity, a LLVM pass called `AddressSpaceFix` is run, which will finally perform the insertion of load and store instruction to and from a newly created temporary object at each member function call site.

Trivial example (corresponding C++ code written in the comments):

```

struct vec3 {
    float x, y, z;
    void normalize() {
        float inv_len = rsqrt(x * x + y * y + z * z);
        x *= inv_len; y *= inv_len; z *= inv_len;
    }
};
kernel void normalizer(buffer<vec3> vectors /* vec3 pointer to global memory */) {
    vectors[global_id.x].normalize();
}

```

Listing 2: simple example that calls a member function and makes use of `this`

```

; => global vec3* vecptr = &vectors[global_id.x];
%vecptr = getelementptr inbounds %struct.vec3, @vectors, i64 %gidx
; => vec3::normalize(vecptr);
tail call spir_func @_ZN4vec39normalizeEv(%struct.vec3 @vectors, i64 %gidx)

```

Listing 3: the "broken" LLVM IR with a wrong `addrspace`

```

; => vec3* tmp = "allocate a vec3 object";
%tmp = alloca %struct.vec3, align 8
; => global vec3* vecptr = &vectors[global_id.x];
%vecptr = getelementptr inbounds %struct.vec3, @vectors, i64 %gidx
; => vec3 tmp_load = *vecptr;
%tmp_load = load %struct.vec3, @vectors, align 4
; ... omitted: copy tmp_load to tmp (assemble tmp from tmp_load) ...
; NOTE: tmp is a pointer and tmp_load is just an object,
;       but the normalize function call requires a pointer
; => vec3::normalize(tmp);
call spir_func @_ZN4vec39normalizeEv(%struct.vec3* %tmp)
; ... omitted: assemble tmp_store from now modified tmp ...
; => *vecptr = tmp_store;
store %struct.vec3 %tmp_store, @vectors, align 4

```

Listing 4: the "fixed" LLVM IR, loading the object from global memory to a private object, then storing the result back to global memory

```

%vecptr = getelementptr @inbounds, %struct.vec3, @addrspace(1)* %vectors, i64 %gidx
; after inlining, can use loaded object directly now
%tmp = load %struct.vec3, @addrspace(1)* %vecptr, align 4
; => float x = tmp.x; ...
%x = extractvalue %struct.vec3, %tmp, 0
%y = extractvalue %struct.vec3, %tmp, 1
%z = extractvalue %struct.vec3, %tmp, 2
; => float inv_len = rsqrt(x * x + y * y + z * z);
%xx = fmul fast float %x, %x
%yy = fmul fast float %y, %y
%xx_yy = fadd fast float %xx, %yy
%zz = fmul fast float %z, %z
%dot = fadd fast float %zz, %xx_yy
%inv_len = tail call @spir_func, float @_Z5rsqrtf(float %dot) #2
; => float dst_x = x * inv_len; ...
%dst_x = fmul fast float %x, %inv_len
%dst_y = fmul fast float %y, %inv_len
%dst_z = fmul fast float %z, %inv_len
; => vec3 tmp_store { dst_x, dst_y, dst_z };
%tmp_store_0 = insertvalue %struct.vec3, undef, float %dst_x, 0
%tmp_store_1 = insertvalue %struct.vec3, %tmp_store_0, float %dst_y, 1
%tmp_store = insertvalue %struct.vec3, %tmp_store_1, float %dst_z, 2
; => *vecptr = tmp_store;
store %struct.vec3, %tmp_store, @addrspace(1)* %vecptr, align 4

```

Listing 5: the final LLVM IR with inlining applied

3.1.2. Language Restrictions

As with every other Compute language, restrictions on C++ have to be imposed either because of graphics hardware limitations, certain limitations in backends, or simply because they don't make much sense on Compute hardware. Further difficulties arise from backends handling things quite differently which then need to be unified in some way, as well as making it implementable with largely standard C++ as is required for Host-Compute.

The following restrictions apply, listed with the reasoning behind them.

- no exception support: this is not possible due to software and hardware support simply not existing, i.e. there is no such thing as a call stack. Also, how to handle the logic behind throwing and handling exceptions with a lot of concurrent execution going on seems not be solvable in any efficient way. Its usefulness in Compute is also very debatable.
- no dynamic memory allocation: one of the more interesting ones, since this is technically possible with CUDA and I could implement this myself by pre-allocating a chunk of memory, always passing this buffer to every kernel and handling allocation and deallocation with some atomics (globally visible). The

cost of allocation/deallocation would of course be relatively high, but might be still be very helpful for certain use cases. Either way, this isn't supported right now, but might be an interesting future extension.

- no (dynamic) function pointers: this is simply impossible with OpenCL and Metal right now as everything is inlined after all. It is however possible to use static function pointers to some degree when the compiler can properly determine the destination and later inline this function.
- no virtual functions, abstract classes (vtable): same as function pointers. Under very specific conditions the compiler can however devirtualize these (when there is only one possibility/destination), but it should not be relied upon.
- no RTTI (run-time type information): this would need some kind of run-time library support, which isn't there and would be rather costly (lookup table, storing type information, ...). Again, usefulness is unclear, especially with the amount of overhead required, and with no polymorphism being supported, `dynamic_cast` is furthermore unnecessary, thus only leaving `typeid`.
- no recursive functions: same issue as before, no call stack, no function pointers, but: this can at least be partially emulated through templates via `template <size_t depth = 0> void fun() { fun<depth + 1>(); }` with a static termination condition. Note that this is for example used in the path tracer demo (Chapter 4.2.1.) and apparent cause of much trouble, due to rather large and complex code that is generated, but can still be useful when used with care.
- no VLAs (variable length arrays): would require runtime support and dynamic memory allocation, thus not possible (also: not even standard C++, although being discussed for proper inclusion in a future standard).
- global memory objects (buffers and images) can only be passed to the kernel as explicit and individual kernel function parameters, i.e. it is not possible to store references or pointers to these inside another memory object that is passed to the kernel, or inside any other kernel parameter. This would make a unified address space absolutely necessary, which is simply not the case right now. Also not supported by either OpenCL or Metal.
- it is not possible to statically allocate global memory in device code, this must always happen on the host. Again, simply not supported by anyone right now, and non-trivial to work around (might be possible to implement by analyzing the complete program, put all allocations inside a big buffer, re-route all accesses to this buffer (the difficult part), and then automatically

allocate the buffer on the host and pass it to the kernel).

- the use of standard library headers and functionality is restricted to the ones that are explicitly supported (Chapter 3.3.5.). With the restrictions listed above this is obviously necessary, as these headers make use of these and are thus not directly compatible without modifications.

I have also looked at existing language restrictions of OpenCL and Metal and determined that some of these are mere software restrictions that have nothing to do with the underlying hardware. These restrictions were hence lifted.

- inheritance: allowed and possible, as long as no virtual/abstractness is used. This is completely prohibited in Metal C++, but obviously very useful (e.g. used in `<functional>` or the device image implementation).
- bit fields: if no dedicated bit field extract/insert instructions exist (which was probably the major reason why they are prohibited in OpenCL 1.2), these can still be computed via left/right shifts and AND/OR bitwise operations, which must always be supported.
- `goto`: especially without exception support, this might come in handy, and there are certainly other valid reasons to use it. In the end, it's all just branches in LLVM or jump/branch instructions in backends, so no reason not to support it.
- variadic macros: I have no idea why OpenCL would forbid these, as this is utterly a software feature and already completely processed at compile-time.
- global, namespace and class `static const/constexpr` variables: OpenCL and Metal would usually require these to be declared `constant` as well (putting them in the constant address space), but as these are already known to be `const`, the `constant` is redundant and it would also be quite ridiculous having to write `static constexpr const constant int global_var = 42;`. `static const` variables are thus now implicitly `constant`. This is also a very practical solution for existing code that won't have to be modified in regards to this.
- read/write images: not at all or not directly possible anywhere, i.e. images must either be write-only or read-only (Chapter 3.3.3.), but are obviously very useful for any kind of in-place image processing. This was worked around by supplying both a read-only and a write-only image to the kernel, but presenting it to the user as a single image object. Note that the "read first, write last, at the same pixel" restriction still applies.

3.1.3. Language Extensions & Modifications

As already stated in the beginning, one of the explicit goals was to keep required language extensions to a minimum and preferably handle things through library functionality or implicitly inside the compiler. As such, the amount of extensions is quite low:

- **kernel** (keyword): signals that a function is a kernel function. This is necessary so that the compiler can create and export certain information about kernel functions which are required by compute backends (number and types of arguments, among others). Note that the kernel return type must always be **void**.
- **global**, **local** and **constant** (keywords): used to signal that memory is located in a specific address space. Should however use wrapper containers in almost all cases (Chapter 3.3.2.). Note that no **private** keyword exists (besides the C++ class one), as thus-unqualified variables are implicitly considered to be in the **private** address space.
- aggregate and opaque image types: these are used with OpenCL and Metal, and will be properly explained in Chapter 3.3.3. This is mostly for internal purposes, as the user won't actually see opaque image types, and aggregate image types are created implicitly by placing opaque image type objects inside a **struct** or **class**, which a user won't do either.
- image access and sample type attributes: again, not visible by the user and only used internally inside the device image implementation.

I had no influence on these, but they are worth mentioning here because they proved to be quite useful and are used throughout this project:

- **#pragma clang unroll**: while loops will already be unrolled as-is if the compiler thinks this is a good idea, it is sometimes necessary or desirable to (fully) unroll even large and/or complex loops (as will be shown in Chapter 4.2.2. N-body simulation).
- **__attribute__((enable_if(constexpr-condition, "message")))** (function attribute): creates a function overload that can be enabled or disabled depending on the "constexpr-condition" which can make use of function arguments if they are known at compile-time. This is a crucial feature that is required to select between constexpr and run-time math functions depending on if a function argument is truly constexpr or not (as used in Chapter 3.3.1.). Another useful application is the ability to test if an array access is out-of-bounds when using constexpr array indices (can already error at compile-time

instead of failing at run-time).

- `__attribute__((const))` (function attribute): signals "function returns the same result if given identical input, and is side-effects free". These are important from an implementation standpoint as they allow external functions to be optimized away if their result goes unused, even though the compiler has no knowledge of the functions body. Since it is however declared side-effects free, it can thus be eliminated if unused (for example, this is used on all image read functions and external math functions as they can safely be removed if their result is never used).
- `__attribute__((noduplicate))` (function attribute): signals that a function (call) can not be safely duplicated for optimization purposes. This could for example happen when moving a function from the outside of an if-else branch to the interior of both the if and the else branch, thus duplicating the function call. This is vital for synchronization functions as compute implementations have very strict requirements about how and when these may be encountered, most importantly the requirement that a synchronization function must be encountered by all work-items in a work-group at the same point, which would obviously be broken by duplicating and moving the function call to the interior of two separate branches.

3.1.4. Platform-specific Information & Idiosyncrasies

This section provides an overview over what modifications to the Clang/LLVM compiler needed to be made for each platform, and a detailed explanation of how Host-Compute is implemented.

On a side note, as this is used by this project, Metal and CUDA: it is possible to declare a C function with an ASM label for which an implementation is then provided inside the compiler (LLVM). This is one of the possible ways to create an "intrinsic function". For Metal, only these kind of function forward declarations need to be written and Apple's Metal compiler will take care of the actual hardware-specific implementation. CUDA in LLVM provides a different type of intrinsic function, a kind of built-in function that can be used as-is and doesn't need to be declared on the library side. It is however implemented in the same way as the other type. OpenCL doesn't use intrinsic or built-in functions, but uses normal C function declarations instead.

3.1.4.1. CUDA CUDA was probably the least demanding backend to support, as CUDA/PTX support in Clang/LLVM was already very mature. Only 3 bug fixes needed to be made over the course of this project (2 backported from LLVM 3.6 and 3.7 resp., one easily fixed by myself, but also fixed in LLVM 3.6, though incompatible to 3.5). Support for newer PTX and device versions was also added, up to unreleased PTX 5.0 and `sm_70`, and defaulting to PTX 4.3 and the respective device `sm_*` version for now. Note that PTX is generally forwards-compatible to future hardware and software (the only thing that changes is the version number), while compiled PTX code (CUBIN) is specific to the GPU (or GPUs) it was compiled for.

As the CUDA target in Clang/LLVM is intended to be used like NVIDIA's `nvcc` CUDA compiler [LLV15e], i.e. supporting both host and device code in the same source file, appropriate changes were made so that all code is always treated as device code. This would have otherwise required that all device specific code (which is *all* code) would have needed to be marked with an `__device__` attribute. Downside to this is that CUDA's understanding of host code is no longer supported, but this was never the intension of this project anyways. It should also be said that the design decision to explicitly mark functions as `__device__` and/or `__host__` is not a particularly good one. The path this project has generally chosen is to handle these things implicitly and support the same source code on both the host and the device, and although incomplete at this point due to device restrictions (Chapter 3.1.2.), a future extension to the compiler could "simply" ignore unsupported C++ features for device code as long as it isn't actually used by devices.

Due to the complexity of CUDA/PTX texture and surface instructions, either requiring thousands of lines of code of trivial inline assembly or (pre-existing) intrinsic function calls, neither of these being manageable with templates and far from anything that can be considered maintainable, an additional LLVM pass (`CUDAFirst`) and some additional LLVM intrinsic functions were added (similar to how OpenCL and Metal handle this). This way, the intrinsic functions can be used with templates and the actual logic to handle texture and surface PTX instructions can be written in plain C++ (both much more maintainable and easier to extend). This pass is run before all other LLVM passes (hence "First") and transforms these intrinsic function calls to the appropriate PTX inline assembly. More information about how images are handled in CUDA can be found in Chapter 3.3.3.

Note that CUDA is the only backend (besides Host-Compute of course) whose final output isn't LLVM IR, but NVIDIA's own PTX format. This makes it however a lot easier to support newer LLVM versions as no conversion of LLVM IR between LLVM version needs to happen (and the modifications of this project could quite easily be ported to LLVM 3.6, 3.7 or even trunk).

3.1.4.2. OpenCL Khronos originally developed OpenCL/SPIR support separately from the main Clang/LLVM repository [KG15a], at first based on LLVM 3.2 for SPIR 1.2, later on 3.4 for SPIR 2.0. This was then gradually merged into main Clang/LLVM, up to where version 3.5 now contains most of the necessary SPIR 1.2 support. As this is intended for OpenCL C99, several modifications needed to be made to make it work with C++ (generally disabling OpenCL C specific checks or handling OpenCL specifics in C++), also lifting some of the software restrictions as already mentioned.

OpenCL/SPIR require that kernel function information is present in the LLVM IR metadata, i.e. how many kernel arguments there are, what their types are, which address spaces they are in, and what kind of images and access qualifiers there are. As this was incomplete in Clang 3.5.2, code from the mentioned Khronos repository had to be merged in and some of the remaining issues had to be fixed by myself. Support for aggregate images was added as well. This is now at a point where it successfully passes `spir-verifier` checks and properly compiles with Intel and AMD compilers (which it did not before even though this is actually an optional feature of SPIR).

SPIR 1.2 does differ from LLVM 3.2 IR to some extent in what kind of instructions are supported [KG14b, 3.3]. As neither the Khronos code nor the Clang/LLVM code had handling for this, an additional LLVM pass called `SPIRFinal` was added that handles these incompatibilities at the IR level, and as the name might suggest is run as the final LLVM pass, as any other pass might introduce incompatible instructions again. Some of the unsupported instructions can be converted to supported ones (e.g. vector type casts to scalar casts), while others are completely unsupported and will now throw compile-time errors (these are however not encountered under normal circumstances). Note that I do not understand why Khronos chose to break compatibility with LLVM IR for instructions that can clearly be supported by other means. Even more bizarre is the fact that some of the unsupported instructions work fine for some OpenCL targets/compilers, while it didn't for others (or: one specific one, Intel's GPU compiler). Either way, this is not something that should have been handled at this stage of the compilation, but rather at each vendors backend compiler.

OpenCL depth and MSAA image type support was added as well, as this was still missing (technically extensions and not part of OpenCL 1.2, but officially mentioned in the SPIR 1.2 specification and supported by vendors).

The `AddressSpaceFix` LLVM pass mentioned in Chapter 3.1.1. is run after the initial analysis and function attributes analysis passes.

To support Apple's OpenCL LLVM IR (AppleCL), the same SPIR code path is

used. This only differs in the way how kernel metadata has to be written and in how image types are named. As no documentation for this existed, this required some light reverse engineering like Metal (as in: compile OpenCL source code with Apple's toolchain and look at the textual metadata representation). The general type of information is however quite similar to SPIR and straightforward to understand (anything AppleCL related in `CodeGenFunction.cpp` in the Clang source code). Some nuisance was presented by the necessity to also emit some of this information (redundantly) as a global variable in LLVM IR, as it required additional kernel function analysis (extracting all kernel argument information again, but also emit additional local memory allocation information). Note here that AppleCL is only supported because I added support for it before Metal became available on OS X (and Metal should generally be preferred there now).

The standardized OpenCL device library functions can simply be used by providing function declarations for them (with C++ name mangling for SPIR, C "mangling" for AppleCL) as they have to be provided and implemented by each OpenCL vendor, since they are of course hardware dependent (accessing images, math functions, ...).

3.1.4.3. Metal Apple's original Metal release in 2014 and update this year are heavily based on LLVM 3.5. Conceptually it is very similar to OpenCL/SPIR in that it first compiles to a common LLVM IR format with generic intrinsic functions still present (no hardware specifics yet), which is also used as a binary and library format, and only compiles to device specific code at run-time using again some LLVM-based vendor compiler. Different to OpenCL/SPIR, Apple provides and handles all vendor compilers themselves and retrieval of device code is at no point possible. Apple call their type of LLVM IR and intermediate format AIR, probably short for Apple IR. As it is based on LLVM 3.5, it also almost directly compatible to the vanilla Clang/LLVM 3.5 LLVM IR used in this project.

Since the Metal compiler code is not open source, Metal compilation would have had to be implemented "from scratch". As it is however almost identical to OpenCL, I decided to use the same code path in LLVM that is also used for OpenCL/SPIR (including the same C++ support of course). Additional code is then comparable to what I needed to do for SPIR and AppleCL. On the device library side things were also quite straightforward with device functions being "documented" in Metal device library header files as intrinsic functions which can simply be used and are passed through to Apple's backend compiler.

Like OpenCL targets, Metal also requires kernel metadata of more or less the same type as SPIR, but again textually different to it, and as with AppleCL completely

undocumented, again requiring some mild reverse engineering. The only challenge here was that it also requires the sample type of images (`float`, `int`, `uint`), which is not the case for SPIR, and hence necessitated the addition of another attribute (type-dependent) in the compiler so that it can be properly set and tracked at compile-time (all handled behind the scenes in the device image implementation).

Because Metal does not support global variables or functions to perform ID-handling (Table 11), but rather requires the manual addition of special ID-handling parameters to each kernel function, additional intrinsic functions acting as placeholders and an additional LLVM pass called `MetalFinal` were added. The pass automatically adds these ID-handling parameters to each function and replaces the placeholder function calls with the respective ID variable. Note that it is also necessary to put this into the kernel metadata, which is done separately in `CodeGenFunction.cpp`.

Oddly, the same LLVM instructions forbidden by SPIR also cause issues when used with Intel GPUs in Metal. Because of this, the `MetalFinal` performs the same invalid to valid instruction transformations like the `SPIRFinal` pass does.

Also added were additional image types that are supported by Metal, but not present in OpenCL (various cube map types). All image types are also named differently in Metal.

3.1.4.4. Host-Compute The major difficulty of implementing Host-Compute is the restriction that it has to be able to compile using a vanilla Clang/LLVM toolchain, i.e. has to be completely implemented on the library side and can't modify the compiler. Furthermore, it must be able to compile using an unmodified STL implementation (LLVM's `libc++`, GCC's `libstdc++` and MSVC's STL are supported and tested) and other system and C standard library headers without choking on them. Except for a minor nuisance with the `<locale>` header which defines a function named `global`³ this actually works remarkably well. This probably owes to the minimalistic keyword approach simply not giving much opportunity for naming conflicts, as well as the fact that STL libraries are meticulously designed not to conflict with user code. System and C library headers are a different story, but I have not yet encountered a problem with these either. Note of course that I can't prevent kernel code from using functionality that is only available to the host and not to actual compute devices, i.e. just because it compiles on the host doesn't mean it will also compile for devices (although it's usually a good indication). On the other hand, I wouldn't want to prevent it either as it allows any host code to be run for whatever (unforeseen) reason there might be.

³luckily a rather rarely used function; the cross section of code using C++ locale functionality and Compute in the same source file should also be very low - if you do require this function, it has been aliased to `locale_global`

Despite the fact that Host-Compute is not intended for high performance Compute execution on the CPU, I still made sure that it runs adequately fast and exploits compiler and run-time functionality as far as it can when using a vanilla compiler. While not being able to keep up with Intel's OpenCL CPU implementation (although same order of magnitude), it interestingly outperforms AMD's and Apple's OpenCL CPU implementations. For some statistics have a look at Chapter 4.2.2. N-body.

Starting with the obvious: multi-threaded execution, running as many threads as there are logical CPUs and pinning each thread to a separate logical CPU core (removing or limiting costly core context switches). As work is already split into global and local sizes, work distribution to threads becomes relatively straightforward and happens dynamically at run-time: a simple *request/claim chunk of work - compute - repeat until done* loop using a single atomic counter.

I already noticed early on that performance is severely reduced when using synchronization functions of the Compute library (like `barrier`, Chapter 3.3.2.), as these require the execution of multiple work-items up to a certain point in the program and the signaling that they have done so. As synchronization functions are a very important utility in implementing cooperative computation across work-items and should thus be very fast (otherwise there would be no point to them), I put importance on getting these as fast as possible.

Using any of the usual suspects of the multi-threading toolbox like mutexes, conditioning variables or even atomics all proved to be inadequate. This is mostly due to the sheer scale of things (easily a few million sync calls for e.g. N-body or when performing some kind of synchronized image processing), but also due to how threads are executed on modern day operating systems (there is no guarantee on when threads are actually run, thus need to wait on the slowest thread/work-item), making even overhead of mere microseconds per call unacceptable at that scale. The next idea was to use POSIX `ucontext` and related functions [IOG04], something that is more akin to programming with fibers [Wik15d], i.e. no longer relying on the operating systems to handle thread execution (at least on the small per work-item scale), but do this manually. However, this again proved to be inefficient as it lead to the situation where actual kernel code was executed for about $1\mu\text{s}$, which was then followed by $100\mu\text{s}$ spent in syscalls just to switch to a different fiber.

The next approach, and now final solution, is to manage these fibers and their associated registers and stack completely in user-space, thus minimizing fiber switching overhead to writing and reading a few bytes to and from memory. This however requires a very low-level ASM implementation that depends on the platform ABI

and CPU, and is for the moment only implemented for the x86-64 SysV ABI [MHJM13] (working on all Unix-based and related x86-64 platforms, e.g. Linux, OS X, FreeBSD). To perform a fiber switch under the x86-64 SysV ABI, only the callee-saved registers `rbp`, `rbx`, `r12 - r15`, the stack pointer register `rsp` and the return address / instruction pointer register `rip` must be saved. All in all 8x 64-bit of memory that need to be written for the switched-from fiber and read for the switched-to fiber (64 bytes each). The max work-item (fiber) limit per thread has been set to 1024 here. On Windows, a different more high-level approach is used, namely the Windows Fiber API [Mic15a]. Conceptually this works in a very similar way (although implementation details are unknown), but is somewhat more limited in that it has more overhead (about 3x to 4x slower) and it is tricky to find an upper limit on how many fibers can actually be spawned within a thread (for the moment, I have chosen to set this limit at 64 fibers per thread as this seems to be working safely). In the future, this should probably also use a more low-level implementation to reach equal performance. On all other platforms, POSIX `ucontext` is used as a fallback option (alternatively it is also possible to always use plain single-threaded execution - this must however be selected when compiling libfloor). Note that each fiber (work-item) has access to a stack of at least 8 KiB, which should generally be more than enough for Compute.

From the top down, a kernel is now executed as follows: a fixed set of worker threads is started (corresponding to the logical CPU count), with each thread pinned to a different logical CPU (this can be done on all host platforms). Next, each thread continuously grabs an unprocessed work-group and fully executes all work-items in this work-group, until all work-groups have been processed. Work-group distribution is achieved through a simple atomic counter that represents the group ID. Inside each thread and for each work-group, the thread will spawn as many fibers as there are work-items in the work-group (each fiber representing one specific work-item and global ID). These fibers are executed sequentially inside the thread, with fiber switching occurring either on termination when the kernel has been fully executed for a work-item, or when encountering a synchronization function like a barrier. Fiber switching will loop once over all fibers (work-items), run each fiber up to the same fiber switching point, until all fibers have executed the kernel code up to the same point in the program. This way, synchronization is accomplished without needing to wait on someone else and with minimal overhead.

With this method it is now possible to execute kernels in a similar fashion to GPUs, i.e. usually preferring many work-items per work-group (512 or 1024 seem ideal). This is in contrast to OpenCL CPU implementations which usually prefer as many work-items as there are logical CPU cores or a value corresponding to the SIMD-width (also see Figure 4).

Note that all vectorization that LLVM will perform happens at the work-item level and can not happen across work-items (as is done for Intel's OpenCL CPU implementation). This is not easily solvable as it would require special compiler support and modifications, hence no longer being able to use a vanilla Clang/LLVM compiler (for which it is intended). Also, when compiling Host-Compute code, appropriate compile-time flags have to be set to enable code generation for a specific SSE or AVX version (e.g. `-msse4.2` for SSE4.2, or `-mavx2` for AVX2) - libfloor and related source code enables SSE4.1 by default.

On the library side, as already mentioned, the normal STL and system headers and functionality can be used (for e.g. run-time math functions, or `printf`). Compute-specific functionality needed to be implemented separately of course. This includes the aforementioned handling of synchronization functions, but also ID handling (handled via globally accessible `thread_local` variables, and by each thread and fiber knowing its own ID). Image functionality will be explained in Chapter 3.3.3. Atomic functions are handled by either aliasing them to the corresponding C11 atomic function, or by using a compare-and-exchange fallback for operations that aren't directly supported in hardware. Address space keywords are completely removed at compile-time (via empty `#defines`). Kernel functions are marked as `extern "C"` so that function names are not mangled, and additionally marked `__declspec(dllexport)` on Windows. This way, kernel functions can easily be retrieved at run-time via `dlsym` or `GetProcAddress` functionality, thus being able to use similar functionality as is used for compute programs on all other Compute platforms (where kernels are retrieved through their function name at run-time).

3.2. Host

The *Library* sections provide a description of the Host API and explanation of some of its inner workings. The source code to this is located in the `compute` folder, with backend specific implementations in the sub-folders `cuda`, `host`, `metal` and `opencl`. The remaining sections detail other functionality or general information that is relevant to the host.

3.2.1. Library: context

`compute_context` is the base interface that holds all compute related things together and is used to create almost all compute objects listed here, with the only exception being `compute_kernels` which are indirectly and automatically created through `compute_programs`. Note also that this class, like all other compute classes, is implemented separately for each compute backend. While the general concepts are pretty much the same for all backends, the actual details and API calls are of course very different, hence separate implementations with only minor parts of truly generic code, but being able to support a shared common interface.

```
class compute_context
```

Since `compute_context` is an abstract class, it must thus be constructed through any of the inheriting classes listed below. The recommended way of construction is to use `auto ctx = make_shared<cuda_compute>()` (or any of the other implementations). Alternatively, the default context that is created automatically depending on the used OS and config settings can be retrieved via `floor::get_compute_context()`.

constructor:

```
cuda_compute(const unordered_set<string> whitelist = {})
```

Constructs a `compute_context` for use with CUDA. If the optional `whitelist` parameter is specified and non-empty, the device name of any discovered device on the platform will be compared against all elements of this list and will only be used if at a least a partial match exists (comparison is always lower-case).

constructor: `host_compute()`

Constructs a `compute_context` for use with Host-Compute. Straightforward constructor with no whitelist, as only one CPU type exists.

constructor:

```
metal_compute(const unordered_set<string> whitelist = {})
```

Constructs a `compute_context` for use with Metal. The `whitelist` parameter incorporates the same functionality as the one in `cuda_context`.

constructor:

```
opencl_compute(const uint64_t platform_index = ~0ull,
               const bool gl_sharing = false,
               const unordered_set<string> whitelist = {})
```

Constructs a `compute_context` for use with OpenCL. Since multiple OpenCL implementations (platforms) can exist in the OS, the `platform_index` specifies the index of the platform that should be used. Note that if a platform implementation is non-viable (no SPIR support, no device matching the whitelist, or any other problem), it will automatically try all other OpenCL platforms on the OS until a viable one has been found. The `gl_sharing` parameter specifies if the OpenCL context should be created with OpenGL/OpenCL interoperability support (this has influence on which devices in the platform can be used). The `whitelist` parameter incorporates the same functionality as the one in `cuda_context` and additionally supports the special names "CPU" (to only allow CPU devices) and "GPU" (to only allow GPU devices).

```
bool is_supported() const
```

If true, the `compute_context` was created successfully and can be used.

```
COMPUTE_TYPE get_compute_type() const
```

Returns the used `COMPUTE_TYPE` (backend type) of the `compute_context`. Possible return values are CUDA, HOST, METAL and OPENCL.

```
const vector<shared_ptr<compute_device>>& get_devices() const
```

Returns an array of all usable `compute_devices` in the `compute_context`. If `is_supported` is true, this will contain at least one device.

```
shared_ptr<compute_device>
```

```
get_device(const compute_device::TYPE type) const
```

All `compute_devices` have a distinct type attached to them. This includes whether the device is a CPU or GPU and the CPU/GPU index of the device in the context (e.g. GPU0, GPU1, and so on). With this function it is furthermore possible to select the fastest CPU or GPU device, or fastest overall device in the context by specifying `FASTEST_CPU`, `FASTEST_GPU` or `FASTEST` respectively. Note that the metric to determine the fastest device is by no means accurate (compute-units * clock speed), but usually does the job.

```
shared_ptr<compute_queue>
```

```
create_queue(shared_ptr<compute_device> dev)
```

This creates a `compute_queue` for the specified `compute_device`. Note that the first queue that is returned for each device is the default queue that has already been created automatically during `compute_context` construction and is used for certain internal functionality.

```
shared_ptr<compute_buffer> create_buffer(...)
```

Multiple overloads for this function exist (too many to sensibly list here). This creates a buffer in global memory of the specified device. Memory initialization can optionally happen through host memory specified by either a C-style `void*` pointer and `size_t` size, or a C++ `std::vector` or `std::array` object. Certain `COMPUTE_MEMORY_FLAG` flags can optionally be specified, signaling if the buffer should be readable and/or writable on the device, and if it should be readable and/or writable from the host (defaults to read/write on both sides). Note that this can have performance and memory usage implications and it's usually a good idea to use the minimum set of flags that are necessary. For use with OpenGL (by specifying `COMPUTE_MEMORY_FLAG::OPENGL_SHARING` and an appropriate OpenGL buffer type), the `compute_buffer` will be created through OpenGL and directly be shared with the compute implementation. This way, it's possible to directly create a buffer that can be used with OpenGL and Compute with minimal effort.

```
shared_ptr<compute_buffer>
wrap_buffer(shared_ptr<compute_device> device,
            const uint32_t opengl_buffer,
            const uint32_t opengl_type, ...)
```

Instead of creating a new `compute_buffer` on the device, this will wrap a pre-existing OpenGL buffer object (specified through `opengl_buffer` and its type `opengl_type`) for use with Compute. This optionally also accepts `COMPUTE_MEMORY_FLAG` flags, defaulting again to read/write on both sides and always setting the `COMPUTE_MEMORY_FLAG::OPENGL_SHARING` flag. Note that all other buffer information will be automatically queried through the OpenGL API.

```
shared_ptr<compute_image>
create_image(shared_ptr<compute_device> device,
            const uint4 image_dim,
            const COMPUTE_IMAGE_TYPE image_type, ...)
```

Similar to `create_buffer`, multiple overloads for this function exist. This creates an image in global memory of the specified device. Memory initialization, memory flags and OpenGL functionality are identical to `create_buffer`. Additionally, this function expects the specification of the image dimensions (width, height, depth and layer count, depending on the type) and a very specific `COMPUTE_IMAGE_TYPE` that contains the image base type (1D, 2D, ...), channel count, data type, format, read/write access flags and misc other flags. Various convenience aliases do exist: `RGBA8UI_NORM`, `D32F`, `RG16I`, etc.

```
shared_ptr<compute_image>
wrap_image(shared_ptr<compute_device> device,
           const uint32_t opengl_image,
           const uint32_t opengl_target, ...)
```

Similar to `wrap_buffer`, this wraps a pre-existing OpenGL texture object (specified through `opengl_image` and its texture target `opengl_target`) for use with Compute. Note that all image information (image dimensions and `COMPUTE_IMAGE_TYPE`) will be queried and constructed automatically via the use of the OpenGL API. `COMPUTE_MEMORY_FLAG` flags can again be specified and default to read/write on both sides.

```
shared_ptr<compute_program>
add_program_file(const string& file_name,
                const string additional_options = "")
```

This compiles the compute program source code specified by the file `file_name` for all devices in the context, i.e. returns a single program object that is valid for all devices. The `additional_options` string will be directly added to the compiler call (this can be used to set defines or add include paths, among other things). On success, this returns a `shared_ptr<compute_program>` object, which can then be used to retrieve individual kernel objects in the program. On failure, a `nullptr` is returned.

```
shared_ptr<compute_program>
add_program_source(const string& source_code,
                  const string additional_options = "")
```

Identical to `add_program_file`, except that the source code is specified in memory through a `string` object.

```
shared_ptr<compute_program>
add_precompiled_program_file(
    const string& file_name,
    const vector<llvm_compute::kernel_info>& kernel_infos)
```

This is currently only implemented for Metal on iOS, as no run-time compilation is supported there. This requires the manual specification of all kernels and their parameters in the program, as well as a pre-built program binary. Note that this is only intended as a preliminary solution that will be improved in the future.

Table 1: `compute_context` member functions

3.2.2. Library: device

`compute_device` objects are automatically created for all viable devices (passed requirements and whitelist) in a valid `compute_context`. These mostly contain miscellaneous information about the device, both generic and applying to all backends (e.g. device name, number of compute units, size of global memory, max supported image dimensions, etc.) and backend specific information (version numbers and other identification values). `compute_device` objects are also used to specify where buffer and image objects are to be created, and in combination with a `compute_queue` that has to be created for a device, how work is to be enqueued on the device.

`compute_device` has no functions and only consists of publicly accessible variables. Backend-specific variables can be retrieved by casting the `compute_device` object to the corresponding *any_device* class.

Only listing the most important variables here:

TYPE `type`

The type of the device, either `CPU` or `GPU` and also containing a distinct index inside the context it is part of.

`string` `name`

The name of the device.

`COMPUTE_VENDOR` `vendor`

The vendor of the resp. device: `AMD`, `APPLE`, `HOST`, `INTEL`, `NVIDIA` or `UNKNOWN`. This might be used to implement (or work around) vendor-specific functionality.

`uint32_t` `units`

The number of compute units in the device (Chapter 2.2.1.).

`uint32_t` `simd_width`

The SIMD-width of the device. This is either a fixed value (32 for NVIDIA, 64 for AMD GPUs, 16 for Intel GPUs) or a max value (for CPUs this depends on SSE, AVX and/or NEON support), or 0 if it couldn't be determined.

`size_t` `max_image_1d_dim`

`size2` `max_image_2d_dim`

`size3` `max_image_3d_dim`

The max supported image dimensions for this device.

`uint64_t` `global_mem_size`

`uint64_t` `local_mem_size`

The devices global memory and local memory sizes.

```
uint32_t max_work_group_size
uint3 max_work_group_item_sizes
```

`max_work_group_size` specifies the max total amount of work-items that a work-group can contain, while `max_work_group_item_sizes` specifies the max dimensions of how these work-items can be distributed in the work-group. These two should not be confused: for example, `max_work_group_size` could be 512 and `max_work_group_item_sizes` could be (512, 512, 64). This allows a local work-size of (512, 1, 1) or (32, 16, 1), but not (1, 1, 512) or (128, 128, 1).

```
uint32_t bitness
```

The bitness of the device, either 32 or 64. Note that CUDA and Metal are always 64-bit.

Table 2: `compute_device` variables

3.2.3. Library: queue

`compute_queues`⁴ are associated with the `compute_device` that they have been created for and are used to actually enqueue any kind of work on a device. This first and foremost includes the launching of `compute_kernels`, but is also used to manage operations on `compute_buffers` and `compute_images` (writing, reading, ...). Note also that it is possible to create and use several queues per device simultaneously, but how efficient and concurrent this actually happens to be is a different matter and mostly dependent on the backend and how I can use it (or am using it). The creation of a `compute_queue` should always happen via `compute_context::create_queue`, i.e. direct construction is not advised as it requires backend-specific objects.

```
void finish() const
```

Blocks until all currently scheduled work in this queue has been executed.

```
void flush() const
```

Flushes all scheduled work to the associated device.

```
shared_ptr<compute_device> get_device() const
```

Returns the compute device associated with this queue.

⁴OpenCL: command queues, CUDA: streams, Metal: command queues/buffers

```
void execute(shared_ptr<compute_kernel> kernel,
            vector_type global_work_size,
            vector_type local_work_size,
            kernel-args...)
```

Enqueues (and executes) the specified kernel into this queue. `vector_type` must be one of `uint1`, `uint2` or `uint3`, thus also determining the dimensionality of the kernel. The total amount of work-items per dimension is specified by `global_work_size`, while the amount of work-items per work-group dimension is specified by `local_work_size`. Note that if `global_work_size` isn't evenly divisible by `local_work_size`, more than `global_work_size` work-items will be run so that it is evenly divisible, and the user has to take care of any occurring overlap. Note also that `local_work_size` should not exceed the max possible local work-size of the device, otherwise it will automatically be set to the max possible amount (which may or may not be a viable strategy of handling this). This function is implemented as a variadic template, thus directly accepting any number of kernel arguments. Kernel arguments must consist of `shared_ptr<compute_buffer>`s for buffer objects, `shared_ptr<compute_image>`s for images objects, or any non-pointer and non-`size_t` type for any of the valued kernel parameters (`size_t` is not allowed as its size might differ between the host and the device - it is generally a very good idea to only use types that have the same size on the host and the device).

Table 3: `compute_queue` member functions

3.2.4. Library: program

`compute_programs` are the final form of user-specified compute source code and created through a `compute_context` in cooperation with the `llvm_compute` class and LLVM Compute toolchain. Each `compute_program` will automatically create `compute_kernel` objects for all kernels which they contain and provide an interface to retrieve them. Additionally, `compute_programs` always contain the program objects for all devices in the context, making it quite easy to handle multiple devices with the same code.

```
shared_ptr<compute_kernel> get_kernel(const string& name) const
```

This returns the kernel object with the exact function name of `name`, or `nullptr` if no such kernel exists.

```
const vector<shared_ptr<compute_kernel>>& get_kernels()
```

Returns a container of all the kernel objects in this program. Note that this will immediately contain all kernel objects for CUDA, OpenCL and Metal after program construction, but will only contain kernel objects that have been retrieved through `get_kernel` for Host-Compute, as it is impossible to retrieve Host-Compute kernel functions any other way.

```
const vector<string>& get_kernel_names()
```

Returns a container of all kernel function names in this program. Same restrictions as `get_kernels` apply.

Table 4: `compute_program` member functions

3.2.5. Library: kernel

`compute_kernels` contain the actual backend-specific kernel objects and implementation of launching kernels, i.e. the setting of kernel arguments, possibly the locking and unlocking of memory objects, the launching or enqueueing of the kernel itself, and for Host-Compute, the actual execution of kernels as well. Note that `compute_kernels` are never called directly, but are always enqueued through a `compute_queue`. Note also that each kernel contains the kernel objects for all devices in the context, selecting the appropriate one at run-time depending on the `compute_queue`. Furthermore, it is very well possible and intended to have different kernel arguments (number, sizes, types) for different devices, as many things are device-dependent and device specialization might be desired.

3.2.6. Library: memory

`compute_memory` is the base class of `compute_buffer` and `compute_image`, providing functionality that is common between the two.

```
const void* get_device() const
```

Returns a pointer to the associated device.

```
const COMPUTE_MEMORY_FLAG& get_flags() const
```

Returns the flags that the memory object has been created with.

```
void* get_host_ptr() const
```

Returns the associated host memory pointer that was specified during creation.

```
const uint32_t& get_opengl_object() const
```

This function can be used to retrieve the OpenGL object (buffer object or texture object, depending on the inheriting class) that has been created for this memory object or that is being wrapped.

```
bool acquire_opengl_object(shared_ptr<compute_queue> cqueue)
```

All OpenGL-compatible compute implementations require that OpenGL objects can only be used with either Compute or with OpenGL at a time, necessitating the execution of specific migration functions when switching from one to the other. Therefore, a release/acquire mechanism becomes necessary, also attaching to a `compute_queue` to safely signal when a memory object should be migrated (e.g. acquire, run kernel, release again). Note that after creating or wrapping an OpenGL object (when constructing a compute memory object), it will immediately be acquired for use with Compute.

```
bool release_opengl_object(shared_ptr<compute_queue> cqueue)
```

The corresponding release function to the above acquire function, releasing the memory object from use with Compute, allowing it to be used with OpenGL.

```
void zero(shared_ptr<compute_queue> cqueue)
```

Sets the memory of the complete memory object to zero. (Potentially faster than `compute_buffers` fill function.)

```
void clear(shared_ptr<compute_queue> cqueue)
```

Alias for `zero`.

Table 5: `compute_memory` member functions

3.2.7. Library: `buffer`

`compute_buffer`s are used to allocate memory and store data in global memory of a device, allowing both the host and the device to read and write to these buffers, thus permitting data sharing between the host and the device, and actually being able to specify the data a kernel has to work on and retrieving the processed data again afterwards on the host. It is also possible to directly couple a host memory pointer with a `compute_buffer` allowing initialization, read and write operations on a host pointer that is only specified once during construction. Furthermore, some compute backends (OpenCL, Metal) actually allow host memory to be directly used as a device's global memory buffer, although it is up to each vendor's implementation how this is ultimately achieved (i.e. the driver might still allocate memory on the device and copy data automatically). Note however that on some compute hardware (integrated GPUs) host memory and global memory might actually be the same physical memory, making host/device data sharing rather efficient ("zero-copy" if used correctly).

```
void read(shared_ptr<compute_queue> cqueue,
          void* dst,
          const size_t size = 0,
          const size_t offset = 0)
```

Copies (reads from) the device memory at `offset` of size `size` (or the whole buffer if 0) to the specified host memory pointer `dst`. An additional overload exists, without the `dst` parameter, which copies the memory to the host memory pointer specified at `compute_buffer` creation instead. The `compute_queue` is necessary for correct ordering. If it is `nullptr`, the default device queue will be used.

```
void write(shared_ptr<compute_queue> cqueue,
            const void* src,
            const size_t size = 0,
            const size_t offset = 0)
```

Copies (writes) the host memory pointed to by `src` to the device memory at `offset` of size `size` (or using the whole buffer size if 0). An additional overload exists, without the `src` parameter, which copies the memory from the host memory pointer specified at `compute_buffer` creation instead.

```
void copy(shared_ptr<compute_queue> cqueue,  
          shared_ptr<compute_buffer> src,  
          const size_t size = 0,  
          const size_t src_offset = 0,  
          const size_t dst_offset = 0)
```

This function copies device memory of size `size` (or the whole buffer if 0) from one compute buffer to another (the calling object acts as the target, the `src` buffer as the source). Data is read from `src_offset` onwards and written to `dst_offset` onwards.

```
void fill(shared_ptr<compute_queue> cqueue,  
          const void* pattern,  
          const size_t& pattern_size,  
          const size_t size = 0,  
          const size_t offset = 0)
```

Fills the complete buffer (or from `offset` onwards, and up to size `size`, if non-zero) with the specified `pattern` of the specified `pattern_size`. This function is similar to a `memset` or `memset_pattern*`, but allows the use of arbitrarily sized patterns (note that there are usually fast-paths for patterns of size 1, 2, 4 or 8).

```
bool resize(shared_ptr<compute_queue> cqueue,  
            const size_t& size,  
            const bool copy_old_data = false,  
            const bool copy_host_data = false,  
            void* new_host_ptr = nullptr)
```

This is a rather complex function which will essentially allocate new device memory of the specified `size`, copies the old device memory to the new device memory if `copy_old_data` is true, or initializes the new buffer from the previously set or new host memory pointer if `copy_host_data` is true, and also allows the host pointer to be replaced by a new host pointer if `new_host_ptr` is non-`nullptr`.

```
void* map(shared_ptr<compute_queue> cqueue,
          const COMPUTE_MEMORY_MAP_FLAG flags,
          const size_t size = 0,
          const size_t offset = 0)
```

Maps the buffer to host memory, returning a pointer to the host memory. This either maps the complete buffer if `size` and `offset` are 0, or from `offset` onwards up to `size` bytes. `COMPUTE_MEMORY_MAP_FLAG` flags are used to signal if the mapping operation should be read/write, read-only or write-only (aka write discard). This has direct performance implications as memory might not need to be read from the device or not need to be written back. `COMPUTE_MEMORY_MAP_FLAG` can also be used to signal if the mapping operation is blocking or asynchronous - note however that not all backends or usage scenarios allow asynchronous mapping, in which case it will fall back to blocking behavior (this is also the default if nothing is specified). Note that the returned host memory is at least 128-bit aligned.

```
void unmap(shared_ptr<compute_queue> cqueue, void* mapped_ptr)
```

The corresponding unmapping call to the above `map`, with `mapped_ptr` of course corresponding to a previous `map`. If this is a write mapping, this will write the host memory back to device memory. Note that after calling this function, the memory pointed to by `mapped_ptr` is no longer usable.

```
const size_t& get_size() const
```

Returns the size of the buffer object.

Table 6: `compute_buffer` member functions

3.2.8. Library: image

`compute_images` are similar to `compute_buffers` in that they represent global memory on a device, but do so in a very specialized form that usually provides additional performance benefits and/or easier access of data. To do so, they are stored in a vendor-proprietary format that is not known to the user and consequently require different specialized functions to read and write data to them. Note that all compute backends handle these quite differently and that no guarantees can be made about which image formats are supported everywhere. It is however always a good idea to stick with 1, 2 or 4 channels (R, RG, RGBA) and well-known data formats (8-bit unsigned normalized, 16-bit and 32-bit floating point, 16-bit and 32-bit signed/unsigned non-normalized).

```
const uint4& get_image_dim() const
```

Returns the image dimensions the image has been created with (width, height, depth, layer count, depending on the image type).

```
const COMPUTE_IMAGE_TYPE& get_image_type() const
```

Returns the `COMPUTE_IMAGE_TYPE` that the image has been created with.

```
const size_t& get_image_data_size() const
```

This function returns the computed (expected) image memory data size. This might be of use when mapping the image to host memory and is also used for several internal purposes.

```
void* map(shared_ptr<compute_queue> cqueue,
          const COMPUTE_MEMORY_MAP_FLAG flags
```

Maps the whole device image memory to host memory, returning a pointer to it. As for `compute_buffer`, `COMPUTE_MEMORY_MAP_FLAG` signals whether this is a read/write, read-only or write-only operation and if it's blocking or asynchronous. Note that the returned host memory is at least 128-bit aligned.

```
void unmap(shared_ptr<compute_queue> cqueue, void* mapped_ptr)
```

The corresponding unmapping call to the above `map`, with `mapped_ptr` of course corresponding to a previous `map`. If this is a write mapping, this will write the host memory back to device memory. Note that after calling this function, the memory pointed to by `mapped_ptr` is no longer usable.

Table 7: `compute_image` member functions

Putting it all together:

```
// get the default context
auto ctx = floor::get_compute_context();
// get the "fastest device"
auto dev = ctx->get_device(compute_device::TYPE::FASTEST);
// create a queue for this device
auto dev_queue = ctx->create_queue(dev);
// compile program (all kernel functions in the program),
// pass additional options directly to the compiler
auto prog = ctx->add_program_file("program.cpp", "-DMAGIC_NUMBER=42 -Weverything"
                                     "-Isome/folder/");

// retrieve a specific kernel from the program
auto kernel = prog->get_kernel("some_kernel");
// wrap a pre-existing OpenGL 2D image
auto img = ctx->wrap_image(dev, opengl_tex, GL_TEXTURE_2D);
const auto dim_xy = img->get_image_dim().xy;
// create a buffer on the specified device
auto buffer = ctx->create_buffer(dev, sizeof(float4) * dim_xy.x * dim_xy.y);
// schedule kernel for execution (here: 2D),
// kernel arguments are passed via variadic template
dev_queue->execute(kernel,
                  // global size, #work-items per dimension
                  dim_xy,
                  // local size, #work-items per work-group dimension
                  uint2 { 16, 16 },
                  // args...
                  img, buffer
);
```

Listing 6: exemplary use of the Host API

3.2.9. Online/Offline Compilation

All of the functionality mentioned in Chapter 2.1.4. is available and used at runtime on all desktop platforms (online compilation). For testing, development and general "nice-to-have" purposes (and to show how it might be more deeply integrated in external projects) I have also created a standalone project that wraps all of this library functionality in a single binary, and can thus be used to compile compute source code without needing to call any libfloor library functions or having to do this by oneself (offline compilation). Note however that none of the produced binaries (with one exception) is intended to actually be used. All of this is just to easily and quickly check if compute source code actually compiles for a specific backend (including ones that are otherwise unavailable, e.g. compiling to CUDA/PTX when no NVIDIA card is present, or compiling to Metal/AIR on non-Apple platforms) and possibly to have a look at the produced output for whatever reasons there might be. The exception here is Metal for iOS, where runtime compilation is not an option and a full device binary has to be compiled on the

host as a result. The offline compiler (`occ`) has furthermore the ability to test if compiled (intermediate) binaries compile with each individual compute backend, just as the library version does in its final compilation step. This should of course usually be the case, but it's an invaluable tool when experimenting with or developing new compiler functionality, or when checking for bugs in either my part of the toolchain or in a vendor's implementation (note that all intermediate products can be transformed to a textual representation that is easily modifiable, and then transformed back into binary format).

3.2.10. Debugging & Development

Since it is quite important that debugging, development and profiling of Compute code is possible, the following tools targeting the CUDA, OpenCL or Metal platform were tested and are supported:

- CUDA: NVIDIA Nsight [NVI15d] generally works when used with binary profiling, i.e. detailed profiling is possible just like with any other CUDA application. Code line info is also available when properly passed through and compilation options are set. Note that it is also possible to enable profiling or debug specific code generation when these options are set in the LLVM and CUDA part of the libfloor config file.
- Metal: the Xcode Metal debugger works just as with normal Metal for the most part. Buffer memory, image data and source code inspection work, encoder performance results are available, and line profiling information works to some degree (Xcode/Apple are rather fickle, sometimes working, sometimes not, sometimes crashing - it's not clear why, but seems to be a Xcode problem).
- OpenCL: AMD CodeXL [AMD15a] is working, but functionality is limited as it is missing proper SPIR support.
- OpenCL: Intel VTune [Int15b] works as well, especially interesting for CPU kernel code.

As debugging on GPU hardware is still an issue nowadays, Host-Compute was specifically created to fill this gap. The LLDB [LLV15n] and GDB [GDB15] debuggers should fully work (including IDEs that integrate them like Xcode), and act just like any other user application as everything is normal C++ code (as intended). It is possible to set breakpoints inside kernel code, and conditional breakpoints work as well (e.g. `global_id.x == 0 && global_id.y == 42`). Debugging in Visual Studio is subject to LLVM/LLDB progress and not possible right now (other than line information when crashing), but progress is being made,

especially since Microsoft open-sourced their PDB format and code recently for the intent purpose of being integrated into LLVM [Mic15b] (expecting this to work sometime next year). Debugging with Clang/C2 [Mic15c] should be possible as well, but the release of this was too close to the end of this thesis that I couldn't make it compatible yet.

The use of Clang/LLVM sanitizers [LLV15a] is possible and encouraged. The AddressSanitizer can be used to find memory issues like out-of-bounds accesses. The ThreadSanitizer can find race conditions, among other things. The MemorySanitizer is used to detect uninitialized reads. The Undefined-behavior sanitizer can be used as well, but is generally problematic when used full-on, since UB is used and relied upon quite often, but it has a lot of sub-options that can be enabled specifically. Note that libfloor and example programs have build options for all of these in the UNIX build script: just build with the "asan", "msan", "tsan" or "ubsan" option. Note that this of course requires run-time library support from the OS it is run on.

Trivial AddressSanitizer example (kernel code first, then the error detected by AddressSanitizer, listing the type and line of the error, then the size and location of the closest memory allocation and where it originated, `main.cpp:600`):

```
kernel void off_by_one(buffer<const float> src, buffer<float> dst) {
    dst[global_id.x] = src[global_id.x + 1]; // NOTE: this is line 61
}
```

Listing 7: intentional off-by-one access kernel

```
==80809==ERROR: AddressSanitizer: heap-buffer-overflow on address
    0x629000013200 at pc 0x00010002a6a2 bp 0x0001491a46a0 sp 0x0001491a4698
READ of size 4 at 0x629000013200 thread T10
    #0 0x10002a6a1 in off_by_one off_by_one.cpp:61

0x629000013200 is located 0 bytes to the right of 16384-byte region
    [0x62900000f200,0x629000013200)
allocated by thread T0 here:
[...]
#4 0x10176d9e3 in host_compute::create_buffer(
    std::__1::shared_ptr<compute_device>, unsigned long const&,
    COMPUTE_MEMORY_FLAG, unsigned int) memory:3732
#5 0x100003673 in main main.cpp:600
```

Listing 8: off-by-one access detected by AddressSanitizer

3.2.11. Graphics Interoperability

While Compute on its own is very useful, it is often necessary to interoperate with Graphics APIs such as OpenGL or Metal, either when utilizing Compute and Graphics in a joint graphics pipeline (e.g. where Compute can take over the job of vertex data processing or post processing of image data or some other task), or simply for visualization purposes of data that has been generated through Compute. Both of these are heavily used in projects of Chapter 4. Application.

To gain this interoperability, CUDA, OpenCL and Host-Compute provide mechanisms to share buffers and images with OpenGL. This is exposed through two ways in the library: one can either create a new OpenGL object (which is created according to the specified libfloor enums and flags) or wrap a pre-existing OpenGL object (in which case all necessary information will be queried through OpenGL API functions⁵). To perform either of these, mapping OpenGL types to, from and between internal libfloor types and other compute backends becomes necessary. While this is mostly trivial for buffer objects (it's all just simple data after all), it is far from trivial for image objects. Have a look at Table 15 to see which image types are generally supported. Note that some types or formats may not be supported in interoperability mode even though they are supported when only using Compute (e.g. CUDA depth image), and others may only be supported in interoperability mode (e.g. OpenCL MSAA images).

Access of shared objects can only happen on either side one at a time (Compute or OpenGL). To switch between Compute and OpenGL, buffer and image objects provide two functions named `acquire_opengl_object` (acquire for use with Compute) and `release_opengl_object` (release from use with Compute), as already detailed in Table 5.

A completely different matter is Metal. Since Metal is both a Graphics and a Compute API, there seems to have been no need for providing interoperability with OpenGL. On the upside, no distinction has to be made between Graphics and Compute, i.e. internal handling is leaner and there is no need to acquire and release buffer and image objects for and from use with Compute.

⁵except for the buffer type or texture target, as these can't be determined through OpenGL, short of trial & error binding, which is a bad idea as this will produce errors and possibly break things and is not even guaranteed to work

3.3. Device

3.3.1. Basic Math Library

This part of the library provides basic math functionality on both the host and the device. Notably, all functions supported on compute devices are also supported at compile-time with `constexpr`. As the amount of functions is rather large, I will only list what is generally supported, instead of describing everything in detail⁶. Generally, functions should work as one would expect from their name and quite a few should be familiar to anyone knowing GLSL or OpenCL. The full source code to this can be found in the `math` and `constexpr` folders of libfloor.

Starting off with the scalar math functions, and building block of the vector library described further down below. `T` in this table stands in for `float` and `double` if the device has `double` support (CUDA, OpenCL with extension, Host-Compute). In OpenCL and Metal these functions are mostly implemented by the platform vendors, while for CUDA some of these needed to be either partially or completely implemented in software, since the hardware doesn't actually have support for these. I presume this is also the case with other GPU hardware, so always consider these functions to be costly. All `const_math` functions are fully implemented in software so that they work with `constexpr` at compile-time. There is also a `const_select` namespace with functions of the same names, which can be used to automatically select between run-time and compile-time functions depending on if the function arguments are `constexpr` (or otherwise compile-time constants) or run-time values.

```
T abs(T val)
T fabs(T val)
T const_math::abs(T val)
```

Computes the absolute value of `val`. For `abs`, `T` can also be `int16_t`, `int32_t` or `int64_t` here. `fabs` simply aliases to `abs` and only exists for compatibility reasons.

```
T sqrt(T val)
T const_math::sqrt(T val)
```

Computes the square root of `val`.

```
T rsqrt(T val)
T const_math::rsqrt(T val)
```

Computes the reciprocal square root of `val`.

⁶also: all functions are documented in the code

```
T fmod(T x, T y)
```

```
T const_math::fmod(T x, T y)
```

Computes $x \% y$, the fp remainder of the division $\frac{x}{y}$ (modulo).

```
T floor(T val)
```

```
T const_math::floor(T val)
```

Computes $\lfloor val \rfloor$.

```
T ceil(T val)
```

```
T const_math::ceil(T val)
```

Computes $\lceil val \rceil$.

```
T round(T val)
```

```
T const_math::round(T val)
```

Rounds val towards the nearest integer value, with values halfway between two integer values rounded away from 0.

```
T trunc(T val)
```

```
T const_math::trunc(T val)
```

Truncates the fractional part of val , or $\text{floor}(val)$ if val is positive and $\text{ceil}(val)$ if it is negative.

```
T rint(T val)
```

```
T const_math::rint(T val)
```

Same as `round`, only exists for compatibility reasons as there is no possibility to select rounding modes.

```
T pow(T a, T b)
```

```
T const_math::pow(T a, T b)
```

Computes a^b , a to the power of b .

```
T exp(T val)
```

```
T const_math::exp(T val)
```

Computes e^{val} , the exponential function value of val . Note: this is a partial software computation in CUDA, computed as $\text{exp2}(val / \log(2))$ or $2^{\frac{val}{\ln(2)}}$.

```
T exp2(T val)
```

```
T const_math::exp2(T val)
```

Computes 2^{val} .

```
T log(T val)
```

```
T const_math::log(T val)
```

Computes the natural logarithm of val . Note: this is a partial software computation in CUDA, computed as $\log_2(val) / \log_2(e)$ or $\frac{\text{ld}(val)}{\text{ld}(e)}$.

```
T log2(T val)
T const_math::log2(T val)
```

Computes the base-2 logarithm of val.

```
T sin(T val)
T const_math::sin(T val)
```

Computes the sine of val, with val given in radians.

```
T cos(T val)
T const_math::cos(T val)
```

Computes the cosine of val, with val given in radians.

```
T tan(T val)
T const_math::tan(T val)
```

Computes the tangent of val, with val given in radians.

```
T asin(T val)
T const_math::asin(T val)
```

Computes the arc sine of val. Note: completely computed in software in CUDA. For the interval $[-0.5, 0.5]$ this is computed as $\text{copysign}(x + 0.1666700692808536 \cdot x^3 + 0.07487039270444955 \cdot x^5 + 0.04641537654451593 \cdot x^7 + 0.01979579886701673 \cdot x^9 + 0.04922871471335342 \cdot x^{11}, val)$ with $x = |val|$, the result of EconomizedRationalApproximation[ArcSin[x], {x, {-0.55, 0.55}, 12, 0}] obtained via Mathematica (and conveniently chosen for it's precision, not requiring divisions and terms of even-numbered exponents being eliminated).

For values outside the interval $[-0.5, 0.5]$, this is computed as $\frac{\pi}{2} - 2 \cdot \text{asin}(\sqrt{\frac{1-x}{2}})$ [Wik15g], thus moving the asin argument to the interval $[-0.5, 0.5]$ and being able to use the other computation.

Overall this boils down to 7 FMAs, 2 MULs, 2 selects, 1 abs, 1 sqrt, 1 copysign.

```
T acos(T val)
T const_math::acos(T val)
```

Computes the arc cosine of val. Note: completely computed in software in CUDA via $\pi/2 - \text{asin}(val)$.

```
T atan(T val)
T const_math::atan(T val)
```

Computes the arc tangent of val. Note: completely computed in software in CUDA via $\text{asin}(a * \text{rsqrt}(a * a + 1))$.

```
T atan2(T y, T x)
```

```
T const_math::atan2(T y, T x)
```

Computes the arc tangent of $\frac{y}{x}$ as per [Wik15b]. Note: completely computed in software in CUDA.

```
T fma(T a, T b, T c)
```

```
T const_math::fma(T a, T b, T c)
```

Computes $a \cdot b + c$, the infinitely precise product of a and b and addition with c, with only the final value being rounded to the floating point representation.

```
T copysign(T a, T b)
```

```
T const_math::copysign(T a, T b)
```

Returns a with the sign of b, essentially $(b < 0.0 ? -1.0 : 1.0) * \text{abs}(a)$.

```
T const_math::clamp(T val, T min, T max)
```

Returns val clamped to the interval $[min, max]$. Works with any arithmetic type (integer, floating point). Note: also intended for run-time use.

```
T const_math::wrap(T val, T max)
```

Returns val "wrapped" around max (essentially modulo, but the mathematically correct one, not the C one, e.g. `wrap(-1, 7)` returns 6, not -1). Works with any arithmetic type (integer, floating point). Note: also intended for run-time use.

```
T const_math::interpolate(T a, T b, T t)
```

```
T const_math::cubic_interpolate(T a_prev, T a, T b, T b_next, T t)
```

```
T const_math::catmull_rom_interpolate(T a_prv, T a, T b, T b_nxt, T t)
```

Returns the linear, cubic or centripetal Catmull-Rom interpolation of the given values and interpolator t. Note: also intended for run-time use.

```
T const_math::rad_to_deg(T val)
```

```
T const_math::deg_to_rad(T val)
```

Conversion functions between radians and degrees. Note: also intended for run-time use.

Table 8: basic math functions

Building on this base functionality, and just as ubiquitous, are the `vectorN` classes, providing 1D, 2D, 3D and 4D vector functionality for all native scalar types: `vector1<T>`, `vector2<T>`, `vector3<T>` and `vector4<T>`. Instantiation is alternatively possible via `vector_n<T, size_t dim>`. Typedefs and partial specializations for the various scalar types do exist: `uint8_t` as `ucharN`, `int8_t` as `charN`, `uint16_t` as `ushortN`, `int16_t` as `shortN`, `uint32_t` as `uintN`, `int32_t` as `intN`, `uint64_t` as `ulongN`, `int64_t` as `longN`, `float` as `floatN`, `double` as `doubleN`, `long double` as `ldoubleN`, `bool` as `boolN`, and `size_t` as `sizeN`. Note that I expressly wrote this vector library as I was rather disappointed by the lack of functionality provided by vector types in certain other Compute libraries. Note also that this doesn't make use of Clang built-in vector types as these are again very limited and are not `constexpr`, but does provide conversion to and from them if necessary. Named member functions usually do exist both in a passive form (e.g. `normalized`) returning a copy of the vector object with the specific operation performed on it, and an active form (e.g. `normalize`) that performs the operation directly on the vector object, returning itself.

constructors and assignment functions

Same as GLSL/OpenCL, allowing construction from scalar types as well as construction from and combination of smaller vector types. Assigning scalar values will set all vector components to the same scalar, assigning lower vector types will only assign the corresponding vector components. Does of course support C++ uniform initialization.

binary operators: `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `|`, `&`, `^`

relational operators: `||`, `&&`, `==`, `!=`, `<`, `>`, `<=`, `>=`

unary operators: `++`, `--`, `+`, `-`, `!`, `~`

assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `|=`, `&=`, `^=`, `[]`

These are supported for all vector types and work with either vector or scalar types on either side (if applicable). Bitwise integer operations on floating point types are explicitly supported, but only work at run-time due to `constexpr` limitations. Relational operators always return `boolN` vectors, which can then be tested with the `any`, `all` or `none` member functions. `%` does work on floating point types. All of the other operators should work as one would expect from their corresponding scalar types.

geometric functions: `dot`, `cross`, `length`, `distance`, `distance_squared`, `normalize`, `angle`, `faceforward`, `reflect`, `refract`, `step`, `smoothstep`, `rotate`, `perpendicular`

Same as GLSL/OpenCL functions, with some additions. Some functions only apply to vectors of a certain dimensionality.

rounding and related: `round`, `floor`, `ceil`, `trunc`, `rint`, `clamp`, `wrap`, `round_next_multiple`

As usual: both passive and active forms, support vector and scalar parameters if applicable.

relational functions: `any`, `all`, `none`, `is_equal`, `is_unequal`, `is_less`, `is_less_or_equal`, `is_greater`, `is_greater_or_equal`, `is_equal_vec`, `is_unequal_vec`, `is_less_vec`, `is_less_or_equal_vec`, `is_greater_vec`, `is_greater_or_equal_vec`

Similar to the relational operators, but optionally accepting an epsilon parameter. Non-`is*_vec` will always return a `bool` (as if called with `.all()`), `is*_vec` functions return a `boolN` vector. `any`, `all` and `none` respectively return true if any, all or none of the components of the specified `boolN` vector is/are true.

testing functions: `is_null`, `is_finite`, `is_nan`, `is_inf`, `is_normal`

As their names suggest, return true if: all components are 0, all components are finite values, any component is NaN, any component is infinite, all components are not denormals.

misc accessors: `x`, `y`, `z`, `w`, `xy`, `zw`, `xyz`, `lo`, `hi`, `data`, `[]`, `ref<'x', ...>`, `ref_idx<0, ...>`, `swizzle<0, ...>`, `as_tuple`, `as_tuple_ref`

Component accessors and lower vector accessors as one would expect. As this is C++, sadly none of the fancy GLSL/OpenCL style accessors are directly possible (e.g. `.xxzz`), but they can be emulated through either `ref` taking the component names as `chars`, or `ref_idx` taking them as indices, or alternatively constructing a new vector manually which is just as efficient. `data` returns a `T*` to the vector object, `[]` accesses the component at the specified index (with compile-time range checking if possible), `as_tuple` returns the vector components as a C++ `tuple<T, ...>`, `as_tuple_ref` as `tuple<T&, ...>`.

functional: `set`, `set_if`, `apply`, `apply_if`, `count`, `average`, `accumulate`

`set` components to the components of another vector (dependent on a `boolN` vector). `apply` an unary function on each vector component (dependent on a `boolN` vector). `count` components that are equal to a specified scalar value, vector or return true on a specified unary function. `average` and `accumulate` compute the average or sum of all components.

misc functions: `fma`, `min`, `max`, `minmax`, `min_element`, `max_element`, `min_element_index`, `max_element_index`, `interpolate`, `cubic_interpolate`, `catmull_rom_interpolate`, `sign`, `signbit`, `sqrt`, `rsqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `exp`, `exp2`, `log`, `log2`, `pow`, `to_rad`, `to_deg`

Should all be fairly obvious. Math functions are applied to each vector component. Cubic interpolation functions do require three vector parameters (in addition to the calling vector object).

misc integer functions: `popcount`, `clz`, `ctz`, `ffs`, `parity`

Population count (number of 1s), count leading zeros (number of 0s until first 1 bit, starting at MSB), count trailing zeros (number of 0s until first 1 bit, starting at LSB), find first set (1 + the index of the least significant 1 bit, or 0 if component is 0), and parity value (number of 1 bits modulo 2).

host-only and run-time-only: `<< on ostream`, `to_string`, `random`

Only usable at run-time, because they obviously require run-time functionality (streams, strings, PRNG). Host-only, because none of these can be properly implemented on Compute devices yet (or: not without unreasonable costs).

Table 9: vector functions

```
// select sliced volume stack from the camera forward vector
const size_t axis = forward_vec.normalized().abs().max_element_index();
// making sure a vector is correctly normalized at compile-time
constexpr float3 norm_vec { float3(4.2f, 2.3f, -8.0).normalize() };
static_assert(norm_vec.dot() == 1.0f, "norm_vec should be normalized");
// host-only: random functions (STL mersenne twister with uniform distribution)
uchar4 random_color = { uchar3::random(), 255u };
// misc compile-time and general functionality
constexpr float2 v1 { 1.0f, 0.0f }, v2 { 0.0f, 1.0f };
static_assert((v1.interpolated(v2, 0.5f).normalize() ==
    float2(const_math::cos(const_math::PI_DIV_4<float>),
        const_math::sin(const_math::PI_DIV_4<float>))).all(),
    "all is well");
```

Listing 9: exemplary uses of the vector library

Additionally, the math primitives `matrix4<T>` (a 4x4 matrix supporting floating point and integer types) and `quaternion<T>` (supporting floating point types), also usable at compile-time, exist as well, but detailed descriptions are omitted here for brevity. These do function as one would expect from them and they can interact with vector library types (e.g. vector/matrix multiplication, rotation of a vector according to a quaternion). The usual host/run-time only restrictions apply.

3.3.2. Compute-specific Functionality

The source code for these can be found in the `compute/device` folder of libfloor.

`kernel`

To signal that a function is to be used as a kernel function, this `kernel` attribute must be attached to them.

`global`

Signals that memory is located in the global address space. Valid on pointers of kernel and function parameters. Valid on pointers inside `structs` and `classes` only if set during the kernel lifetime. Use of this keyword is generally not advised and use of the `buffer` container or `template` functions taking any address space is to be preferred.

`local`

Signals that memory is located in the local address space. Valid on pointers of function parameters. Valid on pointers inside `structs` and `classes` only if set during the kernel lifetime. As with `global`, use of this keyword is not advised.

`constant`

Signals that memory is located in the constant address space. Valid on pointers of kernel and function parameters. Note that use of this keyword is generally not necessary, as `static const` variables defined globally, inside namespaces or as `class` members are considered `constant` by default, since no other address space is allowed for these.

`buffer<typename T>`

This is the preferred way of declaring a global memory pointer, primarily intended for kernel and normal function parameters. This essentially allows `buffer<T>` to be transformed to `global T*` or any of the backend-specific equivalents. Note that qualifiers on `T` are passed through, i.e. `buffer<const float>` results in a `const float*` pointer.

`local_buffer<typename T, size_t X, size_t Y = 0, size_t Z = 0>`

This is the only way to declare and allocate local memory, allowing the creation of 1D, 2D or 3D arrays in local memory. This is essentially equivalent to a `T var[X][Y][Z];` allocation. Note that due to irreconcilable differences between backends only this type of static allocation is currently supported. Note also that even if dynamic allocation was possible, static allocation should still be preferred as it allows the compiler to make safe assumptions about the size of the array, e.g. making it possible to compute static offsets into the array instead of computing indices at run-time which might be costly due to the involved integer math.


```
constant_buffer<typename T>
```

Similar to `buffer`, but for data in constant memory. Essentially results in a `constant const T* const` pointer. Qualifiers are possible, but obviously superfluous as this is read-only memory.

```
constant_array<typename T, size_t N>
```

This is for use on constant variables and essentially equivalent to a `const T var[N]`. As mentioned in `constant`, it is however rather unnecessary as `constant` will be automatically attached to appropriate variables.

```
param<typename T>
```

This is solely for use on kernel parameters and the only way to specify parameters that are not buffers or images. Essentially equivalent to a `const T` parameter (e.g. `param<float> var` becomes `const float var`). Note that this is absolutely necessary due to backends handling this very differently.

Table 10: kernel and address space keywords, address space specific containers

```
uint3 global_id
uint32_t get_global_id(uint32_t dim)
```

Returns the global work-item index inside the global work-size that was specified when launching the kernel. This exists both as a function call and a global variable (which might be implemented via a `get_global_id` call depending on the backend). Note that `dim` should be a value between 0 and 2, returning 1 for unused dimensions and 0 for invalid `dim` values.

```
uint3 local_id
uint32_t get_local_id(uint32_t dim)
```

Returns the local work-item index inside the local work-group size that was specified when launching the kernel.

```
uint3 group_id
uint32_t get_group_id(uint32_t dim)
```

Returns the group index of the work-group the work-item is located in. This returns a value between 0 and $(\text{get_global_size}(\text{dim}) / \text{get_local_size}(\text{dim})) - 1$ for the specified dimension.

```
uint3 global_size
uint32_t get_global_size(uint32_t dim)
```

Returns the global work-size that was specified when launching the kernel.

```
uint3 local_size
uint32_t get_local_size(uint32_t dim)
```

Returns the local work-size that was specified when launching the kernel.

```
uint3 group_size
uint32_t get_num_groups(uint32_t dim)
```

Returns $(\text{get_global_size}(\text{dim}) / \text{get_local_size}(\text{dim})) - 1$ for the specified dimension, the total amount of work-groups that are launched for the specified global and local work-size.

Table 11: ID-handling functions and variables

```
// computes partial sums of  $c = \sum_{i=1}^n a_n \cdot b_n$ ,
// partial sum chunks are of size 512 (== work-group size)
#define REQ_TILE_SIZE 512
kernel void partial_dotn(buffer<const float> a,
                        buffer<const float> b,
                        buffer<float> c) {
    // multiply resp. a and b for this work-item (or init with 0 if >= size)
    // and copy the result to the local memory buffer
    local_buffer<float, REQ_TILE_SIZE> group_c;
    group_c[local_id.x] = (global_id.x >= global_size.x ? 0.0f :
                          a[global_id.x] * b[global_id.x]);
    // sum up (reduction to [0])
    for(uint32_t i = REQ_TILE_SIZE / 2; i > 0; i >= 1) {
        local_barrier(); // sync local mem + work-item barrier
        if(local_id.x < i) {
            group_c[local_id.x] += group_c[local_id.x + i];
        }
    }
    // write partial c for this group
    if(local_id.x == 0) c[group_id.x] = group_c[0];
}
```

Listing 10: exemplary use of address space containers and ID-handling variables

Atomic functions exist both as OpenCL 1.2 style freestanding functions that act on pointers in global or local address space, and as C++’s STL `std::atomic<T>` which has been modified to make use of these freestanding functions. Please note that 32-bit atomics are supported everywhere, while support for 64-bit is limited to CUDA, OpenCL with necessary extensions and Host-Compute. Furthermore, what kinds of atomic operations and on which types these can operate varies between backends, making it necessary to implement these atomics in a somewhat less efficient (but still atomic) compare-and-exchange loop. This is mostly necessary for operations on floating point types, but also for some of the advanced 64-bit atomics for hardware

that only supports basic 64-bit atomic operations. In the table below, `T` stands in for `uint32_t`, `int32_t` and `float` for all backends, and `uint64_t`, `int64_t` and `double` for those supporting 64-bit atomics. Note also that if hardware has basic support for 64-bit atomics, all 64-bit functions are supported.

`T atomic_add(T* ptr, T val)`

Atomically reads the `T` value pointed to by `ptr`, adds `val` to it and writes the result back to `ptr`. Returns the value at `ptr` prior to addition.

`T atomic_sub(T* ptr, T val)`

Atomically reads the `T` value pointed to by `ptr`, subtracts `val` from it and writes the result back to `ptr`. Returns the value at `ptr` prior to subtraction.

`T atomic_inc(T* ptr)`

Atomically reads the `T` value pointed to by `ptr`, adds 1 to it and writes the result back to `ptr`. Returns the value at `ptr` prior to incrementation.

`T atomic_dec(T* ptr)`

Atomically reads the `T` value pointed to by `ptr`, subtracts 1 from it and writes the result back to `ptr`. Returns the value at `ptr` prior to decrementation.

`T atomic_xchg(T* ptr, T val)`

Atomically stores `val` at `ptr` ("exchanges the value through `val`"). Returns the value at `ptr` prior to the exchange.

`T atomic_cmpxchg(T* ptr, T expected, T desired)`

Atomically reads the `T` value pointed to by `ptr`, compares it against `expected`. If identical, atomically stores `desired` at `ptr` and returns `expected`. If different, returns the current value at `ptr`.

`T atomic_min(T* ptr, T val)`

Atomically reads the `T` value pointed to by `ptr`, computes the minimum of this value and `val` and writes the result back to `ptr`. Returns the value at `ptr` prior to the min operation.

`T atomic_max(T* ptr, T val)`

Atomically reads the `T` value pointed to by `ptr`, computes the maximum of this value and `val` and writes the result back to `ptr`. Returns the value at `ptr` prior to the max operation.

`T atomic_and(T* ptr, T val)`

Atomically reads the `T` value pointed to by `ptr`, bitwise-ANDs `val` with it and writes the result back to `ptr`. Returns the value at `ptr` prior to the operation.

```
T atomic_or(T* ptr, T val)
```

Atomically reads the T value pointed to by `ptr`, bitwise-ORs `val` with it and writes the result back to `ptr`. Returns the value at `ptr` prior to the operation.

```
T atomic_xor(T* ptr, T val)
```

Atomically reads the T value pointed to by `ptr`, bitwise-XORs `val` with it and writes the result back to `ptr`. Returns the value at `ptr` prior to the operation.

```
void atomic_store(T* ptr, T val)
```

Atomically stores `val` at `ptr`.

```
T atomic_load(T* ptr)
```

Atomically reads the T value pointed to by `ptr` and returns it.

Table 12: atomic functions

A simple example of how to use these atomic functions, either with C++ `std::atomic` or using the freestanding functions. Note that both kernels compute the same thing.

```
struct counters_t {
    atomic<uint32_t> sum;
    atomic<uint32_t> count;
};

kernel void cpp_atomics(buffer<counters_t> counters) {
    counters->sum += global_id.x;
    ++counters->count;
}

kernel void raw_atomics(buffer<uint32_t> counters) {
    atomic_add(&counters[0], global_id.x);
    atomic_inc(&counters[1]);
}
```

Listing 11: exemplary use of C++ style and freestanding atomic functions

For the sake of completeness, listed below is the (C++ pseudo-code) fallback implementation if an atomic operation is not supported on a specific type or not at all (the aforementioned compare-and-exchange loop). Note that this is intended for 64-bit atomics, 32-bit atomics make use of `uint32_t` instead. This repeatedly computes the specific operation with the provided `val` and value at `ptr`, until the expected value is stored at `ptr`.

```
T atomic_operation(T* ptr, T val) {
    for(;;) {
        const auto expected = *ptr;
        const auto desired = operation(expected, val);
        if(atomic_cmpxchg((address_space uint64_t*)ptr,
                        *(uint64_t*)&expected,
                        *(uint64_t*)&desired) == *(uint64_t*)&expected) {
            return expected;
        }
    }
}
```

Listing 12: atomic fallback implementation

Intra-work-group synchronization between work-items is important for various algorithms. Since work-items in a work-group are executed asynchronously, it becomes necessary to have synchronization points (barriers and fences) to make sure that all work-items have executed up to a certain point or accessed certain memory at a certain point. Note that the listed barrier functions must be encountered by all work-items in the work-group at the exact same point, hence if located inside a branch, all work-items must have taken the exact same path, otherwise undefined behavior ensues⁷. Note also that the following distinction between global and local memory can result in performance improvements over synchronizing everything, but does require that this functionality is supported by the hardware and backend, otherwise it will simply fallback to a full `barrier`.

```
void barrier()
```

This is the full-on barrier, making sure that all work-items have executed up to this point and that all global and local memory accesses have completed.

```
void global_barrier()
```

```
void local_barrier()
```

Similar to `barrier`, except that it only makes sure that all global or local memory accesses have completed.

⁷this can result in a GPU reset or complete deadlock

```
void global_mem_fence()
void local_mem_fence()
```

This ensures that all global or local memory reads and writes have completed for the work-item and are visible to the other work-items in the work-group.

```
void global_read_mem_fence()
void local_read_mem_fence()
```

This ensures that all global or local memory reads have completed for the work-item and are visible to the other work-items in the work-group.

```
void global_write_mem_fence()
void local_write_mem_fence()
```

This ensures that all global or local memory writes have completed for the work-item and are visible to the other work-items in the work-group.

Table 13: synchronization functionality

To facilitate the specialization of code depending on the used device and platform, several `device_info` functions are supported (in a thusly-named namespace). Note that all of these functions are `constexpr` so that they can be used for template specialization at compile-time. Also note that these are available as compile-time defines as well, but omitted here for brevity.

```
constexpr TYPE type()
```

Returns GPU or CPU.

```
constexpr bool has_fma()
```

Returns true if the device has native support for fused-multiply-add instructions.

```
constexpr bool has_64_bit_atomics()
```

Returns true if the device supports 64-bit atomic operations.

```
constexpr bool has_native_extended_64_bit_atomics()
```

Returns true if the device has native support for the extended set of atomic operations (min, max, AND, OR, XOR). This is currently only the case for CUDA `sm_32` or higher devices and OpenCL devices supporting `cl_khr_int64_extended_atomics`.

```
constexpr bool has_pointer_atomics()
```

Returns true if the device is 32-bit (as 32-bit atomics are always supported), or if the device supports 64-bit atomic operations if it is 64-bit. How useful atomic operations on pointers are is however a different matter.

```
constexpr bool has_dedicated_local_memory()
```

Returns true if the device has dedicated local memory hardware. This is the case for all GPUs, but will certainly be false for all CPUs.

```
constexpr uint32_t simd_width()
```

Returns the native SIMD-width (or max SIMD-width) of the device. This is a fixed value for NVIDIA GPUs (32), AMD GPUs (64) and Intel GPUs (16, although it is potentially variable at run-time). For CPUs this returns at least 4 (SSE or NEON is always supported), 8 with AVX/AVX-2 and 16 with AVX-512 support. Note that this is only the upper limit for CPUs as code might not always be vectorized.

```
constexpr VENDOR vendor()
```

Returns the vendor of the device (AMD, APPLE, HOST, INTEL, NVIDIA, UNKNOWN).

```
constexpr VENDOR platform_vendor()
```

Returns the platform vendor (same values as `vendor`), i.e. variable for OpenCL, NVIDIA for CUDA, APPLE for Metal and HOST for Host-Compute.

```
constexpr OS os()
```

Returns the operating system that is currently in use (FREEBSD, IOS, LINUX, OSX, WINDOWS, UNKNOWN).

Table 14: device information functions and macros

3.3.3. Image Functionality

Image functionality plays an important role in Compute just as it does for Graphics, since it allows data to be stored and accessed in a fashion that might be easier to comprehend than simple arrays, furthermore being able to make use of additional caching hardware (which is not to be underestimated: Chapter 4.2.3.), and ultimately necessary for proper interoperability between Compute and Graphics. Providing this image functionality in a unified C++ interface has been one of the major challenges of this project, but could successfully be solved in the end. The difficulty lies with the limitations and restrictions imposed by each backend, as well as the differences between them. Nevertheless, functionality now goes beyond what is currently possible in either CUDA, Metal or OpenCL device languages, allowing single-object read-write images and the use of images in templates, and generally providing an easy to use interface.

CUDA [NVI15b][NVI15e] divides images into textures [NVI15g] (read-only, always combined with a sampler) and surfaces [NVI15f] (read-write, but no sampler support and potentially uncached). Sampler support here is mainly required for being able to use normalized floating point texture coordinates and perform linear sampling (among other things). As a consequence, both textures and surfaces are necessary to implement all image functionality, as textures can not be written to and surfaces do not support all read operations and modes. Another difficulty with surfaces is that they require the exact data location and data size that needs to be written (or read), making generic write support for `float`, `int` and `uint` data non-trivial, e.g. differentiation of a `float` write to either a 32-bit float surface or normalized 8-bit unsigned char surface must happen at run-time. Therefore, it is currently advised to at least specify the channel count of all writable images, so as to keep the possible write cases at a minimum. CUDA further distinguishes between texture/surface references (pointers) and objects, the former required to be known at compile-time and with certain Host API limitations, the latter possible to be only known at run-time (hence also known by the name bindless image/texture in OpenGL) and with fewer Host API problems. As bindless images are less problematic to use (less cruft) and considering the future extension of bindless images everywhere, I have decided to directly go with an object/bindless only implementation. This however limits the image functionality to `sm_30` (Kepler) or higher hardware (which has however been around for almost four years now). All CUDA image functions are implemented through OpenCL-like intrinsic `read_image/write_image` calls which are then transformed to the appropriate PTX assembly inside the compiler (CUDAFirst pass). This was not done on the library side due to inline assembly capabilities and NVPTX intrinsic calls being severely limited (as in: would have required some thousands of lines of template

code to support all image and parameter combinations, which is not a good idea for anyone). Note that CUDA has several issues when using OpenGL interoperability: a) depth images are not directly supported, so that they're now emulated in software internally by mirroring the image data to an R32F image, which can then be used by Compute, b) OpenGL cube map images can not be shared (there is partial support for this in CUDA, but it's completely defunct), c) 3-channel RGB images can not be shared or used.

OpenCL [KG14b][OWG12] and Metal [App15a][App15d] handle images almost identically, only differing in how they name things. Both require that images are only passed to the kernel through opaque struct pointers (also known as opaque image types, opaque because the size and content of the struct is unknown), actively prohibiting them to be obtained through any other means or stored inside other memory objects, thus not providing bindless functionality like CUDA does at this point. Consequently, wrapping this opaque pointer in a nice to use C++ interface seems impossible at first sight, even if the objects content would be directly visible to any vendor compiler. Fortunately, Clang possesses functionality called aggregate (or structure) expansion that it uses for optimization purposes and certain special built-in types, which expands a structure function argument to all of its structure members, resulting in as many function arguments instead. The important part here being that it allows a pointer previously wrapped inside a structure to be unwrapped again inside the compiler, enabling the creation of a nice user interface while still complying to OpenCL/Metal limitations. Note that further extensions inside Clang were made so that this works with inheritance as well and properly recognizes image types and their attributes. Another restriction of OpenCL and Metal, that images are only either readable or writable, has also been lifted through the same functionality by simply storing a read-only and a write-only image pointer inside the class, allowing read-write support through the same user-visible image object. The actual backend-specific read and write image functions are actually quite well designed and easy to handle (compared to CUDA), making it possible to directly call them on the library side.

Host-Compute of course required me to implement all of this in software as well, which has been done for all image base types (except MSAA) and the most common image formats: 2-, 4-, 8-, 16-, 32- and 64-bit non-normalized signed and unsigned integer, 16-, 32- and 64-bit floating point, and 8-bit and 16-bit normalized signed and unsigned integer. MSAA images are not supported as there is no viable way of reading and writing MSAA textures with OpenGL on the host, and I don't see much point to supporting this by itself when there is no OpenGL interop support. For all other types it appears that this backend has the most extensive image support. Other than that, nothing special is to report here.

Note that at this point no user-visible sampler types are supported as the differences between CUDA and OpenCL/Metal are simply too large, with CUDA requiring tight texture/sampler combinations that have to be created on the host, and OpenCL/Metal more liberally allowing standalone samplers and combination of images and samplers both at compile-time inside the program (the preferred method) and at run-time through host-created samplers. Most use cases that would require samplers should however be covered by allowing nearest and linear sampling, as well as the use of integer and normalized floating point coordinates all on the same user-visible image object (and handling the backend-specific samplers behind the scenes). At the same time, this makes image functions easier to use.

The library source code for this can also be found in the `compute/device` folder (anything that has `image` in its name).

As there are many different types of images and support for these varies wildly between backends, have a look at the table below to know what is supported where.

Type	CUDA		OpenCL		Metal		Host Compute	
	read	write	read	write	read	write	read	write
1D								
1D Array								
1D Buffer								
2D								
2D Array								
2D Depth	partial ¹		extension ⁵					
2D Depth Array	partial ¹		extension ⁵					
2D MSAA	Kepler ²		ext ^{5,6}				no ¹¹	
2D MSAA Array	Kepler ²		ext ^{5,6}				no ¹¹	
2D Depth MSAA	partial ^{1,2}		ext ^{5,6}				no ¹¹	
2D Depth MSAA Array	partial ^{1,2}		ext ^{5,6}				no ¹¹	
3D				ext ⁴				
Cube	partial ³		no ⁷					
Cube Array	partial ³		no ⁷		OS X only ⁹			
Cube Depth	partial ^{1,3}		no ⁷					
Cube Depth Array	partial ^{1,3}		no ⁷		OS X ⁹			
any Depth + Stencil			partial ⁸		partial ¹⁰		yes ¹²	

¹ CUDA does not support OpenGL depth image sharing, but can simply read images if they are using a normal 32-bit float format (no depth/non-depth distinction)

² requires `sm_30` or higher (Kepler or higher)

³ while supported by CUDA itself, there is no functional OpenGL sharing support

⁴ requires extensions `cl_khr_3d_image_writes` (supported by AMD, Apple, Intel)

⁵ requires extensions `cl_khr_depth_images`, and `cl_khr_gl_depth_images` when using OpenGL sharing (supported by AMD, Apple, Intel)

⁶ requires extension `cl_khr_gl_msaa_sharing` (implies ⁵), can only be created through OpenGL (supported by Apple, Intel on Windows only)

⁷ not supported at all and no extension in sight

⁸ format is supported (with ⁵), but can only read and write the depth part, not the stencil part

⁹ OS X only, not supported by iOS / PowerVR hardware

¹⁰ format is supported, but can only read and write the depth part, not the stencil part

¹¹ not supported due to missing OpenGL support

¹² since this is a rather asymmetric format, it returns or takes a `pair<float, uint8_t>`

Table 15: image read and write support across backends

```
const_image_*<typename sample_type>
image_*<typename sample_type, bool write_only = false>
```

These are the two image types intended for main use. In a C++ manner, the constant read-only image type is called `const_image`, while the read-write by default type is called `image`. Since read-write capability is not always wanted or necessary, `image` also accepts an optional `bool` template parameter to make it write-only. Alias template definitions for all base image types exist and are detailed below. Note that no writable MSAA type exists, because no backend currently supports this.

1D	<code>image_1d</code> and <code>const_image_1d</code>
1D Array	<code>image_1d_array</code> and <code>const_image_1d_array</code>
2D	<code>image_2d</code> and <code>const_image_2d</code>
2D Array	<code>image_2d_array</code> and <code>const_image_2d_array</code>
2D Depth	<code>image_2d_depth</code> and <code>const_image_2d_depth</code>
2D Depth Array	<code>image_2d_depth_array</code> and <code>const_image_2d_depth_array</code>
2D MSAA	<code>const_image_2d_msaa</code> only
2D MSAA Array	<code>const_image_2d_msaa_array</code> only
2D Depth MSAA	<code>const_image_2d_depth_msaa</code> only
2D Depth MSAA Array	<code>const_image_2d_depth_msaa_array</code> only
3D	<code>image_3d</code> and <code>const_image_3d</code>
Cube	<code>image_cube</code> and <code>const_image_cube</code>
Cube Array	<code>image_cube_array</code> and <code>const_image_cube_array</code>
Cube Depth	<code>image_cube_depth</code> and <code>const_image_cube_depth</code>
Cube Depth Array	<code>image_cube_depth_array</code> and <code>const_image_cube_depth_array</code>

```
const_image<COMPUTE_IMAGE_TYPE>
image<COMPUTE_IMAGE_TYPE, bool write_only = false>
```

These types are mostly intended for internal use (and is how the other image types are implemented), but might also be useful when specification of the full image type is wanted.

```

T read(coord_type coord)
T read(coord_type coord, uint32_t layer)
T read(coord_type coord, uint32_t sample)
T read(coord_type coord, uint32_t layer, uint32_t sample)

```

Reads the pixel at the specified coordinate, image layer (if specified and the image is a layered type) and/or sample index (if the image is of MSAA type) and returns it. If `coord_type` is of floating point type, the coordinate is assumed to be a normalized coordinate in $[0,1]$ for each dimension. Alternatively, if `coord_type` is of integral type, the coordinate is assumed to be a non-normalized absolute coordinate in $[0, \text{image_dimension}[d] - 1]$ for each dimension d . Note that this function always performs nearest sampling. The return type is either: a) `vector4<scalar_type>` if the declared `sample_type` of the image is a scalar type (`float`, `int32_t` or `uint32_t`) or specifically a 4-component vector type (`vector4<scalar_type>`), b) `scalar_type` if `sample_type` is a `vector1<scalar_type>` 1-component vector, c) `vector2<scalar_type>` if `sample_type` is a `vector2<scalar_type>` 2-component vector, or d) `vector3<scalar_type>` if `sample_type` is a `vector3<scalar_type>` 3-component vector. It is thus possible to set a specific channel count.

```

T read_linear(coord_type coord)
T read_linear(coord_type coord, uint32_t layer)
T read_linear(coord_type coord, uint32_t sample)
T read_linear(coord_type coord, uint32_t layer, uint32_t sample)

```

This is identical to `read`, except that it always performs linear sampling. Note that this practically only makes sense with floating point coordinates.

```

void write(coord_type coord, T data)
void write(coord_type coord, uint32_t layer, T data)

```

Writes the pixel value specified by `data` to the specified `coord` coordinate and `layer` (if specified and the image is a layered type) in the image. Note `coord_type` must be of integral type and is thus a non-normalized absolute coordinate. Note again that since MSAA image writing is not supported, no function taking a `sample` parameter currently exists.

```

static constexpr COMPUTE_IMAGE_TYPE type()

```

Returns the underlying `COMPUTE_IMAGE_TYPE` type of the image. Note that this is more generic on the device than it is on the host, as multiple formats must be supported for the same `sample_type`. Note that, like the following functions, this is mostly intended for use in templates.

```
static constexpr uint32_t channel_count()
```

Returns the corresponding channel count of the image if `sample_type` is a libfloor vector type, or always 4 if it is a scalar type.

```
static constexpr bool is_readable()
```

Returns true if the image can be read from (it is not write-only).

```
static constexpr bool is_writable()
```

Returns true if the image can be written to (it is not read-only).

```
static constexpr bool is_read_only()
```

```
static constexpr bool is_write_only()
```

```
static constexpr bool is_read_write()
```

Returns true if the image is read-only, write-only or read-write respectively.

Table 16: image types and functions

```
template <typename image_type, typename coord_type>
auto albedo(const image_type& img, const coord_type& coord) {
    return img.read(coord).xyz.dot(float3 { 0.222f, 0.7067f, 0.0713f });
}
kernel void rgb_to_albedo(const image_2d<float> input_img,
                        image_2d<float1, true> output_img) {
    output_img.write(global_id.xy, albedo(input_img, global_id.xy));
}
```

Listing 13: exemplary use of images

3.3.4. Misc Functionality

CUDA, OpenCL and Host-Compute have support for `printf` in device code. Note that this is an optional feature in OpenCL that might not be supported by the hardware or implementation. In CUDA I had to implement this via a variadic template function that manually builds the `va_list` that is then passed to the `vprintf` device ABI function as no `printf` function actually exists (thus, beware that this is not a C-style variadic function, but a template function).

Additionally, I also created a small wrapper around `printf` callable through `print`, which is also implemented as a variadic template function. This is mostly intended as a simplified and safe replacement for `printf`. Instead of `%` format options this only supports `$` placeholders which will automatically be set to the correct `printf` format option at compile-time. Furthermore, this makes sure that the necessary amount of parameters was specified (at compile-time). Note that this also has direct

support for libfloor vector and matrix types, only requiring a single \$ placeholder for objects of these types. A newline character will always be appended to the end automatically.

```
float3 vec { -1.0f, 23.0f, 42.0f };
// prints "good old printf: 123.456001, -1, -1.000000\n"
printf("good old printf: %f, %d, %f\n", 123.456f, -1, vec.x);
// prints the same
print("good old printf: $, $, $", 123.456f, -1, vec.x);
// prints "also much easier: (-1.000000, 23.000000, 42.000000)\n"
print("also much easier: $", vec);
```

Listing 14: console printing in device code

3.3.5. STL

The STL implementation is provided by libc++ 3.5, with some system C library headers provided by Clang itself (e.g. `stddef.h` and assorted others). As Compute has several restrictions, it is not directly possible to use the unmodified libc++ STL code, hence code needed to be made compatible. Note that some restrictions are hard restrictions that will lead to compile-time errors, while others are soft-ish restrictions that let unsupported code to compile without errors, but won't actually be usable at run-time or when directly called at compile-time. Furthermore, Clang/LLVM allow certain memory functions that would otherwise clearly be run-time functions to be executed at compile-time under the condition that the parameters and destination are known at compile-time. To some degree it also possible to call these at run-time as CUDA and OpenCL explicitly support these, with Metal supporting them as well, but not explicitly mentioning it. These functions are `__builtin_memcpy`, `__builtin_memmove` and `__builtin_memset` and act as drop-in replacements for `memcpy`, `memmove` and `memset`. All occurrences of these run-time functions were replaced with the built-in functions in supported STL headers. Generally, if these can be folded to simple loads and stores at compile-time, or completely optimized away because there where just used on a temporary objects, they will work just fine. The STL code is provided separately and is always part of the toolchain in the folder named `libcxx`.

The STL headers listed below were made compatible and can thus be used in Compute code. Note that all of these are currently included automatically as the device library makes use of all of them (or indirectly as a dependency of another header).

- `<type_traits>`: important SFINAE functionality, `enable_if_t`, `conditional_t`, `is_*`, ...
- `<utility>`: `pair`
- `<tuple>`: `tuple`, `tuple_cat`, `make_tuple`, ...
- `<algorithm>`: `min`, `max`, `sort` (with private memory), `find`, ...
- `<array>`: supported, but prefer libfloor's `const_array` for `constexpr` use
- `<functional>`: misc function wrappers (prefer lambdas)
- `<memory>`: mostly for compilation purposes, limited `unique_ptr` support if it can be folded
- `<iterator>`: `iterator`, `reverse_iterator`, `const_iterator`, ...
- `<iosfwd>`: for compilation only
- `<atomic>`: as described in [ISO14], lacking `atomic_flag` support (prefer `atomic<uint32_t>`)
- `<numeric_limits>`: native types limits

Also note that the remaining portion of unsupported C++ STL headers concerns run-time functionality that is not supported on compute devices right now (strings, streams, vector, mutexes, ...).

4. Application

To demonstrate that all of this actually works as described, multiple applications were written, either to showcase certain aspects of Compute and C++, or to demonstrate more useful applications. The code for this can be found in the `floor_examples` and `libwarp` folders (or resp. repositories, Appendix B.).

4.1. Image-space Warping

A major application focus has been on making image-space warping work on all Compute platforms, which has been accomplished. Furthermore, all of this functionality has been wrapped in a separate library that can be used by external projects.

Image-space warping can be used to extrapolate additional frames from an existing frame that has been rendered with 3D rendering methods. This extrapolation does not require re-rendering of the 3D scene, but exploits the spatial and temporal coherence between frames [NSL⁺07], allowing it to reuse the samples of the originating frame and furthermore move them forwards in time according to per-pixel motion flow [DER⁺10] that is computed for each originating frame. Overall, the intent is to upsample from a lower input frame rate to higher frame rate without costly rendering of the in-between frames and instead replacing it with a much cheaper interpolation method using existing frames and scene information.

There are essentially two different methods of computing the image-space warping. One is a scatter-based approach, i.e. pushing (scattering) pixels to where they are supposed to be. The other is a gather-based approach, somehow searching for the pixel that is supposed to be at a specific location. More comprehensive explanations can be found in the following two sections.

4.1.1. Scatter approach

For scattering, the input motion flow can be computed using two different methods. Either in 2D where each pixel stores the on-screen 2D motion from the previous to the next frame, or in 3D where each pixel stores the camera-space scene 3D motion, which can then however come from the previous to the next frame motion again, or through some other means. For this, I have chosen to use and implement 3D motion as it allows the motion to come from anywhere, i.e. it is possible to use either motion from the previous to the next frame (as for 2D) or compute and use some kind of predicted motion of the camera. Note that the renderer can thus write anything as the motion vector and the scatter kernel(s) will accept it. Additionally,

for 2D motion to look correct, the camera input would need to lag behind by one frame (which may or may not be desirable), as it would otherwise use the previous' frame motion and thus scatter pixels to incorrect positions, resulting in a much too distorted and warped look. For 3D motion, this kind of distortion can be prevented even when the motion vector is not completely correct, i.e. the 3D motion of the previous frame is much more likely to be similar to the current frame than that is the case for on-screen 2D motion, where even minor errors become quickly visible.

When computing the scatter pixel destination, this first reconstructs the camera-space 3D pixel position using the pixels depth value and general camera setup (quite similar to image-space shading methods [Wen06]), requiring the screen width and height, field-of-view, near and far plane, corresponding to the projection matrix setup of [KG15e, 12.1.1]. Using an inverse projection matrix would be possible as well, but a matrix multiplication would unnecessarily waste resources, and doing the math separately allows some of it to be already folded at compile-time (at run-time now just 2 ADDs, 2 FMAs, 2 MULs, 1 negate). Next, from the reconstructed 3D position the pixel is moved along the pixels motion vector under consideration of the time delta. This 3D position is then projected back into 2D, which then represents the scattered-to screen location and is thus used to write the pixel color.

To store the per-pixel 3D motion vector a 16-bit or 32-bit RGBA float texture could have been used, but this would require too much bandwidth and memory. Instead, a single-channel 32-bit unsigned integer texture is used, trading some additional math instructions for bandwidth and memory. Encoding is as follows: 3-bit for X/Y/Z signs, 10-bit |X|, 9-bit |Y|, 10-bit |Z|. All motion vectors are capped at 64.0, which usually proves more than enough. The encoding of this is not linear, but base-2 logarithmic, generally giving more precision to pixels that are closer to the camera origin as these have smaller motion vectors, and also allowing efficient encoding and decoding using `log2` and `exp2` hardware instructions.

A major issue with scatter-based approaches is that multiple pixels can be scattered onto the same location, thus either leading to artifacts or making it necessary to perform some sort of depth-testing so that only the closest pixel to the camera is written (depth-correctness). The former is implemented in the example program, but not part of the library. The latter has been implemented, and is part of the library, by employing a two-pass kernel, atomically storing the minimum depth in a buffer in the first pass, then only writing the pixel with the matching (passing) depth in the second pass (implementation further below). This has proven to be the most efficient way of going about this with Compute, as proper depth-testing with scattering is a non-trivial task (any pixel could end up anywhere, making any form of localized cooperative computation impossible). Note that this is quite a good use of global memory atomics as collisions might occur, but aren't that

numerous that it is an issue (i.e. can easily live with 2 or 3 atomic operations on the same address, this is what they are good for after all). This also has the benefit that both kernels are quite simple, making it possible to achieve high throughput.

A remaining issue is that of hole-filling, i.e. filling in the gaps of moved-away pixels that have not been filled with another (moved-to) pixel. Right now, this can be fixed by either not clearing the color image (thus keeping the color from the previous frame), or by running another kernel that for each not filled in pixel samples the surrounding pixels, computes their average and sets the pixel color to this average. I consider both methods insufficient, but usable, with the latter easily fixing the common missing single-pixel. Either way, this problem remains one of the major downsides of a scatter-based approach that is more easily circumvented by using a gather-based approach, albeit that it just gives more plausible results, as there is simply no way of correctly solving this problem as these pixels are inherently missing.

4.1.2. Gather approach

The gather-based approach is fully based on the image-based interpolation of [YTS⁺11, 5.], with some detail modifications and the general addition that it can also be used in a forward-only fashion, thus having equal memory and bandwidth requirements as the scatter-based approach by trading off quality.

The basis of this is that both the source and the destination frame are already rendered (i.e. frame at time t and at time $t+1$), including the relative depth change and 2D on-screen motion for each pixel from the source to the destination frame, and from the destination to the source (also called flow fields in the paper). The reason why this is done bidirectionally is that this reduces disocclusion problems, since a pixel is very likely to be in at least either one of the two frames. This generally improves the accuracy and thus the resulting image quality.

Concerning the gather part, this operates in a reverse fashion to scatter: instead of moving a pixel around, it searches for the pixel that is supposed to be at a screen location. To achieve that, this performs an iterative greedy search for the correct pixel in the rendered image by utilizing the rendered 2D motion image (flow field). This is done both in the forwards direction (from t to $t+1$) and in the backwards direction (from $t+1$ to t). When this is computed, it additionally computes the clip-space depth of the pixel and a screen-space error value that is essentially dependent on the distance of the found pixel and how much motion there is, thus giving a rough estimate about how "wrong" a pixel probably is. Finally, under consideration of the forwards and backwards screen-space error value (checking if they're below a certain threshold) and either depth values (checking if one occludes the other),

it either computes an interpolated color from both the forwards and backwards direction, or just uses either one if the other one didn't pass the checks.

As mentioned, this requires that the two next frames are already rendered, resulting in the general render and compute order as follows: ..., frame t , warped frames $t - 1 \rightarrow t$, frame $t + 1$, warped frames $t \rightarrow t + 1$, frame $t + 2$, ...

Note here that each (non-warped) frame computes the $t \rightarrow t + 1$ forwards motion and relative depth change, as well as the backwards $t \rightarrow t - 1$ ones. In total, this adds up to rendering to two 32-bit uint images (motion) and one two-channel 16-bit half-float image (depth, half precision is enough here) for each frame.

Just as with the motion vector encoding for the scatter-approach, this also uses a single-channel 32-bit unsigned integer texture, the encoding is however more trivial. Since the 2D motion vector is in $[-1, 1]$, this could have directly used a two-channel 16-bit float texture, but this quickly runs into precision problems and is not future-proof for higher screen resolutions (anything above 2048 would not even be pixel-accurate any more [Wik15f]). Instead, the 2D motion vector is scaled up to $[-32767, 32767]$, so that each component fits into a signed 16-bit integer and makes full use of that range. These are then combined and stored in the 32-bit uint, which has the additional advantage that the motion image format is compatible to the scatter one.

The major upside of the gather-based approach and bidirectionality in particular is that it looks qualitatively very good and superior to the scatter-based approach as it doesn't leave any holes in the final image and disocclusions are generally handled a lot better. Even upsampling from low frame rates such as 5 FPS still produces usable results that might be of use for certain applications. It should also be noted that it can make use of linear sampling for the color image, compared to the scatter approach that is obviously limited to nearest sampling, generally resulting in a smoother (less wavy) look. The major downside of the bidirectional approach is however that it requires that frames are rendered 2 steps ahead, thus requiring input handling 2 steps ahead as well, leading to significant input lag. The general recommendation of the paper is that everything should be handled with at least 15 FPS to hide most of this issue, with 30 FPS producing even better results. I have to concur that 30 FPS should be the minimum frame rate to properly hide the input lag, with the additional benefit that it simultaneously produces images that are almost indistinguishable from real frames. Another problem are the memory and bandwidth requirements that come with storing two full images and all the information that is needed for this gather approach. Including 8-bit RGBA color and 32-bit depth information, this sums up to a requirement of 160-bit per pixel per frame (320-bit for both), compared to just 96-bit that is required for the scatter approach. While most GPUs can easily handle this today, a decision between

quality and less memory and bandwidth still has to be made. As an in-between solution, a forward-only gather implementation is also provided, having the same memory requirements as the scatter approach (discarding the second image and relative depth completely, as it is no longer of use), but obviously sacrificing quality and disocclusions benefits of the bidirectional approach, but also improving input lag issues as it now only requires input 1 step ahead. Optionally, in a similar fashion to [DER⁺10] artifacts are attempted to be reduced by applying a blur to them. As with the bidirectional approach before, the screen-space error and a certain threshold can be used to estimate if a pixel is incorrect, but instead of being able to use the color sample from the other frame, this will now compute a directional blur in the direction of the pixel motion vector (which is already already present), and use this blurred result for the pixel color instead.

4.1.3. Library

Another focus has been on making the scatter-based and gather-based warping available in a library that can easily be used in other projects. For compatibility, the API is completely written in C, with the actual implementation written in C++. All libfloor initialization, kernel compilation and OpenGL/Metal texture wrapping is handled automatically. The actual image creation and correct motion vector and depth rendering must however be performed by the user, as this is obviously dependent on the employed rendering pipeline and any shader code. Code snippets that can simply be inserted into GLSL or Metal shaders are however available.

```
// scatter-based warping
LIBWARP_ERROR_CODE libwarp_scatter(
    const libwarp_camera_setup* const camera_setup,
    const float delta,
    const uint32_t color_texture,
    const uint32_t depth_texture,
    const uint32_t motion_texture,
    const uint32_t output_texture);
// bidirectional gather-based warping
LIBWARP_ERROR_CODE libwarp_gather(
    const libwarp_camera_setup* const camera_setup,
    const float delta,
    const uint32_t color_current_texture,
    const uint32_t depth_current_texture,
    const uint32_t color_prev_texture,
    const uint32_t depth_prev_texture,
    const uint32_t motion_forward_texture,
    const uint32_t motion_backward_texture,
    const uint32_t motion_depth_forward_texture,
    const uint32_t motion_depth_backward_texture,
    const uint32_t output_texture);
// forward-only gather-based warping
LIBWARP_ERROR_CODE libwarp_gather_forward_only(
    const libwarp_camera_setup* const camera_setup,
    const float delta,
    const uint32_t color_texture,
    const uint32_t motion_texture,
    const uint32_t output_texture);
```

Listing 15: libwarp OpenGL API

Identical functions with id <MTLTexture> instead of `uint32_t` for use with Metal exist as well.

```
typedef enum {
    //! normalized in [0, 1], default for OpenGL and Metal
    LIBWARP_DEPTH_NORMALIZED,
    //! z/w depth (manually written to a R32F texture in the shader)
    LIBWARP_DEPTH_Z_DIV_W,
    //! linear depth [0, far-plane]
    LIBWARP_DEPTH_LINEAR,
} LIBWARP_DEPTH_TYPE;
typedef struct libwarp_camera_setup {
    uint32_t screen_width;
    uint32_t screen_height;
    float field_of_view;
    float near_plane;
    float far_plane;
    LIBWARP_DEPTH_TYPE depth_type;
} libwarp_camera_setup;
```

Listing 16: libwarp camera setup

4.1.4. Implementation

The following code listings show how the two-pass scattering and forward-only gathering are implemented, including most helper functions. Note that code and comments were abbreviated and packed to fit in here.

```
// computes the "scattered" destination coordinate of the pixel at 'coord'
// according to it's depth value (which is also returned) and motion vector,
// as well as the current time delta
static auto scatter(const int2& coord,
                   const float& delta,
                   const_image_2d_depth<float> img_depth,
                   const_image_2d<uint1> img_motion) {
    // read rendered/input depth and linearize it
    // (linear distance from the camera origin)
    const auto linear_depth =
        warp_camera::linearize_depth(img_depth.read(coord));
    // get 3d motion for this pixel
    const auto motion = decode_3d_motion(img_motion.read(coord));
    // reconstruct 3D position from depth + camera/screen setup,
    // then predict/compute new 3D position from current motion and time
    const auto new_pos =
        warp_camera::reconstruct_position(coord, linear_depth) + delta * motion;
    // -> return
    const struct {
        const int2 coord;
        const float linear_depth;
    } ret {
        // project 3D position back into 2D
        .coord = warp_camera::reproject_position(new_pos),
        .linear_depth = linear_depth
    };
    return ret;
}
```

Listing 17: the general scatter function employed by all kernels

```

// NOTE: compile-time defines: SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_FOV,
// WARP_NEAR_PLANE, WARP_FAR_PLANE, DEFAULT_DEPTH_TYPE (normalized or z/w)
namespace warp_camera {
    static constexpr const float2 screen_size
        { float(SCREEN_WIDTH), float(SCREEN_HEIGHT) };
    static constexpr const float2 inv_screen_size { 1.0f / screen_size };
    static constexpr const float aspect_ratio { screen_size.x / screen_size.y };
    // projection up vector
    static constexpr const float _up_vec
        { const_math::tan(const_math::deg_to_rad(SCREEN_FOV) * 0.5f) };
    // projection right vector
    static constexpr const float right_vec { _up_vec * aspect_ratio };
#if defined(SCREEN_ORIGIN_LEFT_BOTTOM) // OpenGL: left bottom, Metal: left top
    static constexpr const float up_vec { _up_vec };
#else // flip up vector for "left top" origin
    static constexpr const float up_vec { -_up_vec };
#endif
    static constexpr const float2 near_far_plane
        { WARP_NEAR_PLANE, WARP_FAR_PLANE };

    // reconstructs a 3D position from a 2D screen coordinate and
    // its associated real world depth
    static float3 reconstruct_position(const uint2& coord,
                                       const float& linear_depth) {
        return {
            ((float2(coord) + 0.5f) * 2.0f * inv_screen_size - 1.0f) *
                float2(right_vec, up_vec) * linear_depth,
            -linear_depth
        };
    }
    // reprojects a 3D position back to 2D
    static float2 reproject_position(const float3& position) {
        const auto proj_dst_coord = (position.xy *
            float2 { 1.0f / right_vec, 1.0f / up_vec }) / -position.z;
        return ((proj_dst_coord * 0.5f + 0.5f) * screen_size);
    }
    // linearizes the input depth value according to the depth type and
    // returns the real world depth value
    template <depth_type type = DEFAULT_DEPTH_TYPE>
    constexpr static float linearize_depth(const float& depth) {
        if(type == depth_type::normalized) { // depth normalized in [0, 1]
            constexpr const float2 near_far_projection {
                -(near_far_plane.y + near_far_plane.x) /
                    (near_far_plane.x - near_far_plane.y),
                (2.0f * near_far_plane.y * near_far_plane.x) /
                    (near_far_plane.x - near_far_plane.y),
            };
            return near_far_projection.y / (depth - near_far_projection.x);
        }
        else if(type == depth_type::z_div_w) { // z/w depth in shader
            // need to perform a small adjustment to account for near/far plane
            return depth + near_far_plane.x -
                (depth * (near_far_plane.x / near_far_plane.y));
        }
        floor_unreachable(); // signal compiler that this is unreachable
    }
};

```

Listing 18: warp_camera namespace with compile-time variables and helper functions

```

static constexpr const float3 signs_lookup[8] {
    float3 { 1.0f, 1.0f, 1.0f },
    float3 { 1.0f, 1.0f, -1.0f },
    float3 { 1.0f, -1.0f, 1.0f },
    float3 { 1.0f, -1.0f, -1.0f },
    float3 { -1.0f, 1.0f, 1.0f },
    float3 { -1.0f, 1.0f, -1.0f },
    float3 { -1.0f, -1.0f, 1.0f },
    float3 { -1.0f, -1.0f, -1.0f },
};
// decodes the encoded input 3D motion vector
// format: [3-bits signs x/y/z][10-bit x][9-bit y][10-bit z]
static float3 decode_3d_motion(const uint32_t& encoded_motion) {
    // lookup into constant memory + 1 shift is faster than 3 ANDs + 3 cmps/sels
    const float3 signs = signs_lookup[encoded_motion >> 29u];
    const uint3 shifted_motion {
        (encoded_motion >> 19u) & 0x3FFu,
        (encoded_motion >> 10u) & 0x1FFu,
        encoded_motion & 0x3FFu
    };
    constexpr const float3 adjust {
        const_math::log2(64.0f + 1.0f) / 1024.0f,
        const_math::log2(64.0f + 1.0f) / 512.0f,
        const_math::log2(64.0f + 1.0f) / 1024.0f
    };
    return signs * ((float3(shifted_motion) * adjust).exp2() - 1.0f);
}

// decodes the encoded input 2D motion vector
// format: [16-bit y][16-bit x]
static float2 decode_2d_motion(const uint32_t& encoded_motion) {
    const union {
        ushort2 us16;
        short2 s16;
    } shifted_motion {
        .us16 = {
            encoded_motion & 0xFFFFu,
            (encoded_motion >> 16u) & 0xFFFFu
        }
    };
    // map [-32767, 32767] -> [-0.5, 0.5]
#ifdef SCREEN_ORIGIN_LEFT_BOTTOM
    return float2(shifted_motion.s16) * (0.5f / 32767.0f);
#else // if the origin is at the top left, flip the y component
    return float2(shifted_motion.s16) *
        float2 { 0.5f / 32767.0f, -0.5f / 32767.0f };
#endif
}

```

Listing 19: 3D and 2D motion decoding

```

// pass #1: compute min depth for each pixel
kernel void warp_scatter_depth(const_image_2d_depth<float> img_depth,
                              const_image_2d<uint1> img_motion,
                              buffer<float> depth_buffer,
                              param<float> delta) {
    const auto scattered = scatter(global_id.xy, delta, img_depth, img_motion);
    if(scattered.coord.x >= 0 && scattered.coord.x < SCREEN_WIDTH &&
        scattered.coord.y >= 0 && scattered.coord.y < SCREEN_HEIGHT) {
        atomic_min(&depth_buffer[scattered.coord.y * SCREEN_WIDTH +
                                scattered.coord.x],
                   scattered.linear_depth);
    }
}

// pass #2: perform depth test and write scattered pixel color if it passes
kernel void warp_scatter_color(const_image_2d<float> img_color,
                              const_image_2d_depth<float> img_depth,
                              const_image_2d<uint1> img_motion,
                              image_2d<float4, true> img_out_color,
                              buffer<const float> depth_buffer,
                              param<float> delta) {
    const auto scattered = scatter(global_id.xy, delta, img_depth, img_motion);
    if(scattered.coord.x < 0 || scattered.coord.x >= SCREEN_WIDTH ||
        scattered.coord.y < 0 || scattered.coord.y >= SCREEN_HEIGHT) {
        return;
    }
    if(scattered.linear_depth <=
        depth_buffer[scattered.coord.y * SCREEN_WIDTH + scattered.coord.x]) {
        img_out_color.write(scattered.coord, img_color.read(global_id.xy));
    }
}

kernel void warp_gather_forward(const_image_2d<float> img_color,
                              const_image_2d<uint1> img_motion,
                              image_2d<float4, true> img_out_color,
                              param<float> delta) {
    const float2 p_init = (float2(global_id.xy) + 0.5f) * warp_camera::inv_screen_size;
    float2 p_fwd = p_init;
    for(uint32_t i = 0; i < 3; ++i) { // search
        const auto motion = decode_2d_motion(img_motion.read(p_fwd));
        p_fwd = p_init - delta * motion;
    }
    const auto motion_fwd = decode_2d_motion(img_motion.read(p_fwd));
    const auto err_fwd = ((p_fwd + delta * motion_fwd - p_init).dot() +
        // large error for out-of-bound access
        ((p_fwd < 0.0f).any() || (p_fwd > 1.0f).any() ? 1e10f : 0.0f));
    float4 color;
    if(err_fwd >= 0.0025f * 0.0025f /* eps */) { // incorrect pixel?
        // compute directional blur in the motion direction of the pixel
        constexpr const auto coeffs = compute_coefficients<TAP_COUNT>();
        constexpr const int overlap = TAP_COUNT / 2;
        const auto dir = motion_fwd.normalized();
        for(int i = -overlap; i <= overlap; ++i)
            color += coeffs[overlap + i] *
                img_color.read(global_id.xy + int2(float(i) * dir));
    }
    else color = img_color.read_linear(p_fwd); // correct pixel
    img_out_color.write(global_id.xy, color);
}

```

Listing 20: putting it all together: two-pass scattering and forward-only gather kernels

4.1.5. Example

An example program of how these two approaches are used in combination with an OpenGL and Metal renderer can be found in the `warp` folder. A video demonstrating the gather-based warping can be found at <https://www.youtube.com/watch?v=PmfJ2kzC49A> or the accompanying DVD.

4.2. Demos & Examples

4.2.1. Path Tracer

This was the original proof-of-concept demo, originally running with OpenCL/SPIR first, then a little later with CUDA as well (Metal and Host-Compute even later). As such, it was to test arguably more or less complex C++ constructs such as classes, templates, recursion through templates, (generic) lambdas, auto, use of global and constant memory, as well as some portions of the support facilities like vector classes and compile-time math. Fundamentally covering all of the crucial base functionality that you want to have in a modern C++ compute language. Most of the original path tracer code was written during the time I took the "Image Synthesis" course, but for this demo it has been condensed down to only its essential parts with further changes so that it is actually possible to run it on compute platforms. Hence, while obviously parallelizable, the code was originally meant to run on CPUs and is as such very branch-heavy and GPU-unfriendly. Another problem was that the original code was recursion-based (not possible on Compute: Chapter 3.1.2.), which has been solved by using template recursion, although the max depth/bounces had to be set to 2, as it proved impossible to compile and run this on weaker hardware and implementations (CUDA and CPU OpenCL can quite easily handle 4 bounces, but trying this with the Intel GPU compiler will keep it busy for a few minutes and finally won't even run properly). Thus, this demo has also served another purpose: showing the limits of current compute compilers and hardware. Another interesting tidbit is that the Cornell Box model and its material information is fully stored in constant memory and actually build into the binary. The employed PRNG is based on [Í 05], with seeds being dependent on the global ID and an initial seed from the host.

4.2.2. N-body Simulation

This was the second demo that I've written for this project and its main purpose was to test and demonstrate cooperative computation through the use of local memory and work-group internal synchronization, graphics interoperability⁸ with OpenGL and Metal, loop unrolling and that high performance computing is indeed possible with this toolchain. All of these I consider essential features that a compute framework must support. This is largely based on [NHP07] with some additional optimizations in the form of full inner loop unrolling and making the damping and softening parameters compile-time parameters (one less operation per body/body interaction). A video demonstration of this can be found at <https://www.youtube.com/watch?v=DoLe1c-eokI> or on the DVD.

Performance was measured for various hardware and backends (with 1 body/body interaction corresponding to 19 floating point operations):

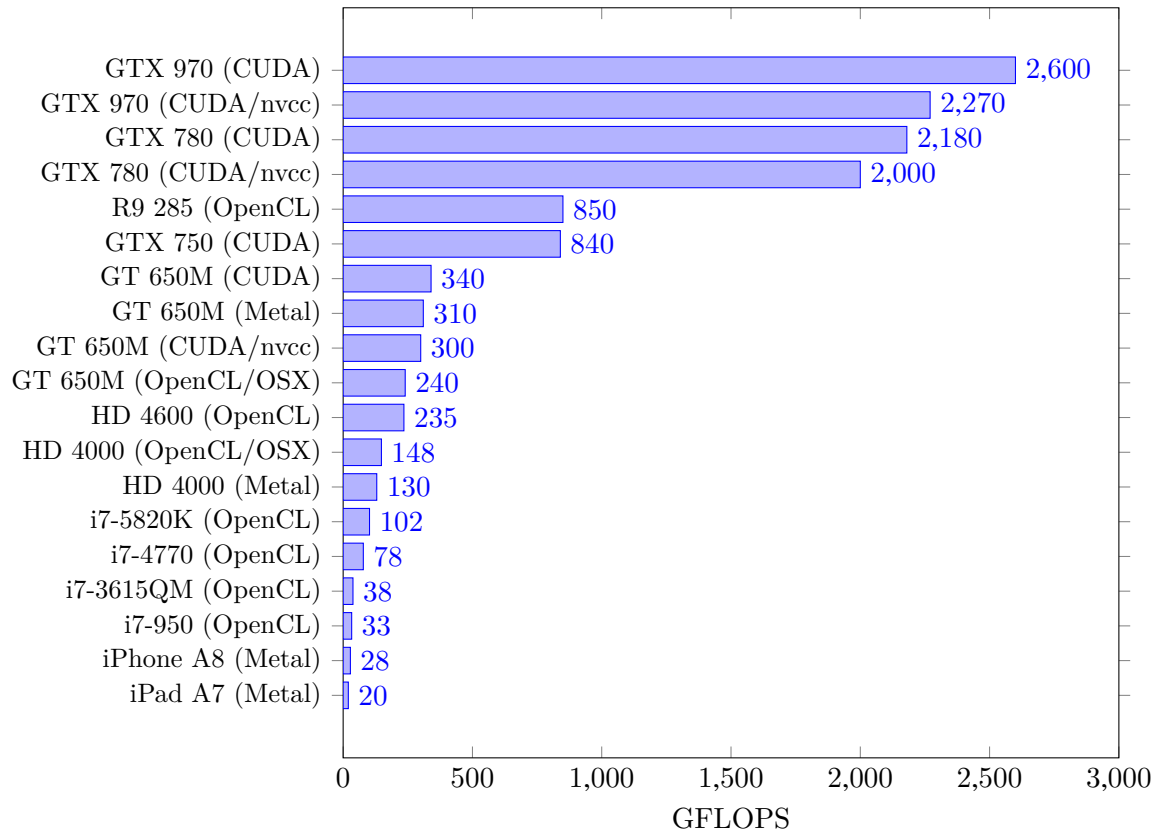


Figure 3: general N-body performance

Several notes and observations here:

- Body count and tile size parameters were tuned for each device to reach peak

⁸there is also a software rasterizer if you like 1980's graphics or X11 forwarding

performance, hence results represent the upper limit that is possible. As devices behave quite differently, anything else would have been unfair.

- Results marked as CUDA/nvcc were obtained through NVIDIA's toolchain (nvcc) and NVIDIA's own N-body example (part of the CUDA SDK under `5_Simulations`). The host setup was otherwise identical, using the same CUDA 7.5 drivers. The performance difference comes down to two reasons: a) the inner loop is fully unrolled instead of just 128 times as NVIDIA does, and b) NVIDIA uses dynamic local memory instead of static local memory, resulting in dynamic offset calculations instead of fully static ones. Note that the NVIDIA example and toolchain are actually at an advantage, since they're counting 20 fp operations per body/body interaction and I am only counting 19 fp operations per body/body interaction due to being able to fold/optimize one more operation (as N-body is largely local memory bandwidth limited, the additional operation comes for free and thus generally improves the result).
- Results marked as OpenCL/OSX were acquired on OS X, all other OpenCL ones were run on Linux using Intel's CPU and GPU OpenCL implementations, or AMD's GPU one.
- AMD's R9 285 is performing surprisingly bad, as I would have expected a result somewhere in the range of 1500 GFLOPS when taking into account max theoretical performance and how NVIDIA GPUs behave compared to their max theoretical performance. From testing with different tile sizes, I can only assume that this is due to synchronization overhead and lower bandwidth local memory, performing at its best when the tile size was identical to the SIMD-width (wavefront size) of 64, and significantly worse when it wasn't (ranging from only 300 to 600 GFLOPS for tile sizes above 64).
- Intel GPUs performed comparatively to NVIDIA GPUs. Tile size behavior is interestingly coupled to compute unit count, with the HD 4600 (20 units) peaking at a size of 80, while the HD 4000 (16 units) peaked at both 128 and 256.
- Apple's A7 and A8 GPUs also perform badly, causing Compute to be generally slow even when only using small buffers with high locality (704 KiB for 16384 bodies, which were necessary to reach peak performance).
- CUDA outperforms both Metal (although quite close) and OpenCL, while OpenCL outperforms Metal for Intel.

Performance of the Intel and AMD OpenCL CPU implementations and Host-Compute were also tested:

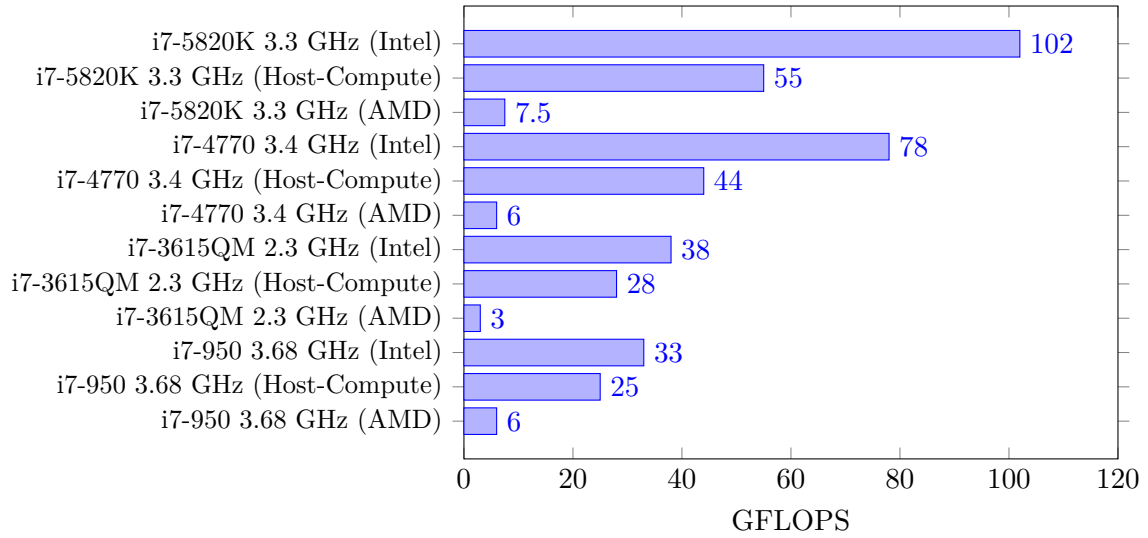


Figure 4: CPU Compute comparison

- Intel's and AMD's OpenCL CPU implementations prefer small tile sizes, peaking at 8 - 32 (close to their logical CPU count and SIMD-width), while Host-Compute prefers large tile sizes, similar to most GPUs, peaking at 1024.
- Host-Compute code (libfloor and nbody) was compiled with SSE4.2 for i7-950, AVX for i7-3615QM, AVX2 for i7-4770 and i7-5820K.
- Intel OpenCL is to be preferred for performance, with Host-Compute not too far off. Consider AMD CPU OpenCL unusable.
- The i7-950 was overclocked from its standard clock speed of 3.07 GHz to 3.68 GHz. Turboboost was enabled for the other CPUs.
- There still seems to be some optimization potential for AVX2 CPUs (and AVX to a lesser degree) when using Clang/LLVM for Host-Compute.

4.2.3. Image Gaussian Blur

This was written with the purpose of being the initial image functionality testing grounds, test how well or bad Compute performs against OpenGL/GLSL, showcase some of the compile-time math computation and perform some more testing of cooperative computation in work-groups. As such, computing the Gaussian blur of a 2D image was chosen as it is simple enough to compute with both Compute and GLSL. This is implemented with kernel weights/coefficients obtained via the Pascal's triangle approximation [Rá10] supporting odd-numbered tap counts of 3 up to 21 (approximating 3*3px to 63*63px sized blurring). Note that at the time of writing this demo, linear sampling was not supported yet, so it always uses nearest sampling (which should however be easier to understand). On the Compute side, this is implemented as both a tile-based approach with local memory caching and cooperative computation, and as a simple two-pass kernel (horizontal and vertical pass, as Gaussian blur is a separable filter [Wik15e]). On the GLSL side only the latter is implemented. For testing purposes an image size of 1024*1024px and a tap count of 17 is used. For the manual caching and cooperative computation approach, this involves about 4 million texture fetches, while for the two-pass method about 36 million texture fetches are required.

The general performance observation is that texture caches on GPU hardware are extremely fast and useful, especially when using it with high-locality like it is done here. This has the effect that the trivial two-pass kernel is usually about 2x to 3x faster than the manual caching and cooperative computation kernel on GPU hardware. For CPU hardware the opposite is true, which is to be expected as they don't have a dedicated texture cache and image functionality is generally quite costly on CPU hardware. Concerning the comparison between Compute and GLSL, both performed equally well (both using the two-pass kernel/shader) with no leaning towards either side being observable.

Note that some of this code can be found in Appendix A.

5. Conclusion & Future

This thesis and projects have successfully shown that a unified C++14 Compute language supporting the majority of platforms and hardware is possible, and provide an implementation building on the LLVM compiler infrastructure, which has proven to be an invaluable tool to accomplish all of this. With C++14, this now provides the most advanced C++ support among all Compute platforms to date, furthermore lifts software restrictions that are imposed by other Compute languages (Chapter 3.1.2.), and should in general be useful for many applications that require or can benefit from the versatility of C++ and the ability to run on many platforms. Compute development and debugging is additionally aided by enabling compute code to solely run on the host alongside a user's program, thus allowing the use of existing host debugging tools.

While this unification was not easy due to the countless minor and major differences between Compute platforms, this project got there in the end. Even more so, this could be achieved without introducing major language quirks that could have easily been the case considering these differences, and it is in fact possible to implement the device language and library entirely in standard C++, as is largely demonstrated by Host-Compute (any nonstandardness is merely for performance reasons). Address space handling proved rather troublesome in the beginning, but the current solution worked out quite well in the end - although a future generic hardware extension that never sacrifices performance is still to be preferred. Image functionality also proved to be incredibly complex, and I would estimate that half of the library and compiler code that I have written is dedicated just to this.

The applications demonstrated in Chapter 4. should have shown that this project is usable for various purposes, from plain old unbound Compute to real-time Graphics applications, and that it can keep up with other Compute/Graphics implementations as well, maybe even surpassing them. Furthermore, I hope that the Image-space Warping library or code within will prove useful, and building on this, future extensions are very well possible.

5.1. new Compute & Graphics APIs

In the future, this project could be extended to support Graphics APIs as well. For Metal, this would even be possible today, with most of the compiler infrastructure already in place and being able to generate Metal-compatible LLVM IR and metadata, the only additions required in the compiler would be support for additional annotations on input and output structures (e.g. vertex program output, fragment program input, framebuffer output), which is fairly straightforward, although further semantic checking would be necessary as well, and of course some special wiring to flag and recognize functions as vertex or fragment programs (similar to what `kernel` does for Compute). That would cover the shading language part, the host API is an entirely different matter. As Graphics APIs go, doing Graphics is far more complicated than doing Compute, not only requiring the execution of shaders and handling of buffers and images, but also management and correct setup of the fixed-function rendering pipeline (blending, depth/stencil testing, render-target setup, to name a few), which can get rather messy and isn't always obvious on how it actually maps to hardware, or is stuck with how hardware used to do things (like intermediate mode rendering or OpenGL vertex attribute arrays). Moreover, it will also have to show if a unified approach with other future Graphics APIs is possible at all, or if this can only be done insufficiently due to APIs simply being too distinct or causing too much overhead and loss of performance (a state-based "setup everything on-the-fly" API like OpenGL would not map well to a "setup everything once at program start" API like Metal, although the other way around might work better, but would entail overhead on OpenGL's side due to possibly unnecessary API calls). Vulkan [KG15f], the successor API to OpenGL, is currently being developed and will for the first time support a standardized binary format for shaders and compute programs alike, going by the name SPIR-V [KG15c]. While obviously similar in name to SPIR, the binary format is not related to it, but a Khronos-own development. However, and maybe more interestingly, a LLVM backend for SPIR-V is currently also in the works [KG15d], thus making it a genuine output format that isn't dependent on the structure of LLVM IR. Hence, if properly maintained and supported, this would alleviate one of the major problems that SPIR has today: compatibility with newer versions of LLVM without the need for IR conversion tools. And of course, with it being a proper LLVM target, this project can (and will) be extended to support SPIR-V as well, practically making it just another backend format that can be chosen. Concerning support of Vulkan's Graphics API itself, the same reasoning and problems go as for Metal, being possible in principle, but if a unified approach is feasible remains to be seen. Albeit that Vulkan's design goals seem to be closer to Metal's than they are to OpenGL's [KG15g], making this a bit more likely. OpenCL 2.x [OWG15a]

support on the other hand will be less of an issue, as it generally preserves OpenCL 1.x's API and primarily adds new features. Notable ones are the addition of dynamic parallelism, the ability to directly launch kernels inside a kernel running on a device without involving host communication (bringing it up to par with CUDA and second-generation Kepler/`sm_35` hardware), as well as shared virtual memory and pipes, making more complex host/device interaction and memory structures possible and efficient. Note that while SPIR-V will provide a joint Compute and Graphics binary format, there is no mention of a joint Compute/Graphics language from Khronos' side yet and it currently seems more likely that they will keep on using GLSL and OpenCL C/C++ separately. As already said for Metal, this project could be extended to support Graphics, thus providing such a joint language.

Additionally, with Vulkan and SPIR-V, this project could be extended to support other mobile platforms and hardware as well, which is impossible right now as they do not offer programmability via LLVM, but will do so with Vulkan/SPIR-V [Goo15].

Another interesting aspect where Compute and the ability to handle and write more complex code might come in handy is with indirect GPU rendering and general pipeline management. Most GPUs today can already perform rendering through GPU generated command buffers [KG12], but the possibilities to manage this still seem rather primitive. With latest compute devices already being capable of launching kernels themselves, the possibility to issue render calls on the GPU itself doesn't seem that far-fetched, but will probably require more hardware support. Furthermore, with other parts of a pipeline also happening on the GPU already (like physics [Cou15], audio processing [AGLP11], or even AI [BN11]), it seems quite prudent to put any remaining logic from the CPU to the GPU, thus relegating the CPU to creating fixed render state objects that the GPU can reference and use for draw calls, and uploading new data like 3D model data, image data or camera input. The facilities for this should largely already be in place (OpenCL SVM, CUDA unified memory - although either one not universally usable just yet), hence low-level and low-overhead CPU/GPU communication through memory seems possible.

5.2. new language and hardware functionality

Being based on Clang/LLVM has other benefits as well: new language standards and features are implemented quickly and with the next C++ standard on the horizon (speculatively named C++17), it is to be expected that most or all functionality will already be implemented by the time C++17 is finally ratified, as has been the case for C++14 [Sta13]. Of notability here [Sta15] are concepts

and fold expressions on the language side (useful and usable on Compute), and concurrency/parallelism extensions on the library side, which might even allow Host-Compute to be implemented in a standard-compliant and equally performing way in the future. Some new functionality has already been implemented at this point [LLV15b]. If everyone involved can keep up (which seems likely at this point), it should thus be possible to rather quickly progress to having C++17 support in this project as well.

As briefly mentioned in the previous section, the addition of new hardware functionality like dynamic parallelism will be possible. Another feature that is already usable with CUDA today (and already made use of in this project as all CUDA image functionality is implemented using these), but not yet with OpenCL or Metal, are bindless images. These enable arbitrary access to images simply through the use of a 64-bit integer handle that can be retrieved through any means (e.g. loading through a buffer, having stored the handle in some structure) and therefore lift the limitation that all used images must be known when launching the kernel, since images had to be specified as kernel parameters. This has also been supported by OpenGL for a while now (ARB since 2014 [KG14a]), but none of this is exposed through any externally programmable means yet, but will be very interesting once there is support for them.

A. Gaussian blur compile-time coefficient computation

Compute languages like CUDA C++ or OpenCL C, or Graphics languages like GLSL require that constant values that are used for any kind of computation are either already present in a pre-computed state that was obtained through some external means before compilation ("magic values"), or compute them on the host at run-time and then supply these as run-time parameters to the kernel or shader. The former is to be preferred when performance matters, as such constant values are a direct part of the binary and very likely part of hardware instructions, allowing them to be used directly in a single instruction. The latter will always add additional instructions as run-time parameters have to be loaded first before they can be used in the actual computation (thus, slower) - it makes it however possible to compute these constant values via complex code at run-time and generally keep code more generic. Both of these use cases can possibly be served at compile-time as well when using C++, and C++14 especially. While it was already possible to compute some values at compile-time through template metaprogramming in C++98, this was always a rather clumsy way of achieving this and limited in several ways, including being very heavy on compilation time (instantiating templates is expensive). C++11 introduced a very limited form of `constexpr` functions (essentially limited to a single return statement), but C++14 extended `constexpr` quite far in that it is now possible to write proper C++ functions that run at compile-time (with some restrictions [ISO14, 5.19.2]). This makes it both a lot easier to write compile-time code (or even possible in the first place), but is also a lot lighter on compilation time. The Chapter 3.3.1. Basic Math Library was intentionally written to be used in that fashion.

As an example use case consider the computation of the Gaussian blur coefficients that are necessary for each pixel (or tap). As in Chapter 4.2.3. this is implemented as a separable filter and so only needs to compute a 1D array of coefficients that can be applied to both the vertical and horizontal pass. Likewise, this also uses the Pascal's triangle approximation. As an addition, this does not only compute the blur coefficients, but also finds the most effective row N in the triangle that should be used with a given tap count ("how many pixels are read") by checking whether pixel contributions on the outside are too small to affect the final result. The actual blur kernel width is thus "dynamic" as it depends on the tap count, but will give the most efficient result for a given amount of pixels that have to be sampled. Thus, a tap count of e.g. 21 results in an effective kernel width of 63 (31px radius), and the general input tap count to effective N mapping that is found is as follows: $3 \rightarrow 3$, $5 \rightarrow 5$, $7 \rightarrow 7$, $9 \rightarrow 11$, $11 \rightarrow 15$, $13 \rightarrow 21$, $15 \rightarrow 29$, $17 \rightarrow 39$, $19 \rightarrow 51$, $21 \rightarrow 63$. Note that tap count must always be odd-numbered

and that single-precision floating point math beyond $N = 63$ gets very wonky, so 21 is chosen as a cutoff value. Also note that a specific `TAP_COUNT` define has to be provided at compile-time in this example (it is however possible to instantiate for any tap count).

```
template <uint32_t tap_count> static constexpr uint32_t find_effective_n() {
    // minimal contribution a fully white pixel must have to affect the result
    // (ignoring the fact that 0.5 gets rounded up to 1 and that multiple outer
    // pixels combined can produce values > 1)
    constexpr const auto min_contribution = 1.0L / 255.0L;
    // start at desired tap count and go up by 2 "taps" if the row is unusable
    // (and no point going beyond 63 as described above)
    for(uint32_t count = tap_count; count < 64u; count += 2) {
        // 1 / 2^N for this row
        const long double sum_div = 1.0L /
            (long double)const_math::pow(2ull, (int)(count - 1));
        for(uint32_t i = 0u; i <= count; ++i) {
            const auto coeff = const_math::binomial(count - 1u, i);
            // is the coefficient large enough to produce a visible result?
            if((sum_div * (long double)coeff) > min_contribution) {
                // if so, check how many usable values this row has now
                // (should be >= desired tap count)
                if((count - i * 2) < tap_count) {
                    break;
                }
                return count;
            }
        }
    }
    return 0;
}
```

Listing 21: find effective row N for a given tap count at compile-time

```
// computes the blur coefficients for the specified tap count at compile-time
template <uint32_t tap_count> static constexpr auto compute_coefficients() {
    // std::array is not constexpr capable -> need to use floor const_array
    const_array<float, tap_count> ret {};
    // compute binomial coefficients and divide them by 2^(effective N - 1)
    // this is basically computing a row in pascal's triangle, using all values
    // (or the middle part) as coefficients
    const auto effective_n = find_effective_n<tap_count>();
    const long double sum_div = 1.0L /
        (long double)const_math::pow(2ull, (int)(effective_n - 1));
    for(uint32_t i = 0u, k = (effective_n - tap_count) / 2u;
        i < tap_count; ++i, ++k) {
        // coefficient_i = (n choose k) / 2^n
        ret[i] = float(sum_div *
            (long double)const_math::binomial(effective_n - 1, k));
    }
    return ret;
}
```

Listing 22: compute Gaussian blur coefficients at compile-time

```
// actual blur computation, instantiated sep. for horizontal and vertical
template <uint32_t direction /* 0 == horizontal, 1 == vertical */>
static void image_blur(const_image_2d<float> in_img,
    image_2d<float4, true> out_img) {
    const int2 coord { global_id.xy };
    // instantiate coefficients array for this tap count
    // (note that this is the first use of TAP_COUNT and this function could
    // be extended to be instantiated using an additional tap count parameter)
    constexpr const auto coeffs = compute_coefficients<TAP_COUNT>();
    constexpr const int overlap = TAP_COUNT / 2;
    float4 color; // assuming we have RGBA data
    for(int i = -overlap; i <= overlap; ++i) {
        // this all boils down to FMAs in the end
        color += coeffs[overlap + i] * in_img.read(coord + int2 {
            // note that this is a compile-time selection -> single ADD instr.
            direction == 0 ? i : 0,
            direction == 0 ? 0 : i,
        });
    }
    out_img.write(coord, color);
}

kernel void image_blur_horizontal(const_image_2d<float> in_img,
    image_2d<float4, true> out_img) {
    image_blur<0>(in_img, out_img);
}

kernel void image_blur_vertical(const_image_2d<float> in_img,
    image_2d<float4, true> out_img) {
    image_blur<1>(in_img, out_img);
}
```

Listing 23: actual blur computation and kernels

B. Source Code Release

All library and toolchain source code is released as open source and can be found on GitHub at the following URLs (this includes current development code and released binary packages):

- libfloor and toolchain patches: <https://github.com/a2flo/floor>, with automated toolchain builds at <https://libfloor.org/builds/toolchain>
- libwarp: <https://github.com/a2flo/libwarp>
- libfloor examples: https://github.com/a2flo/floor_examples

All libfloor, libwarp and example source code is released under GPLv2 license. All modifications to Clang, LLVM and libc++ are released under the same "UIUC" BSD-Style license as these projects [LLV15i], with libc++ modifications also dual-licensed under MIT license.

References

- [AGLP11] Lakulish Antani, Nico Galoppo, Adam Lake, and Arnon Peleg. Next-Gen Sound Rendering with OpenCL. https://www.khronos.org/assets/uploads/developers/library/2011-siggraph-openc1-bof/OpenCL-BOF-Intel-Sound-Rendering_SIGGRAPH-Aug11.pdf, 2011.
- [AMD12] AMD. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012.
- [AMD15a] AMD. CodeXL – Powerful Debugging, Profiling & Analysis. <http://developer.amd.com/tools-and-sdks/openc1-zone/codex1/>, 2015.
- [AMD15b] AMD and LLVM. AMDGPU LLVM target. http://llvm.org/viewvc/llvm-project/llvm/branches/release_37/lib/Target/AMDGPU/, 2015.
- [App15a] Apple. Metal Feature Set Tables. https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/MetalFeatureSetTables/MetalFeatureSetTables.html#//apple_ref/doc/uid/TP40014221-CH13-SW1, 2015.
- [App15b] Apple. Metal for Developers. <https://developer.apple.com/metal/>, 2015.
- [App15c] Apple. Metal Shading Language Guide. https://developer.apple.com/library/prerelease/ios/documentation/Metal/Reference/MetalShadingLanguageGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014364, 2015.
- [App15d] Apple. `metal_texture` and `GPUCompiler.framework`. part of Xcode and OS X, 2015.
- [Ars15] Ars Technica. OS X 10.11 El Capitan: The Ars Technica Review. <http://arstechnica.com/apple/2015/09/os-x-10-11-el-capitan-the-ars-technica-review/7/#h1>, 2015.
- [Bei15] Beignet. Beignet. <http://www.freedesktop.org/wiki/Software/Beignet/>, 2015.
- [BN11] Avi Bleiweiss and NVIDIA. GPU Acceleration for Board Games. <https://developer.nvidia.com/gpu-ai-board-games>, 2011.

- [Cou15] Erwin Coumans. experimental Bullet 3 GPU rigid body pipeline. <https://github.com/bulletphysics/bullet3>, 2015.
- [DER⁺10] Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *Computer Graphics Forum (Proceedings Eurographics 2010, Norrköpping, Sweden)*, 29(2):713–722, 2010. <http://resources.mpi-inf.mpg.de/3DTemporalUpsampling/3DTemporalUpsampling.pdf>.
- [GCC15] GCC. GCC, the GNU Compiler Collection. <https://gcc.gnu.org>, 2015.
- [GDB15] GDB. GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>, 2015.
- [GH13a] Benedict R. Gaster and Lee Howes. Formalizing Address Spaces with Application to Cuda, OpenCL, and Beyond. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6*, pages 32–41, New York, NY, USA, 2013. ACM. http://www.leehowes.com/files/gaster-2013-formalizing_address_spaces.pdf.
- [GH13b] Benedict R. Gaster and Lee Howes. OpenCL C++. In *Proceedings of the Sixth Workshop on General Purpose Processing Using GPUs (GPGPU-6)*. ACM, 2013. <http://www.leehowes.com/files/gaster-2013-openclc++.pdf>.
- [Goo15] Google and Shannon Woods. Android Developers Blog: Low-overhead rendering with Vulkan. <http://android-developers.blogspot.de/2015/08/low-overhead-rendering-with-vulkan.html>, 2015.
- [Hol12] Justin Holewinski. [LLVMdev] [PATCH][RFC] NVPTX Backend. <http://lists.llvm.org/pipermail/llvm-dev/2012-April/049220.html>, 2012.
- [Int15a] Intel. Intel SDK for OpenCL Applications. <https://software.intel.com/en-us/intel-opencl>, 2015.
- [Int15b] Intel. Intel® VTune™ Amplifier 2016. <https://software.intel.com/en-us/intel-vtune-amplifier-xe>, 2015.
- [Int15c] Intel. The Compute Architecture of Intel® Processor Graphics Gen9. <https://software.intel.com/en-us/file/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0pdf>, 2015.

- [IOG04] The IEEE and The Open Group. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition: `ucontext.h`. <http://pubs.opengroup.org/onlinepubs/009695399/basedefs/ucontext.h.html>, 2004.
- [ISO14] ISO/IEC JTC1/SC22/WG21. Working Draft, Standard for Programming Language C++. ISO/IEC, March 2014. adopted C++14 DIS: <https://github.com/cplusplus/draft/blob/b7b8ed08ba4c111ad03e13e8524a1b746cb74ec6/papers/N3936.pdf?raw=true>.
- [KG12] The Khronos Group. `GL_ARB_draw_indirect`. https://www.opengl.org/registry/specs/ARB/draw_indirect.txt, 2012.
- [KG14a] The Khronos Group. `GL_ARB_bindless_texture`. https://www.opengl.org/registry/specs/ARB/bindless_texture.txt, 2014.
- [KG14b] The Khronos Group. The SPIR Specification, Standard Portable Intermediate Representation, version 1.2. https://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf, January 2014.
- [KG15a] The Khronos Group. SPIR on GitHub. <https://github.com/KhronosGroup/SPIR>, 2015.
- [KG15b] The Khronos Group. SPIR-Tools on GitHub. <https://github.com/KhronosGroup/SPIR-Tools>, 2015.
- [KG15c] The Khronos Group. SPIR-V Specification. <https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.html>, 2015.
- [KG15d] The Khronos Group. SPIRV-LLVM on GitHub. <https://github.com/KhronosGroup/SPIRV-LLVM>, 2015.
- [KG15e] The Khronos Group. The OpenGL Graphics System: A Specification (Version 4.5 (Compatibility Profile) - May 28, 2015). <https://www.opengl.org/registry/doc/glspec45.compatibility.pdf>, 2015.
- [KG15f] The Khronos Group. Vulkan. <https://www.khronos.org/vulkan>, 2015.
- [KG15g] The Khronos Group. Vulkan Overview. <https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf>, 2015.
- [LLV10] LLVM. Clang Successfully Self-Hosts! <http://blog.llvm.org/2010/02/clang-successfully-self-hosts.html>, 2010.

- [LLV11] LLVM. LLVM 2.9 PTX Backend. http://llvm.org/viewvc/llvm-project/llvm/branches/release_29/lib/Target/PTX/, 2011.
- [LLV13] LLVM and Richard Smith. Clang support for C++11 and beyond. <http://blog.llvm.org/2013/04/clang-support-for-c11-and-beyond.html>, 2013.
- [LLV14a] LLVM. FTL: WebKit’s LLVM based JIT. <http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html>, 2014.
- [LLV14b] LLVM. NVPTXFavorNonGenericAddrSpaces.cpp. http://llvm.org/viewvc/llvm-project/llvm/branches/release_35/lib/Target/NVPTX/NVPTXFavorNonGenericAddrSpaces.cpp?view=markup, August 2014.
- [LLV15a] LLVM. . <http://clang.llvm.org/docs/UsersManual.html#controlling-code-generation>, 2015.
- [LLV15b] LLVM. C++1z implementation status. http://clang.llvm.org/cxx_status.html#cxx17, 2015.
- [LLV15c] LLVM. clang: a C language family frontend for LLVM. <http://clang.llvm.org>, 2015.
- [LLV15d] LLVM. "compiler-rt" runtime libraries. <http://compiler-rt.llvm.org>, 2015.
- [LLV15e] LLVM. Compiling CUDA C/C++ with LLVM. <http://llvm.org/docs/CompileCudaWithLLVM.html>, 2015.
- [LLV15f] LLVM. DragonEgg. <http://dragonegg.llvm.org>, 2015.
- [LLV15g] LLVM. Introduction to the Clang AST. <http://clang.llvm.org/docs/IntroductionToTheClangAST.html>, 2015.
- [LLV15h] LLVM. "libc++" C++ Standard Library. <http://libcxx.llvm.org>, 2015.
- [LLV15i] LLVM. LLVM Developer Policy: Copyright, License, and Patents. <http://llvm.org/docs/DeveloperPolicy.html#license>, 2015.
- [LLV15j] LLVM. LLVM Language Reference Manual. <http://llvm.org/docs/LangRef.html>, 2015.
- [LLV15k] LLVM. LLVM’s Analysis and Transform Passes. <http://llvm.org/docs/Passes.html>, 2015.
- [LLV15l] LLVM. LLVM’s Analysis and Transform Passes: Deduce

- function attributes. <http://llvm.org/docs/Passes.html#functionattrs-deduce-function-attributes>, 2015.
- [LLV15m] LLVM. MSVC compatibility. <http://clang.llvm.org/docs/MSVCCompatibility.html>, 2015.
- [LLV15n] LLVM. The LLDB Debugger. <http://lldb.llvm.org>, 2015.
- [LLV15o] LLVM. The LLVM Compiler Infrastructure. <http://llvm.org>, 2015.
- [LLV15p] LLVM. Writing an LLVM Pass. <http://llvm.org/docs/WritingAnLLVMPass.html>, 2015.
- [LP14] Michael Larabel and Phoronix. LLVM & Clang Had A Killer 2014 With Lots Of Improvements. http://www.phoronix.com/scan.php?page=news_item&px=MTg30TA, 2014.
- [MHJM13] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6. <http://www.x86-64.org/documentation/abi.pdf>, 2013.
- [Mic15a] Microsoft. Fibers (Windows). [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx), 2015.
- [Mic15b] Microsoft. microsoft-pdb: Information from Microsoft about the PDB format. <https://github.com/Microsoft/microsoft-pdb>, 2015.
- [Mic15c] Microsoft and Dave Bartolomeo. Clang with Microsoft CodeGen in VS 2015 Update 1. <http://blogs.msdn.com/b/vcblog/archive/2015/12/04/introducing-clang-with-microsoft-codegen-in-vs-2015-update-1.aspx>, 2015.
- [Mic15d] Microsoft and Jim Springfield. Rejuvenating the Microsoft C/C++ Compiler. <http://blogs.msdn.com/b/vcblog/archive/2015/09/25/rejuvenating-the-microsoft-c-c-compiler.aspx>, 2015.
- [NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch31.html, 2007.
- [NSL⁺07] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In Graphics Hardware, August 2007. http://gfx.cs.princeton.edu/pubs/Nehab_2007_ARS/NehEtA107.pdf.

- [NVI07] NVIDIA and Simon Green. CUDA 1.0 Released. <https://devtalk.nvidia.com/default/topic/373147/announcements/cuda-1-0-released/>, 2007.
- [NVI15a] NVIDIA. CUDA 7 Release Candidate Feature Overview: C++11, New Libraries, and More. <http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/>, January 2015.
- [NVI15b] NVIDIA. CUDA C Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2015.
- [NVI15c] NVIDIA. CUDA Toolkit Documentation v7.5. <http://docs.nvidia.com/cuda/index.html>, 2015.
- [NVI15d] NVIDIA. Nsight Eclipse Edition. <https://developer.nvidia.com/nsight-eclipse-edition>, 2015.
- [NVI15e] NVIDIA. Parallel Thread Execution ISA Version 4.3. <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html>, 2015.
- [NVI15f] NVIDIA. PTX ISA: Surface Instructions: sust. <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#surface-instructions-sust>, 2015.
- [NVI15g] NVIDIA. PTX ISA: Texture Instructions: tex. <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#texture-instructions-tex>, 2015.
- [OWG12] Khronos OpenCL Working Group. The OpenCL Specification, Version: 1.2, Document Revision: 19. <https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>, November 2012.
- [OWG15a] Khronos OpenCL Working Group. The OpenCL Specification, Version: 2.0, Document Revision: 29. <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>, July 2015.
- [OWG15b] Khronos OpenCL Working Group. (provisional) the OpenCL C++ Specification, Version: 1.0, Document Revision: 08. <https://www.khronos.org/registry/cl/specs/opencl-2.1-openc++-pdf>, March 2015.
- [OWG15c] Khronos OpenCL Working Group. The OpenCL Extension Specification, Version: 1.2, Document Revision: 22. <https://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf>, 2015.

- [poc15] pocl. Portable computing language. <http://pocl.sourceforge.net>, 2015.
- [Pow14] PowerVR. PowerVR Graphics Processors. <https://imgtec.com/?do-download=4381>, 2014.
- [Rob13] Paul T. Robinson. Developer Toolchain for PS4. <http://llvm.org/devmtg/2013-11/slides/Robinson-PS4Toolchain.pdf>, 2013.
- [Rá10] Daniel Rákos. Efficient Gaussian blur with linear sampling. <http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>, 2010.
- [Sta13] Standard C++ Foundation. Clang is (draft) C++14 feature-complete! <https://isocpp.org/blog/2013/11/clang-is-draft-c14-feature-complete>, 2013.
- [Sta15] Standard C++ Foundation. Current Status : Standard C++. <https://isocpp.org/std/status>, 2015.
- [Wen06] Carsten Wenzel. Real-time Atmospheric Effects in Games. http://developer.amd.com/wordpress/media/2012/10/Wenzel-Real-time_Atmospheric_Effects_in_Games.pdf, 2006.
- [Wik15a] Wikipedia. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree, 2015.
- [Wik15b] Wikipedia. atan2. <https://en.wikipedia.org/wiki/Atan2>, 2015.
- [Wik15c] Wikipedia. Clang: Status history. https://en.wikipedia.org/wiki/Clang#Status_history, 2015.
- [Wik15d] Wikipedia. Fiber (computer science). [https://en.wikipedia.org/wiki/Fiber_\(computer_science\)](https://en.wikipedia.org/wiki/Fiber_(computer_science)), 2015.
- [Wik15e] Wikipedia. Gaussian blur. https://en.wikipedia.org/wiki/Gaussian_blur, 2015.
- [Wik15f] Wikipedia. Half-precision floating-point format. https://en.wikipedia.org/wiki/Half-precision_floating-point_format, 2015.
- [Wik15g] Wikipedia. List of trigonometric identities. https://en.wikipedia.org/wiki/List_of_trigonometric_identities, 2015.
- [Wik15h] Wikipedia. Lists of NVIDIA, AMD and Intel GPU hardware. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units, https://en.wikipedia.org/wiki/List_of_

- AMD_graphics_processing_units, https://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units, 2015.
- [Wik15i] Wikipedia. LLVM. <https://en.wikipedia.org/wiki/LLVM>, 2015.
- [Wik15j] Wikipedia. Mac OS X Snow Leopard: OpenCL. https://en.wikipedia.org/wiki/Mac_OS_X_Snow_Leopard#OpenCL, 2015.
- [YTS⁺11] Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. Image-based bidirectional scene reprojection. In Proceedings of the 2011 SIGGRAPH Asia Conference, SA '11, pages 150:1–150:10, New York, NY, USA, 2011. ACM.
- [Í 05] Íñigo Quílez. float, small and random. <http://iquilezles.org/www/articles/sfrand/sfrand.htm>, 2005.