Openhipify: An OpenCL to HIP transpiler

Ben Mudd

Bachelor of Science in Computer Science
The University of Bath
2023-2024

Openhipify: An OpenCL to HIP transpiler

Submitted by: Ben Mudd

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

To harness the unique performance of the Graphics Processing Unit (GPU) a large variety of separate interfaces for device communication and execution have been developed. This has resulted in projects requiring separate programming languages and frameworks to target different GPU architectures and niches, exponentially increasing development time, cost and maintenance effort. Tools called transpilers attempt to translate one programming language into another, typically between two high-level languages. A variety of transpilers exist in the GPU programming space to help navigate the schism between frameworks, but the graph of interoperability has many missing edges. We propose a transpiler to add an edge to this graph between OpenCL and HIP (Heterogeneous-Compute Interface for Portability). We aim to create a complete compilation pipeline from a typical OpenCL program to conformant, isomorphic HIP. We test our tool on a constructed testing repository of OpenCL programs ranging in size, complexity and output. Testing metrics include the quality and correctness of code produced and the runtime performance of the transpiled HIP binary, all achieving favourable results. We conclude by discussing the development process and future steps to be taken, along with providing links to all of the software developed during this report.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Dr Russell Bradford for guiding me through this process. I would also like to thank Wyn Price, Jake Davies and Adam El Kholy for providing invaluable feedback and keeping morale high through the year.

Chapter 1

Introduction

As the size of transistors decreases to the atomic level, there appears to be a fundamental performance limit for silicon-based CPUs. Alternative methods to increase program execution speeds have been researched and interest in parallel architectures has arisen. Graphics cards are the current most popular form for this, appearing in 8 of the top 10 supercomputers, and over 150 of the top 500 [37]. Originally designed for graphics processing, the increased parallelism provided proves to be useful in a wide variety of unrelated fields, for example signal processing [6]. The term GPGPU (General Purpose Graphics Processing Unit) is used when GPUs are used for non-graphical purposes.

Programming for GPGPUs was initially conducted with existing rendering APIs (Application Programming Interface), for example OpenGL, which was used in one of the first examples of GPGPU computing [22]. This is not what these frameworks were designed for, resulting in 'hacky' programming and creating a need for more general-purpose solutions. Nvidia released CUDA (Compute Unified Device Architecture) in 2007, which was the first example of such a language, moving away from supporting graphic-specific features into more general, lower-level concepts. In the following years, many more GPGPU APIs were created, each with a unique set of advantages and disadvantages.

Translation between different GPGPU frameworks therefore provides various upsides. By-hand translation requires developers to have extensive knowledge of the source and target and dedicate a significant time investment into the process. A wide variety of transpilers have been developed, for example HIPIFY (CUDA to HIP), and CU2CL (CUDA to OpenCL), however, this is not a solved issue and is a field with significant gaps and research opportunities.

Chapter 2

Literature Review

2.1 Languages

2.1.1 CUDA

CUDA, labeled by Nvidia as the 'world's first solution for general-computing on GPUs' [29], leverages the resources of the GPU through a heterogeneous computing model. This consists of host code run on the CPU and device code offloaded onto the GPU. Host code runs serially, and deals with general problem setup, for example initial memory allocation. Device code, launched from the host, runs in parallel (specifically a form of SPMD[1]) on the GPU. The device threads are organised into a hierarchy of grids and blocks, designed to take advantage of a typical Nvidia architecture, shown in Figure 2.1.

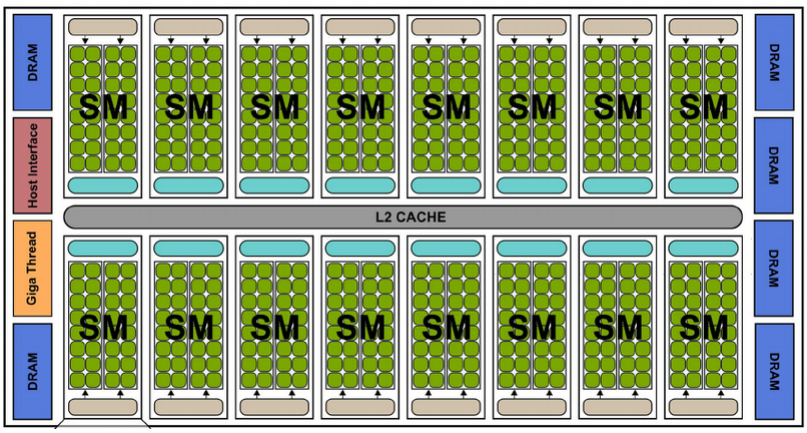

Figure 2.1: Typical Nvidia architecture [17]

CUDA is shipped by Nvidia alongside an extensive SDK (Software Development Kit), containing high-quality libraries, profilers, and more internally developed tools. In addition, Nvidia cards occupy a leading position in a variety of markets [32, 35], resulting in large incentives for developers to learn CUDA. This is troublesome as CUDA is only available on Nvidia's cards. Codebases would have to be rewritten to support AMD's cards (Nvidia's largest competitor), along with other acceleration devices. This is further accentuated by the fact that CUDA has considerably more optimised performance statistics compared to competing APIs on Nvidia cards, one example being OpenCL [11]. It could be speculated that Nvidia will not allow other APIs to compete with the performance metrics of CUDA, possibly due to the closed nature of the platform with themselves being the sole maintainer.

2.1.2 OpenCL

Released in 2009, OpenCL is a standard designed for general, parallel programming targeting a diverse range of accelerators. Initially developed at Apple, collaborators from a large range

[1]Single Program Multiple Data, a form of parallelism

of companies have since become involved, including Nvidia, AMD, IBM and more. This was formalised through the 'Heterogeneous Computing Initiative' from the Khronos group, which encourages participants in the parallel computing space to join and collaborate on mutually beneficial projects. The Khronos Group is an 'open, member-driven consortium that provides a space for the creation of the interoperability standards' [14]. Members include large software corporations discussed previously, but also academic institutions such as the University of Bristol [15].

This open standard has been designed with generality in mind, and takes ideas from CUDA (for example kernel launch layouts) and incorporates some of their own. The abstract idea of heterogenous computing has allowed numerous vendors to support OpenCL for their hardware [16]. Vendors can tailor their OpenCL implementation and extend it to target unique attributes of their devices. This can be seen with AMD's numerous OpenCL extensions, for example `cl_amd_device_attribute_query`, allowing AMD-specific information for the device to be returned. These extensions are necessary for OpenCL as graphics cards are not the only devices the standard caters to, it also includes CPUs, FPGAs and more. This introduces more required knowledge for OpenCL programming compared to other, non-general frameworks. If optimisation is desired from a multitude of separate architectures, knowledge of each of the separate OpenCL extensions must be known and incorporated.

The cost of supporting a wide variety of devices is reflected in the generality of the OpenCL programming model, resulting in verbose code with large sections of boilerplate. OpenCL is not able to make assumptions about the user's device, meaning the programmer is responsible for the bulk of device setup. For example, CUDA knows that the device can only be an Nvidia GPU, therefore is safe to precompile Nvidia GPU binaries. OpenCL does not have this luxury and is forced to compile device code at runtime as the device architecture cannot be inferred, and therefore must instead be queried at runtime to select the correct compilation settings. OpenCL provides numerous ways of doing this, however even for the smallest programs a general pipeline of host and device setup must be adhered to.

Du et al. [8] discuss the large overhead of runtime compilation for OpenCL kernels. While a well-documented experiment, some of the points raised are outdated. In 2020, an offline compilation workflow[2] was detailed by the Khronos Group [13], bringing forward techniques to compile OpenCL kernels to SPIR-V (Standard Portable Intermediate Representation), a popular accelerator language IR (Intermediate Representation). While an improvement from before, this IR is once again device agnostic, and will still need to be assembled at runtime to match the available architecture. This will be faster than a full OpenCL C kernel compilation but is still an overhead.

Sun [36] argues this runtime compilation leads to an 'opportunity to optimise OpenCL kernels for a specific compute device'. This is done in OpenCL via `autotuning`. This technique is discussed further in work from Petrovič et al. [33], where tuning parameters can be specified to optimise for the hardware. This requires training data from the hardware for optimisation, which either comes at a cost of runtime overhead or required pre-training on all targeted devices. Consider a video game developer requiring acceleration device(s) for their game. The majority of their user base will be using hardware from Nvidia, AMD or Intel. All device-facing code will still need to be compiled at runtime as OpenCL does not assume architecture. OpenCL competitors, for example HIP (discussed in section 2.1.3), will supply this information at compile time, reducing runtime overhead.

[2]Compilation of kernels at host compile time, not runtime

OpenCL kernels are written using OpenCL C, a subset of C99. Recently with the release of OpenCL 2.1 (2015), OpenCL C++ was released and is now a subset of C++17 with certain restrictions. For example, the `dynamic_cast` operator is not supported which is crucial for polymorphic C++ programming, along with a lack of standard C++ libraries. OpenCL C also contains restrictions, including a multitude of standard library exclusions, lack of recursion, and more. Some of these are small restrictions, however the limited library support is a large flaw, something which CUDA and other competitors do not suffer from. This stems from OpenCL simply not being as popular as CUDA, so there are fewer libraries developed. In addition, programming with separate host and device-side languages can become a cause for confusion, especially for new OpenCL programmers who may not have a complete understanding of the separate limitations.

The performance of an OpenCL program is dependent on the implementation of OpenCL by the vendor, resulting in a range of quality across architectures. Nvidia, as discussed previously, leaves much to be desired with their implementation; we speculate due to preferring to develop CUDA. After the release of the OpenCL 3.0 specification, a full year passed before it was fully supported on Nvidia hardware, while CUDA was enjoying quarterly releases. This leaves on Nvidia's platforms the choice for programmers on whether to port their existing OpenCL implementation (designed for interoperability) to CUDA to achieve the largest performance benefits. This splitting of a codebase defies the core values of OpenCL yet is something parallel developers must take into account.

2.1.3 HIP

HIP is the GPGPU programming model for the AMD ROCm software stack, released in 2016. HIP borrows features from both OpenCL and CUDA, alongside some of AMD's own. ROCm is AMD's software stack for heterogeneous GPU computing, containing a large variety of tools and frameworks, all of which are open-source and therefore available free of charge [4]. This allows for the general programming community to contribute towards the project which gives the possibility for a greater rate and lower cost of development compared to a closed-source alternative. Borrowed from OpenCL is the liberal approach to target devices, as while managed by AMD, their cards are not the only valid targets. HIP can run on AMD's cards, Nvidia's cards and more, which is a large advantage over CUDA. HIP chooses to implement this portability in a separate way from OpenCL, with different binaries being produced depending on the device architecture. On Nvidia's platforms, CUDA code is produced from a set of HIP wrapper functions which can then be compiled via nvcc (Nvidia's CUDA compiler) therefore benefitting from all its optimisations. For AMD's platforms a separate compiler, hcc, is used. This allows kernels to be compiled at compile time rather than at runtime, reducing this overhead that OpenCL has.

HIP has similar performance statistics on Nvidia's platforms when compared with CUDA as HIP produces CUDA code during its compilation pipeline. Tsai et al. [38] profile a range of algorithms with both CUDA and HIP code on Nvidia GPUs, finding '90% of the test cases show less than 10% performance difference'. This is consistent with other sources, however a tuned porting workflow is used which will be unique per problem, reducing confidence in certain results. The similarities to CUDA continue into the HIP syntax, which is designed to be almost identical to CUDA allowing for this wrapper system to be utilised. CUDA is the most popular GPGPU API, so parallel programmers tend to be most familiar with the CUDA programming design, which aids in the HIP learning process.

The growth of HIP in the GPGPU industry has resulted in AMD platforms having a greater set of compliant software. The prominent rise of deep-learning technology in the past decade has had many large frameworks lacking AMD support, and only recently with ROCm being properly adopted. An example of this is PyTorch, initially released in 2016, but only in 2018 including ROCm tools for compilation on AMD GPUs. Due to HIP's open-source code base, adding new libraries and tools is much easier for developers, and does not need to first be facilitated by AMD themself. A ROCm drawback however is a limit on platforms supported, both hardware and software. HIP only supports discrete GPUs, excluding integrated GPUs in base processors, something which OpenCL and Nvidia support. HIP is also primarily targeted at the Linux platform, with limited support for Windows, and none for macOS. Windows 10, 11, and Server are supported, however the Linux platform contains better and more extensive support, for example GPU virtualisation.

As ROCm is much newer than CUDA and OpenCL, less online discourse and documentation surrounds it. Getting started guides produced by AMD are limited, and community-driven guides and explanations are sparse compared with CUDA.This may be another reason why HIP is designed to be as similar to CUDA as possible; users can instead use CUDA documentation as the overlap between the two models is large. ROCm ships with common GPGPU libraries, including rocFFT (implementation of cuFFT, CUDAs version), and more. AMD have recognised developers will already be familiar with these libraries and have attempted to make switching frameworks as easy as possible with one-to-one API mapping guides [2].

2.2 Transpilers

As different GPGPU APIs have unique sets of pros and cons, the usefulness of having tools to translate between them is paramount. The time spent having to manually port a project to a different framework will be reduced, along with lowering the barrier to entry. This also reduces the cost of having to maintain multiple, separate codebases, as instead a centralised codebase can be developed and then transpiled with minimal programmer interaction. We showcase the current transpiler development scene in Figure 2.2.

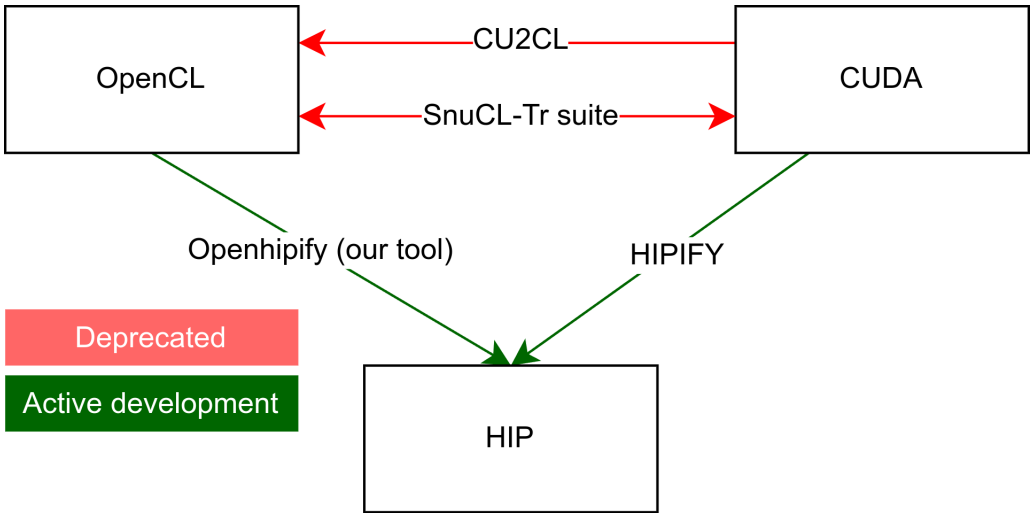

Figure 2.2: GPU Transpiler scene

2.2.1 HIPIFY

AMD ships the HIPIFY suite [3] which provides tools to translate CUDA code into corresponding HIP code. This does not require large-scale source transformations due to HIP-CUDA syntax similarities, therefore this can mostly be done via find and replace actions on CUDA semantics with corresponding HIP ones. However, certain scenarios require more context-driven translations, therefore HIPIFY contains two separate translation tools, hipify-clang, and hipify-perl, each handling translation in a separate way.

hipify-perl

Hipify-perl is an autogenerated Perl script built from hipify-clang, which relies on regular expressions to parse and translate CUDA to HIP. This has the bonus that the CUDA, HIP, and other third-party SDKs do not need to be installed on the system used for translation, only Perl. This is more useful for containerised applications as produced images will be less bloated. Regular expression parsing comes with a large number of issues when applied to C++, a non-context-free grammar. Consider the following code snippet:

```
1 foo * bar;
```

This is conformant C++ (and by extension CUDA) code, with two context-driven parses. This is either a declaration of a variable `bar` with the type `foo*` or is a multiplication of `foo` and `bar`, potentially overloaded. Regular expression matching cannot deduce which is the correct parse, therefore is unable to be confident in a correct transformation. Furthermore, regular expression matching via Perl is slow in comparison to other translation methods which is an additional drawback. This tool exists not to be a standalone translator, but more as an estimation tool to evaluate the general time needed to port an existing CUDA code base to HIP [36]. This results in the drawbacks of the tool being overlooked as it is not designed to be able to translate with complete correctness. For more complex programs, hipify-clang is used.

hipify-clang

Hipify-clang is AMD's more heavyweight solution for translating CUDA to HIP. This is a clang-based tool, meaning it utilises the LLVM framework [23] during transformation. Hipify-clang first preprocesses the code, handling macro expansion, header insertion and more, something that hipify-perl was not able to do. The code is then converted into an AST (abstract syntax tree), allowing for situations like the previous parsing issue to be handled correctly. Hipify-clang has access to the full program context during transformation, allowing for more complex and precise translations. The AST is traversed, and transformation matchers are applied, producing HIP code. One of the advantages of leveraging LLVM is that updates to CUDA will be automatically supported as the clang frontend from LLVM is used for AST construction. LLVM is a regularly updated framework, with companies including Google, Sony, Apple and the wider open-source community contributing. Any improvements to LLVM are therefore indirectly added to this tool.

2.2.2 CUDA to OpenCL translators

There are few CUDA to OpenCL translators in existence, the most widely known being CU2CL. Another example, SnuCL-Tr, contains both a CUDA to OpenCL translator and an OpenCL to CUDA translator. Both CU2CL and SnuCL-Tr have been deprecated since 2017 and 2015 respectively, one may argue due to the continued development of HIP giving a portability layer for CUDA to heterogeneous systems.

CU2CL

CU2CL, like HIPIFY, uses LLVM as a base for the creation of the tool. This is specified due to the ability to offload parsing, semantic analysis, and more to the compiler framework. Clang is used to generate an AST, which is then walked through with relevant sections rewritten. The approach by CU2CL is to find a CUDA snippet and rewrite this to a corresponding OpenCL snippet. Many CUDA API calls have a one-to-one correspondence with OpenCL calls, which allows for the majority of translation to take the form of simple replacements. Certain API calls, for example cudaGetDevice, do not have an OpenCL equivalent, and corresponding OpenCL boilerplate code is injected. This requires a more complex transformation to maintain program efficiency. For example, instead of duplicating the corresponding OpenCL boilerplate, a function may instead be added and tracked for future use.

An interesting point raised by CU2CL [12] discusses the differences between the translation of CUDA source code and the translation of the PTX IR ('a low-level parallel thread execution virtual machine and instruction set architecture' [31]) generated by nvcc. A source-to-source translation is argued to preserve program semantics and also allow for continued development in the newly translated source. A downside is brought up that the source code however must be available for this type of translation, and PTX translation can instead be done with a CUDA binary. We however believe this downside can be disregarded depending on the goal of the translator project. If the translator simply aims to run a program on a different platform, then PTX translation will work fine. If the translator aims to aid in porting from one framework to another then the source code can be assumed to be provided, which is the approach CU2CL takes.

SnuCL-Tr

SnuCL-Tr [20] takes a different approach to a CUDA to OpenCL translator. Similarly to CU2CL, CUDA kernels are source-to-source translated into corresponding OpenCL kernels. However, for host-side code, an API wrapper is used, similar to how HIP handles compilation on Nvidia platforms by wrapping CUDA API calls. The SnuCL-Tr wrapper contains CUDA function definitions with OpenCL implementations. The following shows the cudaMalloc implementation:

```
1  cudaError_t cudaMalloc(void **devPtr, size_t size) {
2  // Allocate memory on the device.
3  cl_mem mem = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL,
4    &cl_error);
5  if (cl_error != CL_SUCCESS)
6    fatal_CL(cl_error, __FILE__, __LINE__);
7
8  *devPtr = (void *)mem;
9  return (cudaError_t)cl_error;
10 }
```

Apart from a few exceptions, all CUDA functions are implemented via this method. For SnuCL, this achieves their goal of executing CUDA code via OpenCL, however calling this a 'translation' is using the term loosely. The host source code is untouched and only converted at link time. This is in effect no different than PTX translation from a developer's point of view; the source will still be syntactically valid CUDA code, not OpenCL.

SnuCL also ships an OpenCL to CUDA translator. This is the only OpenCL to CUDA translation tool that is currently released, but has been deprecated for over a decade. This takes a similar approach to translation as their CUDA to OpenCL tool: source-to-source translation of device code, and API wrappers for host-side code. The use of wrappers is argued here due to certain translation cases needing information only available at link time, an argument also brought forward by CU2CL. While correct, this results in the lack of a pure source-to-source OpenCL to CUDA translator. For a large portion of cases the information available for correct source-to-source translation is available before link time, so a tradeoff can be made to reduce the set of translateable programs in order to facilitate the existence of such a tool.

2.3 Conclusion

We have examined the issues surrounding the complex landscape of GPGPU programming. Tradeoffs between architectural target range and code verboseness and performance have been discussed in the context of OpenCL and CUDA, along with HIP offering a unique space in the market between them. Through this we have argued the importance of the transpiler as a tool for interoperability. We analysed the graph of existing transpilation targets, particularly for missing edges and identified a gap for our tool from OpenCL to HIP. We investigated existing transpilers and their strategies and attitudes towards translation, and weighed up the pros and cons for each. Through this we can start to plan the development and direction of our tool to be of paramount importance in this programming space.

Chapter 3

Methodology

While there is extensive documentation on source-to-source translators, there is a missing edge for an OpenCL to HIP pipeline (shown in Figure 2.2) which would provide many benefits, from simpler portability to Nvidia SDK usage for OpenCL and more. Our OpenCL to HIP translator, named Openhipify, will take the form of three separate stages, host translation, kernel translation, and post-translation. We have analysed the problem space and have constructed a set of primary and secondary requirements for our tool which has allowed us to narrow our development path.

3.1 Requirements analysis

3.1.1 Core requirement

A core requirement we define as a baseline goal we aim to achieve for this project. Our core requirement is to have the possibility for Openhipify to take an input OpenCL program and produce HIP code which is compilable and produces an isomorphic binary. Isomorphic here is used to denote that the two binaries behave identically, i.e. for the same input data, the same output is observed. Being able to translate all valid OpenCL programs as a core requirement would be unrealistic to the scope of our project, but it should be capable of producing conformant HIP code for basic OpenCL programs.

3.1.2 Primary requirements

Primary requirements we designate to be crucial for the completion of our core requirement, and remained relatively unchanged throughout the development process. Analysing the core requirement, to achieve a 'full' OpenCL translation we would need to:

- Translate kernel definitions
- Translate host-side programs
- Link kernel and host code together

These three necessary steps thus make up our primary requirements. Kernel definitions are programmed in a different language than host code, so we must handle the translation in a separate pass. We have ranked by importance a large quantity of OpenCL intrinsics and functions, i.e. for host translation, the function call `clEnqueueNDRangeKernel` is necessary to launch OpenCL kernels, therefore we must have a translation process for this call. However, `clCreateSubDevices`, while a useful call, is not a core part of an OpenCL program; The priority for handling translation here is therefore lower. As there will be separate stages for kernel and host translation, we have introduced another primary requirement that these stages should be callable independently of each other, i.e. a user can translate an OpenCL kernel without the corresponding host code, and vice versa. This would introduce added OpenCL and HIP interoperability as OpenCL kernels could be inserted into existing HIP projects, and developed HIP kernels could be launched from OpenCL host code. When used together the full pipeline should be made use of and the individual passes appropriately linked together.

3.1.3 Secondary requirements

Our secondary requirements, defined as requirements not directly related to our core requirement, have been iterated frequently during the development of Openhipify. For example, after the decision to use LLVM for this project (detailed in section 3.2.2), we set a secondary requirement to do extra syntactic analysis of the input OpenCL program to show errors that the standard OpenCL compilation workflow would not catch. This is due to LLVM containing extra tools that allow us to conduct such analysis that other transpilation strategies would not allow. Another secondary requirement surrounds the user experience of the tool. For example, if a translation is not able to take place, informative errors and warnings should be presented to the user to showcase how, why and where the error occurred. These warnings and errors should be presented professionally and be informative and concise. These requirements, while important, are not inherently crucial to our tool's core requirement, and are thus of lower importance.

3.2 Architecture Design

The project is split into two separate, modular components: kernel translation and host translation. When combined we aim for a 'full' translation, i.e. utilise a pseudo-linker step where unresolved kernel calls in host code can be linked together with their definition in kernel code. This requirement influences our choice for the architectural design of the two component passes. We now discuss different strategies for building these separate parts.

3.2.1 Regex Solutions

A regex translation (seen in hipify-perl 2.2.1) solution would allow for a relatively simple framework design as the bulk of translation would consist of a chain of find-replace operations. The intention of the code would not need to be considered, only a one-to-one relationship between OpenCL and HIP keywords and functions would be used. For a kernel pass this would be barely adequate. This is due to a large number of the OpenCL kernel functions having a direct HIP equivalent. Consider the following function call:

```
1 long id = get_global_id(0);
```

We can translate the following based on the CUDA convention to find the unique thread ID:

```
1 long id = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
```

This could be matched and replaced via the regex expression: 'get_global_id\((0)\)', where we could extract the argument (0, corresponding to the x dimension in the hip calls) from the first capture group for subsequent evaluation. While this specific case succeeds other cases will not, consider the following:

```
1 #define X_DIMENSION 0
2 long id = get_global_id(X_DIMENSION);
```

In this case we would match and extract the argument for the dimension to be 'X_DIMENSION' instead of 0 as the preprocessor has not been run so macro expansion has not taken place. We therefore cannot evaluate the expression, so our replacement fails. Consider another example:

```
long ids[3];
for(int i = 0; i < 3; i++) {
  ids[i] = get_global_id(i);
}
```

A find-replace solution completely falls apart here. `get_global_id` is returning different values in each for loop pass. The HIP equivalent for `get_global_id` does not take any arguments: the method for accessing different thread dimensions is separated into different function calls. For these two translation cases conventional regex methods are inadequate and would instead require a full code parse to confidently attempt a translation.

Regex translation for host code is inadequate as variable tracking is a necessity. Consider the following OpenCL pseudocode:

```
setKernelArgs(kernel, args);
launchOpenCLKernel(kernel);
setKernelArgs(kernel, args2);
launchOpenCLKernen(kernel);
```

The call to `launchOpenCLKernel` is the same, however the intended behavior is different. Arguments in OpenCL kernels are passed in through external function calls compared to being directly passed into the function launch as in HIP. The correct translation for this would be:

```
launchHIPKernel(kernel, args);
launchHIPKernel(kernel, args2);
```

`launchOpenCLKernel` cannot be matched and replaced without considering the setKernelArgs calls. Regex-based solutions do not allow this without considerable modification; we would have to implement a methodology that would mimic a parse-based solution. Ideally, we could track these variables to generate correct and consistent kernel launches.

3.2.2 Parsing Solutions

This is the solution we have chosen to utilise to give Openhipify a complete view of the input context. Our next choice surrounds how we choose to parse our code. A parser could be made from the ground up, complete with a lexer and an AST generator. Building a C frontend pipeline is far from trivial and is a problem where a variety of pre-built solutions already exist. We have decided to utilise the LLVM Project to offload parsing.

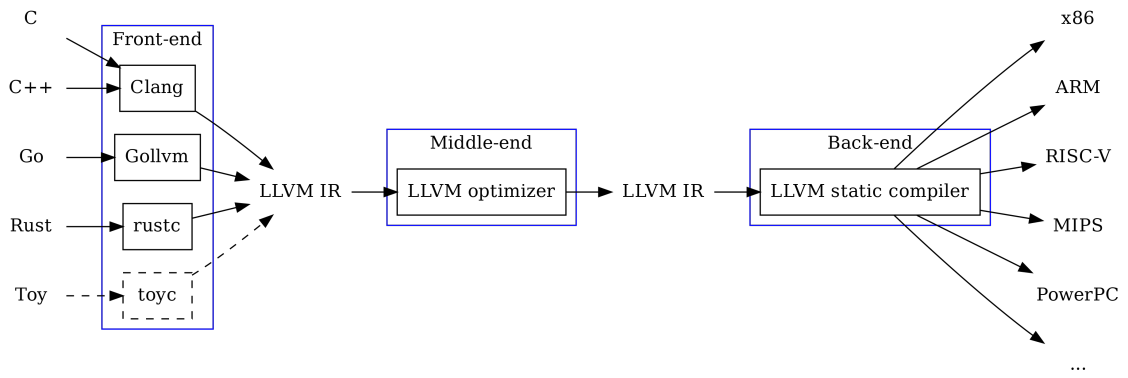

Figure 3.1: LLVM architecture [10]

Figure 3.1 shows an overview of a subsection of the LLVM toolkit, with a variety of compiler frontends and backends all utilising a shared optimisation pipeline. While this type of compiler design (separate compilation steps) is not unique to LLVM, the ability to take apart and isolate sections is. We could make use of the clang frontend and use it to parse OpenCL C code, then use this information during our translation stage. We then must decide on which parts of LLVM to use, and which to leave. We have a few options for LLVM outputs to use for our input OpenCL programs. We could use the clang frontend to lex, parse, and emit LLVM IR (an intermediate representation between C and assembly generated via LLVM), and then translate this IR to a HIP equivalent. This gives us a few advantages; we could run the LLVM opt tool to optimise this IR and generate less verbose, more optimised HIP code in comparison to the OpenCL input.

However, the downsides to this strategy for translation we believe vastly outweigh the upsides. Code comments are not propagated through to the IR, so would be lost, along with a large amount of the original code intention due to the inherent differences in syntax between LLVM IR and C. Consider the following kernel body:

```
int id = get_global_id(0);
// Make sure we stay inside the problem space
if (id < N)
  C[id] = A[id] + B[id];
```

A correct HIP translation for this would be to replace get_global_id with a HIP counterpart and leave the rest of the kernel untouched. However, compiling this to IR we see a vastly different program structure:

```
  %5 = tail call i64 @_Z13get_global_idj(i32 noundef 0) #2
  %6 = trunc i64 %5 to i32
  %7 = icmp slt i32 %6, %3
  br i1 %7, label %8, label %17

8:                                                ; preds = %4
  %9 = shl i64 %5, 32
  %10 = ashr exact i64 %9, 32
  %11 = getelementptr inbounds i32, i32* %0, i64 %10
  %12 = load i32, i32* %11, align 4, !tbaa !10
  %13 = getelementptr inbounds i32, i32* %1, i64 %10
  %14 = load i32, i32* %13, align 4, !tbaa !10
  %15 = add nsw i32 %14, %12
  %16 = getelementptr inbounds i32, i32* %2, i64 %10
  store i32 %15, i32* %16, align 4, !tbaa !10
  br label %17

17:                                               ; preds = %8, %4
  ret void
```

We have lost our higher-level program structure; control flow operators are lost in place of goto's, comments are discarded, et cetera. The IR is also in SSA (static single-assignment) form, meaning each variable is assigned just once before it is used and is not modified. This makes sense for compiler IR due to it aiding the optimisation pipeline, however produces an added constraint when raising to a higher level language as we require. Raising this IR representation into a C++ HIP equivalent (i.e. replacing get_global_id) would produce:

```
__global__ void vecAdd(int32_t *zero, int32_t *one, int32_t *two,
                       int32_t three) {
  int64_t five = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
  int32_t six = (int32_t) five;
  bool seven = six == three;
  if (seven) {
    goto eight;
  } else {
    goto seventeen;
  }

  // ... function continues ...
}
```

We lose our variable names, comments, and general structure. We have delved too greedily, and too deep into the LLVM pipeline. We instead choose to stop after AST generation. This allows us to observe the intention of the original program, track variables and more while keeping our source document to apply replacements too. Figure 3.2 showcases how we use the clang frontend to construct an AST of an OpenCL program and use this to form the basis of translation.

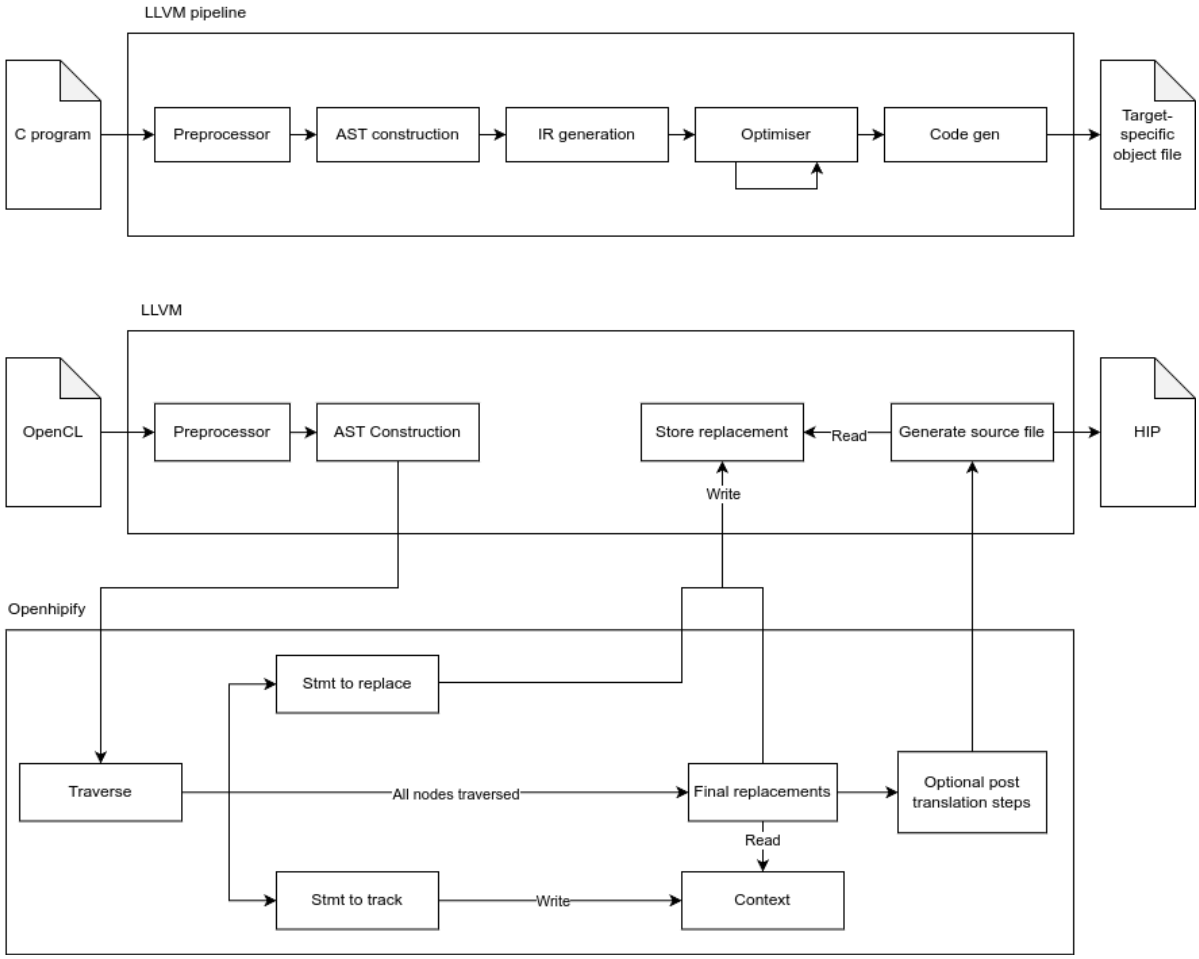

Figure 3.2: Openhipify architecture

The usefulness of this strategy for translation can be showcased using our previous example of `clSetKernelArg` calls in conjunction with kernel launching in Section 3.2.1. We can now track these statements while traversing the AST to see at what stage in the program they are executed. After the AST has been fully traversed, we can infer what arguments are present for each kernel launch function call, and therefore can be confident in generating an isomorphic HIP kernel launch.

Another positive of utilising the LLVM project is having access to the large volume of information that is present in the generated AST. We can track variables to their initial declaration which is useful to us when encountering multiple kernel definitions. We would need to determine which arguments are assigned to what kernel launch, which is all information present in the AST and surrounding LLVM data structures. The sheer volume of auxiliary tools provided is staggering, and it is no coincidence that most translator tools make use of it.

3.3 Kernel Translation

We now move onto the first stage in the translation pipeline for Openhipify: kernel file translation. This is done before host file translation so we can make use of gathered information from kernel AST traversal for HIP host code generation.

An AST encodes the structure of a program by dissecting it into its constituent parts, called nodes. In LLVM each node has a specific type and can contain extra information that may link it to other nodes in the tree. For example, a `DeclRefExpr` node signifies a 'reference to a declared variable, function, enum, etc.'[1], and will contain a pointer to a `ValueDecl` node representing the initial declaration of that value. We can use this to track uses of specific OpenCL intrinsics and replace not just the definition of them but all of their subsequent uses.

Consider the following OpenCL kernel function with its AST:

```
__kernel void vecAdd(__global int *A, __global int *B, __global int *C,
  const int N) {
int id = get_global_id(0);

if (id < N)
  C[id] = A[id] + B[id];
}
```

```
FunctionDecl 0x13e92b8 <test.cl:1:1, line:6:1> line:1:15 vecAdd 'void (__global int *__private, __global int ↩
    *__private, __global int *__private, const __private int)'
  |-ParmVarDecl 0x13e8fb0 <col:22, col:36> col:36 used A '__global int *__private'
  |-ParmVarDecl 0x13e9058 <col:39, col:53> col:53 used B '__global int *__private'
  |-ParmVarDecl 0x13e90d8 <col:56, col:70> col:70 used C '__global int *__private'
  |-ParmVarDecl 0x13e9158 <col:73, col:83> col:83 used N 'const __private int'
  |-CompoundStmt 0x1400170 <col:86, line:6:1>
  | |-DeclStmt 0x13e96e8 <line:2:3, col:33>
  | | `-VarDecl 0x13e93e8 <col:3, col:32> col:7 used VARNAME '__private int' cinit
  | |   `-ImplicitCastExpr 0x13e96d0 <col:17, col:32> 'int' <IntegralCast>
  | |     `-CallExpr 0x13e9690 <col:17, col:32> 'unsigned long'
  | |       |-ImplicitCastExpr 0x13e9678 <col:17> 'unsigned long (*)(unsigned int)' <FunctionToPointerDecay>
  | |       | `-DeclRefExpr 0x13e9610 <col:17> 'unsigned long (unsigned int)' Function 0x13e9480 'get_global_id' ↩
    'unsigned long (unsigned int)'
  | |       `-ImplicitCastExpr 0x13e96b8 <col:31> 'unsigned int' <IntegralCast>
  | |         `-IntegerLiteral 0x13e9630 <col:31> 'int' 0
  | `-IfStmt 0x1400150 <line:4:3, line:5:40>
  |   |-BinaryOperator 0x13e9770 <line:4:6, col:16> 'int' '<'
  // AST continues
  |
  `-OpenCLKernelAttr 0x13e9378 <line:1:1>
```

This is a textural representation of the AST generated by clang showing the nodes connected to the `FunctionDecl` of `vecAdd`. We can see the main `FunctionDecl` node has an `OpenCLKernelAttr` child. When searching through a kernel file's AST, this would be something we would match on to identify OpenCL kernel functions, and then run our function-specific pass on its children. Functions in OpenCL kernel files that do not have this attribute can be deduced to be device-only functions.

Device-only functions are only callable by kernel functions, being implemented in the OpenCL compiler via a guaranteed inline. As these are generated inside specific kernel files there is no ambiguity as to the type of function this is, i.e. the compiler can correctly assume this is not callable from host-side code. This is not the case in HIP, all functions are contained within standard .cpp files. HIP therefore incorporates syntactic support for this type of function to remove ambiguity as to whether non-kernel functions should be callable device-side or not. HIP contains the function attribute `__device__` which prepends a function definition which we add during this stage for function declarations lacking an `OpenCLKernelAttr` child.

When replacing inbuilt OpenCL C functions (e.g. `get_global_id`), there are two different methods that we use. The first method is a simple, one-to-one replacement with a HIP equivalent, which is similar to how a regex replacement would function. Consider:

```
long id = get_global_id(0);
```

```
long id = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
```

This is isomorphic to the regex method for translation. We implement a separate strategy when presented with unevaluatable arguments. Consider the previous kernel example:

```
long ids[3];
for(int i = 0; i < 3; i++) {
  ids[i] = get_global_id(i);
}
```

The variable `i` cannot be folded to a constant for evaluation, therefore we cannot deduce what dimension we need for the generated HIP replacement. One translation option would be to interpret the syntax of the for-loop, resulting in an insertion of 3 HIP function replacements, one for each dimension. This strategy of deducing the intention of the program falls apart quickly however. Consider the following:

```
__kernel void vecAdd(unsigned int n) {
  long id = get_global_id(n);
}
```

There is insufficient information to fold this call into a correct HIP equivalent. We instead choose to generate an auxiliary function to mimic `get_global_id` with HIP conformant code. We then call this auxiliary function in places where the function call cannot be statically deduced. The previous example is therefore translated to:

```
////////////////// Generated by Openhipify /////////////////
#include "hip/hip_runtime.h"

__device__ size_t __get_global_id(uint dim) {
  switch (dim) {
  case 0: {
    return hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
  } break;
  case 1: {
    return hipBlockDim_y * hipBlockIdx_y + hipThreadIdx_y;
  } break;
  case 2: {
    return hipBlockDim_z * hipBlockIdx_z + hipThreadIdx_z;
  } break;
  default: {
    return 0;
  } break;
  }
}
////////////////////////////////////////////////////////////

__global__ void vecAdd(unsigned int n) {
  long id = __get_global_id(n);
}
```

This strategy for function translation is similar to the strategy used by the SnuCL-Tr compiler 2.2.2, where the input CUDA program is unchanged, but the underlying function definitions wrap OpenCL implementations. As discussed previously we do not perceive this to be a 'translation', so we only use this method when necessary. This is the general strategy for the kernel translation pass, we can now start thinking of strategies to aid the host translation pass.

Host and kernel files should be translated separately as per our primary requirements, but there is a myriad of strategies we can implement in the kernel pass to aid and improve the quality of the generated host-side code if both are provided. One of these is the tracking and propagation of individual kernel definitions. During the kernel phase, we store information about kernel functions in a map, indexed by the kernel name. Kernels must be named uniquely in each file which is why we use this as our key. There could be kernel name overlap when multiple kernel files are present, but we deem this overlap case to be out of scope for the initial development of our tool, with plans to implement in the future. When we enter the host translation and post translation phases this map can be accessed to aid in translation, e.g. header file generation (discussed in section 3.5). This process of kernel tracking is detailed in Figure 3.3.

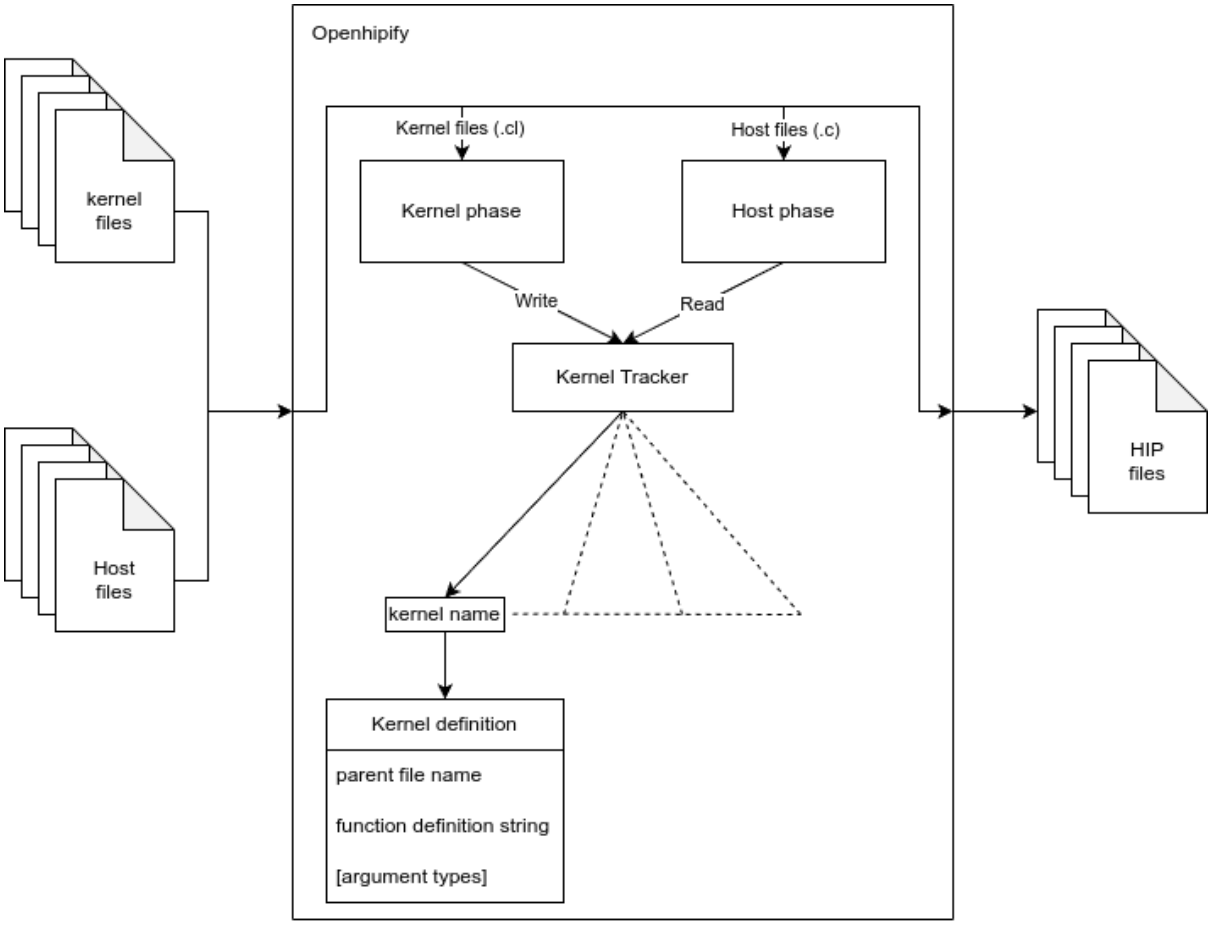

Figure 3.3: Kernel tracking

3.4 Host Translation

The host translation section for Openhipify operates on host side OpenCL code, i.e. the programs running on the CPU, responsible for GPU memory allocation, kernel scheduling, et cetera. This section of code is often more complex than kernel-side code due to it holding responsibility for a larger number of jobs. This results in translation for host-side code being more complex and involved.

Due to OpenCL's target independence, host-side code must be verbose to operate on such a large set of architectures. HIP does not have this drawback due to its smaller target pool. We therefore see less of the 'for free' translations that were present in the kernel phase of translation, with fewer direct one-to-one mappings for OpenCL functions and types to a HIP equivalent. There is not a complete absence of these however, one example being the translation of `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`.

These function calls are responsible for writing and reading data from the device, with a typical call taking the form:

```
1 clEnqueueWriteBuffer(queue, device_mem, CL_TRUE, 0, numBytes, ↩
    host_mem, 0, NULL, NULL);
```

This can be directly translated (under the assumption that no scheduling constraints are present) to the shorter HIP equivalent:

```
1 hipMemcpy(device_mem, host_mem, numBytes, hipMemcpyHostToDevice);
```

There are a few extra steps to this translation that we have skipped over in this example, mainly to do with the type of the variable `device_mem`, being a specific OpenCL type requiring its own translation strategy. This is discussed later on in this section.

Other OpenCL features will require some degree of semantic analysis instead of a one-to-one replacement, kernel launches being an example of this. Consider the previous example:

```
1 setKernelArgs(kernel, args);
2 launchOpenCLKernel(kernel);
3 setKernelArgs(kernel, args2);
4 launchOpenCLKernel(kernel);
```

`launchOpenCLKernel(kernel)` is syntactically the same function call, but requires translation to different HIP calls:

```
1 launchHIPKernel(kernel, args);
2 launchHIPKernel(kernel, args2);
```

As seen in Figure 3.2, we 'track' these relevant kernel calls. More specifically, we inspect `clCreateKernel`, `clSetKernelArg`, and `clEnqueueNDRangeKernel` function calls present in the OpenCL input program. For translation we require the whole context of the kernel-related functions which is why we traverse the whole AST before attempting replacement. After traversal all relevant kernel function calls will be tracked, and we produce the data structure shown in Figure 3.4.

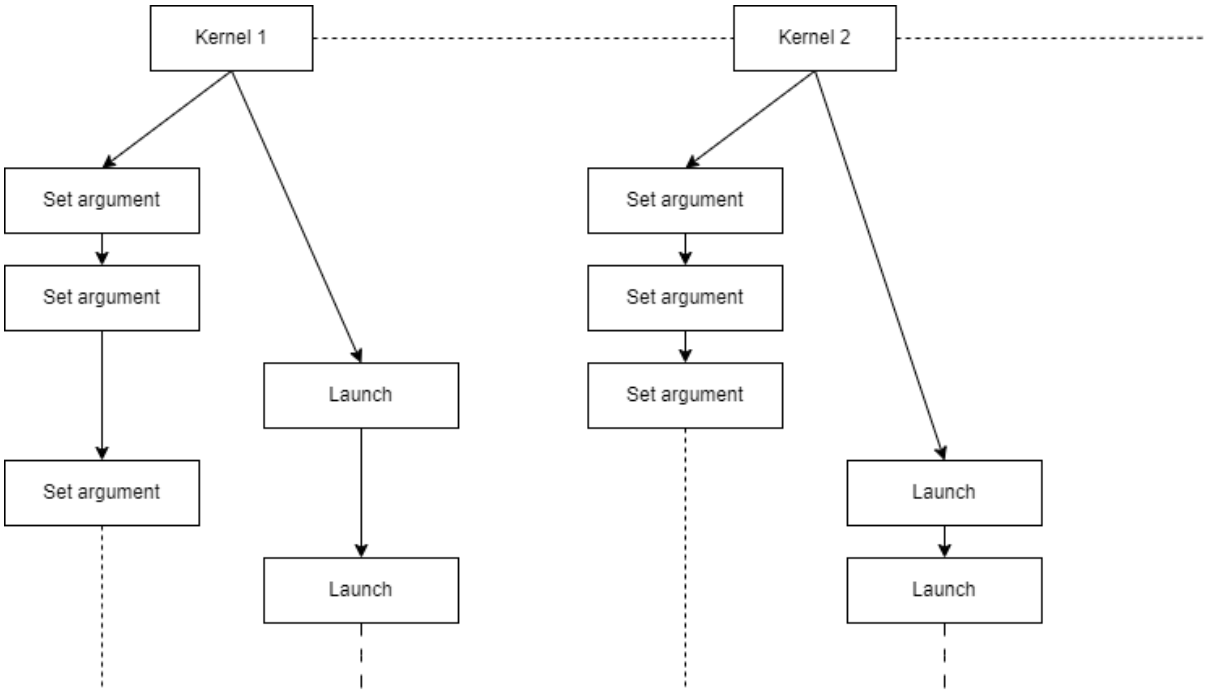

Figure 3.4: Kernel graph

Each `clSetKernelArg` and `clEnqueueNDRangeKernel` are grouped into their relevant `cl_kernel` group (generated via `clCreateKernel`), which is then used to build up an ordered list of kernel argument settings and launches. We must now determine a suitable method for sorting this list i.e. decide what `clSetKernelArg` calls come before what `clEnqueueNDRangeKernel` calls. We also must decide how we choose to define before in this context.

In the above example a suitable method would be to use the source line number, it is trivial to see what arguments should be where. This is unfortunately not suitable for all possible cases, consider the following pseudocode:

```
for arg in args{
  setKernelArgs(kernel, arg);
}

launchOpenCLKernel(kernel);
```

There is only one function call for `setKernelArgs` which is called several times. We could use some form of control flow analysis, but this comes with numerous issues. This is something we aim to implement in the future however, with an example problem and solution detailed in section 5.1. We instead choose to step back and take a larger look into our problem space. Through analysing a large number of OpenCL code repositories, we see rare uses of non-trivial kernel functions, OpenCL programmers seem to deal with kernel launches as standard function calls, i.e. setting arguments immediately before calling. We use this insight to direct our approach to translation and use the line number schema originally discussed. This covers most launch cases, but we recognise this is a drawback to our tool. The following shows an example of a translation:

```
cl_kernel kernel = clCreateKernel(program, "vecAdd", &err);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, ↩
   &localSize, 0, NULL, NULL);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_c);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, ↩
   &localSize, 0, NULL, NULL);
```

```
hipLaunchKernelGGL(vecAdd, dim3(globalSize), dim3(localSize), 0, 0, ↩
   (double *)d_a, (double *)d_b);
hipLaunchKernelGGL(vecAdd, dim3(globalSize), dim3(localSize), 0, 0, ↩
   (double *)d_a, (double *)d_c);
```

On occasions where we recognise this schema will produce incorrect launches, we generate a warning message shown in Figure 3.5 as per our secondary requirements, remove the function call and continue with translation.

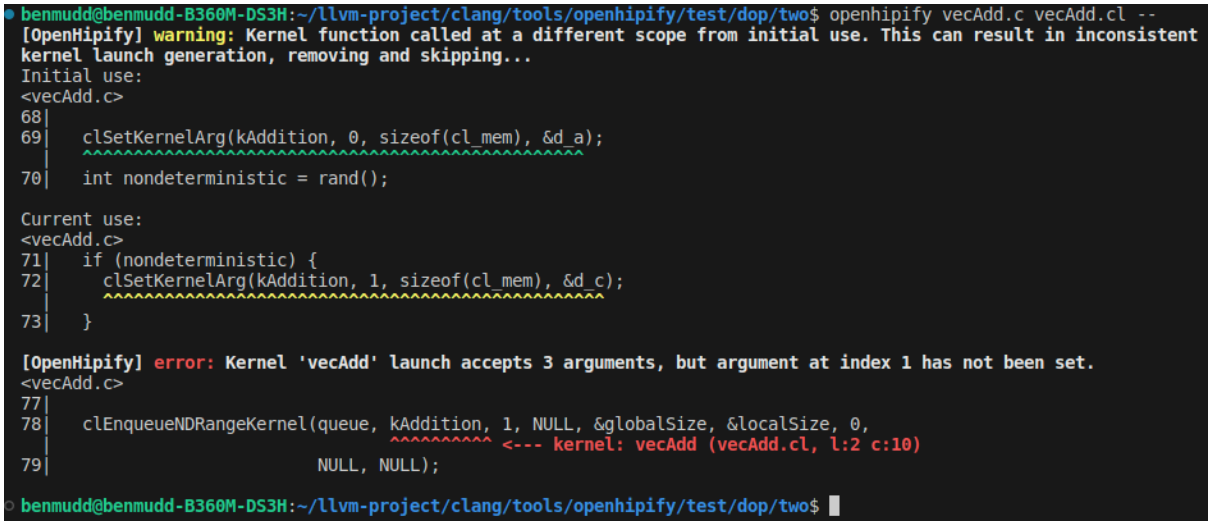

Figure 3.5: Openhipify kernel tracking error message

Sometimes this removal may cause other errors further on in the translation process (also seen in Figure 3.5), but this is intended. In this example, if argument 1 is set nondeterministically, certain runs will have it omitted which we believe justifies the use of a compiler error over a warning.

One of the major benefits we gain from our tracking of kernel-related functions is we can determine certain errors in OpenCL applications that the OpenCL compiler and runtime would not catch. An example of this is shown in Figure 3.5, demonstrating incorrect kernel argument setting. If an argument is not defined, we have a choice of throwing an error or implicitly passing in a void pointer. The latter is the choice that OpenCL takes, which is due to the number of kernel arguments only being known to the OpenCL runtime, not the compiler. We have the possibility of knowing this information at compile time which broadens our scope of possibilities when translating kernel-related functions:

Possibility one: Kernel declaration is known at compile time

This case occurs when the file containing the kernel declaration is passed into Openhipify for translation along with the host file containing the kernel launch. As seen in Figure 3.3, we parse kernel files first and track the kernel definitions. When arguments are set via `clSetKernelArg`, we check if the argument index is situated inside the kernel's parameter bounds. If not we can be certain that this argument will not be present in the kernel launch, where we then subsequently throw an error, shown in Figure 3.6. An argument could be made that this should be a warning, and the incorrect argument call be implicitly removed, but we have chosen instead to throw an error. We believe that it should not be present in the source OpenCL file in the first place, so the programmer should be explicitly made aware of this.

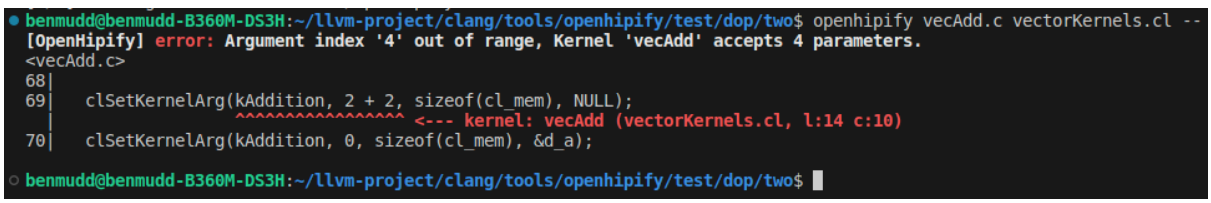

Figure 3.6: Openhipify kernel argument error (case 1)

Possibility two: Kernel declaration is not known at compile time

This occurs when the file containing the kernel declaration is not passed into openhipify during translation (which is possible as per our primary requirements). We are still able to infer erroneous `clSetKernelArg` and `clEnqueueNDRangeKernel` calls however. For example, if there are 3 `clSetKernelArg` calls for a kernel, and one of them references argument number 7, we deduce this to be an out-of-bounds error, and propagate this to the user (Figure 3.7).

```
benmudd@benmudd-B360M-DS3H:~/llvm-project/clang/tools/openhipify/test/dop/two$ openhipify vecAdd.c  --
[OpenHipify] error: Argument index '7' out of range, Kernel 'vecAdd' accepts at maximum 2 parameters.
<vecAdd.c>
70|   clSetKernelArg(kAddition, 1, sizeof(cl_mem), &d_b);
71|   clSetKernelArg(kAddition, 7, sizeof(cl_mem), &d_b);
   |                  ^^^^^^^^^^^^^ <--- kernel: vecAdd (untracked)
72|
```

Figure 3.7: Openhipify kernel argument error (case 2, example 1)

Additionally, if we set 5 arguments for a kernel, launch it, and then attempt to edit kernel argument number 8, we can once again infer this argument is out of range, and throw an error (Figure 3.8).

```
benmudd@benmudd-B360M-DS3H:~/llvm-project/clang/tools/openhipify/test/dop/two$ openhipify vecAdd.c  --
[OpenHipify] error: Argument index '8' out of range, Kernel 'vecAdd' accepts at maximum 5 parameters.
<vecAdd.c>
77|
78|   clSetKernelArg(kAddition, 8, sizeof(cl_mem), &d_b);
   |                  ^^^^^^^^^^^^^ <--- kernel: vecAdd (untracked)
79|
```

Figure 3.8: Openhipify kernel argument error (case 2, example 2)

The OpenCL specification does allow for kernel arguments to be null, so the use of an error here would disallow valid programs. We therefore have included a flag `--no-kernel-arg-protection` to lower this to a warning if desired.

Investigating further into scenarios with known and unknown kernel declarations, we now discuss kernel argument types. Once again, due to the nature of OpenCL kernels not being known at compile time the types defined in host code for kernel arguments are `cl_mem`. This is an OpenCL wrapper for a memory object; we can simply think of this as a void pointer for generic areas of device-side memory. During runtime, these are implicitly cast to the correct type as per the kernel definition. We are not able to to simply rename `cl_mem` to `void*` however, HIP does not support implicit argument casting and will error in the compilation process. The arguments must be type correct. Let us take a closer look at some OpenCL host-side code, and see if there is a way to deduce the type:

```
1 /// ...
2 cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, ↩
    NULL);
3 clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, bytes, h_a, 0, NULL, ↩
    NULL);
4 clEnqueueNDRangeKernel(queue, kAddition, 1, NULL, &globalSize, ↩
    &localSize, 0,
5   NULL, NULL);
6 // ...
```

There is no discernable type information here, everything is referenced in terms of generic pointers or pointer wrappers. The type information is only present in the kernel declaration, therefore to ensure correct type checking for HIP translations the kernel definition must be present. Once again we are presented with two cases:

Possibility one: Kernel declaration is known at compile time

We know the type of the argument and can include an explicit cast in the HIP definition. The types of variables in the program can be changed from `cl_mem` to `void*`. We use `void*` for declaration instead of the type defined in the kernel as it more accurately reflects the OpenCL intention. For example, if a kernel wants to process an array of integers (argument type of `int*`), and then another kernel wants to instead read this memory in a byte representation (argument type of `unsigned char*`), OpenCL will implicitly handle type casting, whereas HIP will not and will error if the original variable is defined as `int*`. We therefore replace the `cl_mem` type with `void*` and generate explicit casts during kernel launches.

```
1 void *ma, *mr; // Previously: cl_mem ma, mr;
2 /////////// FUNCTION CONTINUES ///////////
3 hipLaunchKernelGGL(..., (const float *)ma, (float *)mr);
4 //     Explicit casts:  ^^^^^^^^^^^^^^^    ^^^^^^^^
```

Possibility two: Kernel declaration is not known at compile time

As stated previously we are unable to deduce argument type information without the kernel definition being tracked. We therefore skip cast generation and emit a warning message highlighting that the casts have not been generated. This does produce uncompileable code, but we believe it is a better option than throwing an error and halting translation. This choice of route would result in host code without kernel files present being untranslateable, which goes against one of our primary requirements.

3.5 Post translation actions

There are a few post-translation actions we must take, with the most important being header-file generation. These steps only take place if a 'full' OpenCL program is being translated (i.e. host and kernel files are both included; these actions do not take place in their respective passes). Our philosophy for these changes is that the OpenCL program is expected to be compilable therefore the HIP output should also be compilable directly after transpilation as detailed by our core requirement 3.1.1. We must include a strategy for linking together the kernel definitions and the kernel launches. We have presented multiple strategies for this in the upcoming section:

Our first strategy incorporated pasting the translated kernels into the host code. They would then be visible to the HIP compiler allowing kernel launches to link to their definition. Figure 3.9 showcases this solution. This is the simplest solution but produces a multitude of issues mainly stemming from the presence of duplicate kernel definitions. While this would compile with no errors, we instead step back and attempt to put ourselves in the user's position and recognise that the output HIP files will most likely still be in continued development. If a kernel definition requires modification, this would need to be done in multiple places resulting in a substandard developer experience.

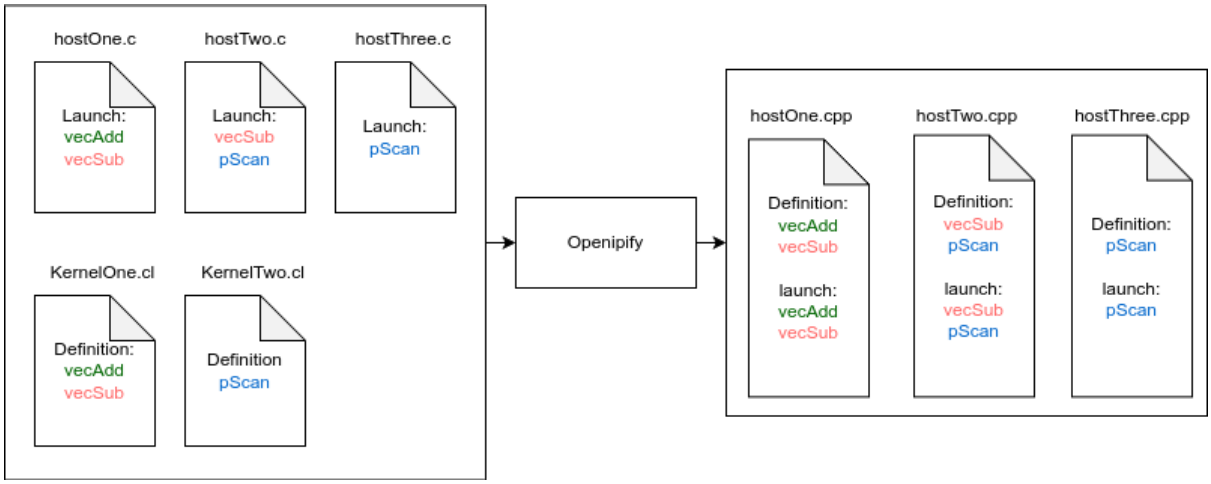

Figure 3.9: Kernel and host combination solution

Another issue would be the vast change in program structure. Translating a kernel file on its own produces the corresponding HIP file, but translating a kernel file with a host file would only produce a single host file with no HIP kernel file. While compilable, this is too vast a change in output for us to move forward with this solution.

Figure 3.10 showcases the solution we have implemented. This involves generating header files for the input kernel files containing declarations of the appropriate **launched** kernels. We highlight launched here as un-launched kernels along with device functions (i.e. only launchable on the target, not the host) we believe should not be exposed in the header file. Device functions can only be called via other kernels, not the host, so these functions will never be called host side and therefore should be kept private.

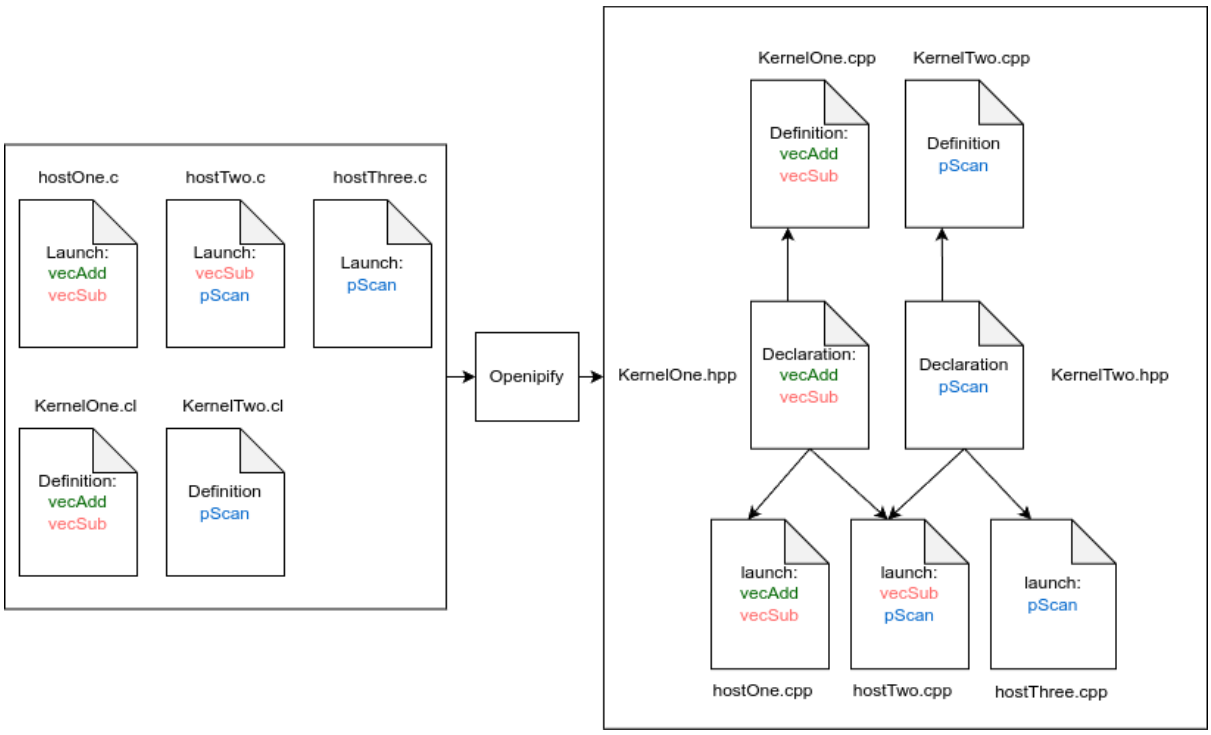

Figure 3.10: Header generation solution (Code example found in the 'header-gen' test case in the Openhipify test suite [25], creation detailed in section 4.1)

We then implement a strategy to reduce unnecessary includes from being prepended into host files. We identify the header containing kernel declarations with matching kernel launches and only include these matched headers in the produced HIP host file. When the HIP compiler is run with inputs of kernel files and host files, these includes then allow the linker to insert the kernels into the final binary. This then produces a finished and complete HIP executable which should be isomorphic to its OpenCL counterpart, meeting our core requirement.

3.6 Conclusion

We have discussed our strategy of development, splitting our problem into three distinct stages: kernel, host, and post-translation. We have justified our choice for utilising LLVM and explained what features we chose to include and exclude. Care has been taken regarding our tool's user experience, along with defenses for our choice and design of errors and warnings. Our tool can be found on GitHub [26] where it sees continued development and is open for contributions from the wider open-source community.

Chapter 4

Evaluation

Testing for Openhipify has been organised around two categories: translation ability and binary runtime performance. We believe these are the two main factors that will determine the usability of Openhipify. We have developed a test suite of OpenCL programs for these two sections, detailed in the following section.

4.1 Test set construction

We have used a selection of OpenCL sample repositories [5, 7, 9, 18, 34, 39] to combine into a single testing set [28] comprising of 9 OpenCL programs. The criteria for selection of the initial repositories was to encompass a wide range of program size, complexity and output. Through this we have evaluated the limits of Openhipify and have constructed accurate and realistic aims for the future.

From using a wide variety of samples we run into the first issue to fix in our validation set, being the different methods of loading OpenCL kernels. As discussed previously OpenCL kernels are compiled during runtime which only requires the programmer to pass in the kernel as a string to OpenCL. Lots of samples would instead of loading in the kernel from a kernel file simply store the kernel definition directly as a string in the host program:

```
const char *kernelSource =
  "__kernel void mult(__global double *v) {        \n"  \
  "    int id = get_global_id(0);                  \n"  \
  "    v[id] = 2*v[id];                            \n"  \
  "}                                               \n"; \
// ...........
program = clCreateProgramWithSource(context, 1, (const char **) & ↩
    kernelSource, NULL, &err);
```

We do not translate these kernels via Openhipify, we instead require kernels to be present in a separate file and then be loaded in. This is detailed in the Openhipify docs section along with necessary steps to take to ensure valid translation. We include auxiliary OpenCL build functions in the Openhipify GitHub repository [27] to allow users to easily modify their programs to be Openhipify compliant. After these changes are made to the incompatible programs we can now utilise the test suite.

```
// mult.cl
__kernel void mult(__global double *v) {
  int id = get_global_id(0);
  v[id] = 2*v[id];
}
// host.c
// ...........
program = build_program(context, "mult.c");
//        ^^^^^^^^^^^^^ <--- Openhipify provided auxiliary function
```

4.2 Translation ability

This section will evaluate the ability of Openhipify to correctly parse and interpret OpenCL programs and output syntactically correct and high-quality HIP code that compiles down to an isomorphic binary. We run our test cases through Openhipify, compile them with hipcc, and compare the resulting executable with the original OpenCL one to see if any differences in structure and output are present. Figure 4.1 demonstrates our results along with Appendix B.2 showcasing a section of translation taken from the 'dijkstra' test case.

Compilation results		
Test case	Compiles	Isomorphic binary [1]
hello-world		
mat-add		
square-array		
matrix-mul		
sum-array		
vec-add-simple		
dijkstra		
qr		
mandelbrot		

Table 4.1: Test set compilation results

Openhipify did not crash during transpilation for any of the test cases but did produce a variety of warnings and errors which were useful during post-translation actions. More than half of our test cases failed HIP compilation. While disappointing, we believe this is not reflective of our inability to build a functional transpiler but instead indicative of the large number of different compilation cases available. The errors produced were primarily due to a few syntactically incorrect lines where after manual edits produced compilation. This is not a case of us having to modify our translation framework, we would instead simply have to add more translation cases. We take note of these in our future plans section, and move on with testing after manual changes are made.

Figure 4.1 showcases the modifications we have made to the un-compilable HIP files (with an example diff file shown in Appendix B.3). We can see most of the necessary modifications we have made are surrounding the host files as opposed to the kernel and header files. This is unsurprising as the host file pass in Openhipify is more structurally modifying than the kernel file pass or the header generation stage. We generally had to modify less than 10 lines for compilation to succeed, preserving >99% of the original translation. These changes were centered around OpenCL semantics that we did not anticipate during the development of Openhipify. For example, I was unaware that OpenCL permits a null pointer argument to be passed into the kernel work and group size parameters. This was translated as is by Openhipify and produced a compiler error from hipcc as this is not permitted in HIP. This we have marked for future development and moved on.

[1] Isomorphic: Identical output to the OpenCL binary, detailed in our Requirements Section 3.1.1

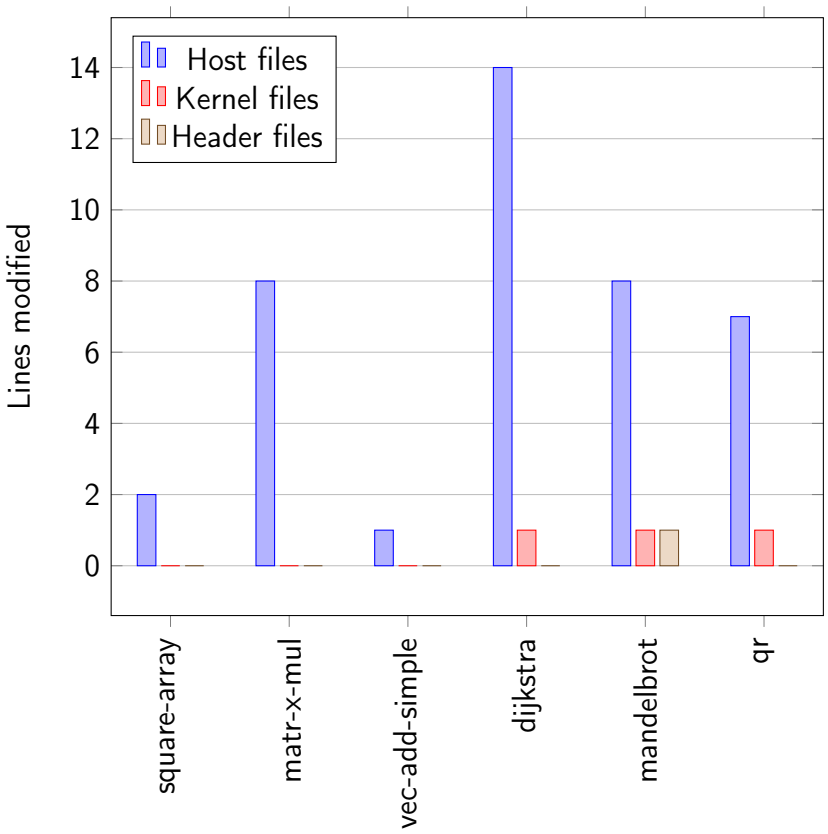

Figure 4.1: Modifications to test set compilation fails

After modification, we rerun our original suite, producing Figure 4.2. We see that 7 out of the 9 of the samples showcased isomorphic binaries. We have tested isomorphism by printing the results of the outputs and comparing them using the Linux `cmp` program [24]. We also inspect the number of kernels launched, along with the block and grid sizes in each to see if there are any discrepancies. Our failed test cases, 'square-array' and 'mandelbrot' produced non-isomorphic binaries for reasons in this final category.

Compilation results		
Test case	Compiles	Isomorphic binary
hello-world		
mat-add		
square-array		
matrix-mul		
sum-array		
vec-add-simple		
dijkstra		
qr		
mandelbrot		

Table 4.2: Test set compilation results (After modification)

Inspecting 'square-array', the HIP program does indeed produce identical results for the same input used in the original OpenCL program. This binary is only revealed to be non-isomorphic during the inspection of kernel block sizes. In OpenCL, this was calculated via a call to `texclGetKernelWorkGroupInfo` with the `CL_KERNEL_WORK_GROUP_SIZE` parameter, producing the value 256. We translate this in Openhipify to a call to `hipGetDeviceProperties`, and then extract the `maxThreadsPerBlock` member, which instead returns 1024. This discrepancy is especially worrying as the maximum number of threads per block is a hardware limitation, it should be independent of the software framework. The card we test on is a Nvidia GTX 1060, so we can use the Nvidia provided binary `deviceQuery` provided in the CUDA SDK. This sample 'enumerates the properties of the CUDA devices present in the system' [30], so we can find the actual hardware limit for this value. This is shown to indeed be 1024 (shown Appenix A.1). This highlights one of the advantages of HIP over OpenCL where results are more specific and device tailored. The 'mandelbrot' test case exhibited similar properties which is the cause for non-isomorphism.

Taking a step back from binary results, we now investigate the quality of HIP code produced in comparison to the OpenCL input. We initially stated that HIP code was less verbose than OpenCL which we can evaluate by calculating the percentage difference in source file size; Less verbose code should generally be shorter and more readable. Kernel files typically remain the same size due to the non-invasive kernel translation pass, so we instead investigate the difference in host file size (specifically the difference in the number of characters present), shown in Figure 4.2.

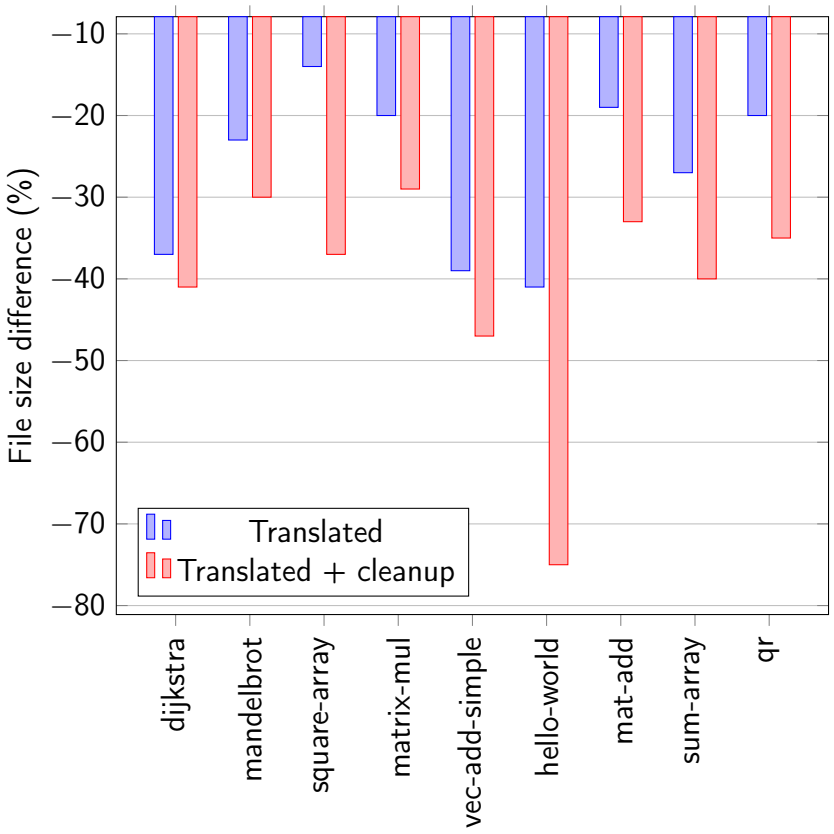

Figure 4.2: Percentage difference of host source file size

`Translated` refers to the raw translated file generated through Openhipify, `Translated + cleanup` contains manual changes to remove dangling statements and redundant comments. An example of one of these changes would be to remove comment chains referencing culled OpenCL intrinsics:

```
// Populate input array
for (i = 0; i < ARRAY_SIZE; i++) {
  hdata[i] = 1.0f * i;
}

/* Create device and context

Creates a context containing only one device - the device structure
created earlier.
*/

/* Build program */

/* Create a command queue */

/* Create data buffer
Create the input and output arrays in device memory for our
calculation. 'd' below stands for 'device'.
*/
hipMalloc((void **)&ddata, bytes);
hipMalloc((void **)&doutput, bytes);
```

Openhipify of course does not remove this, we skip over comments. We can see a reduction average of around 40%. This reduction will be the removal of argument-setting calls, large verbose memory-related calls, et cetera. OpenCL functions typically accept more parameters which we are able to cull which is another cause for reduction here. We believe this produces more readable code, and shows another example of the benefits of HIP over OpenCL, especially when the produced program showcases identical outputs.

4.3 Runtime Performance Analysis

The machine tested on can be seen in Appendix A.1 and A.2. To evaluate runtime performance we measure the difference in runtime of the translated HIP binaries against the OpenCL binaries, shown in Figure 4.3. This is the percentage difference between runs of the HIP program compared to the OpenCL equivalent. The test cases 'dijkstra' and 'hello-world' have not been included due to the speed of each binary being less than half a second; this would only produce inconsistent and inconclusive results. We see that 5 out of 7 binaries produce favourable results, with the HIP programs completing their task in a shorter space of time. If we remove the outliers of 'qr' and 'sum-array' which showcase vastly uneven runtimes, we see a 1.55% performance increase on average. Parallel programming thrives on small but consistent speedups. When thousands of devices and human hours are spent on a problem each percentage of performance increase can save vast sums of money and time. We are generally happy with these performance results, but now choose to investigate the large, negative outlier in the test case 'qr'.

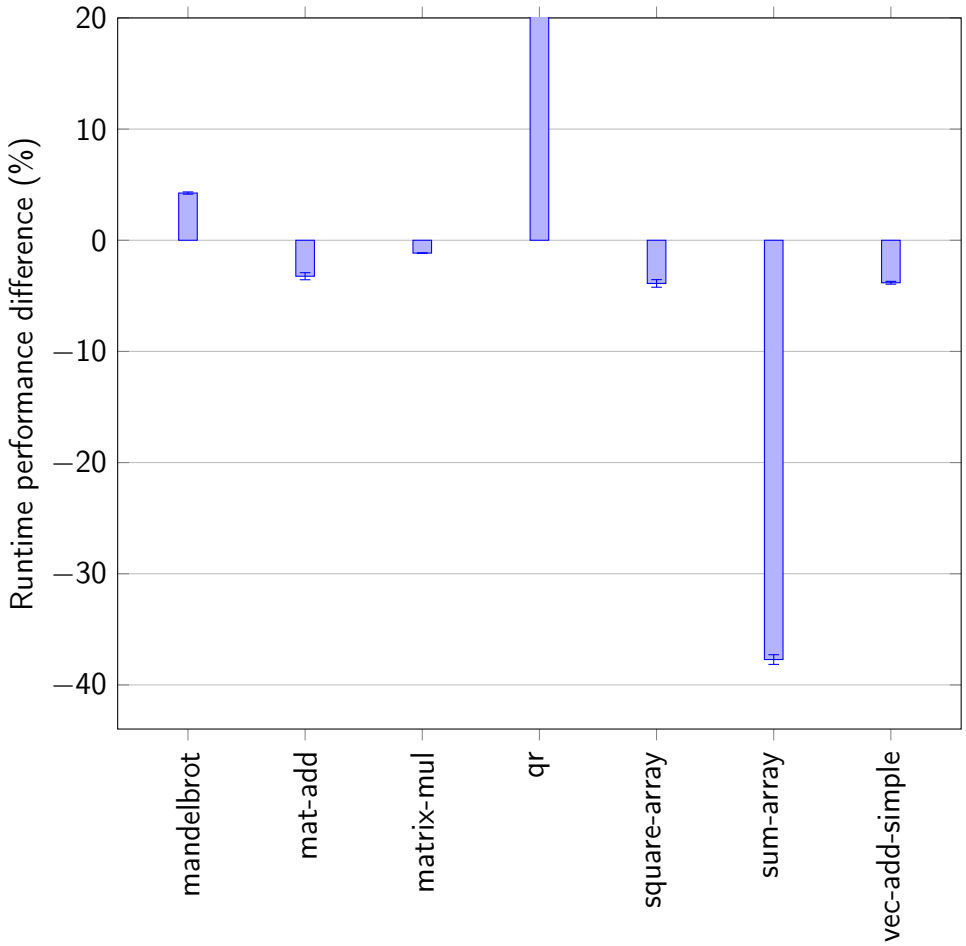

Figure 4.3: Test case runtime performance (HIP against OpenCL)

The 'qr' HIP program finished in around 25 seconds while the OpenCL program finished in less than 1 second. After investigation, we found the cause is due to the array pattern access with the kernel launch parameters, which are handled differently by HIP and OpenCL. Care would have to be taken during the translation of 'qr' with some required manual changes. We have not made these in this report as we are testing the performance of the code produced via Openhipify, not HIP against OpenCL. We speculate that if these measures are taken then the HIP binary will be equivalent performance-wise to the OpenCL binary.

'matrix-mul' showcases a wide variety of OpenCL and HIP techniques, and produces large GPU computation. Matrix multiplication is vital for modern-day problems, for example solving linear systems of equations, weather modeling and deep neural network training. For translation we used Openhipify with small manual changes surrounding the 2D kernel launching, which is currently not yet implemented. Figure 4.4 showcases the percentage difference in the binary runtime performance of the HIP program compared to the OpenCL program. We can see a large performance increase in the HIP program for small problem sizes which will be the cause of OpenCL's more expensive setup costs (discussed later in this section). As the problem size increases, the performance of both programs converge, with the final tested size of 16000 x 16000 arrays having a 1.16% performance increase. This will be due to a greater proportion of time being taken up by the GPU kernel, which will be highly optimised for both OpenCL and HIP platforms.

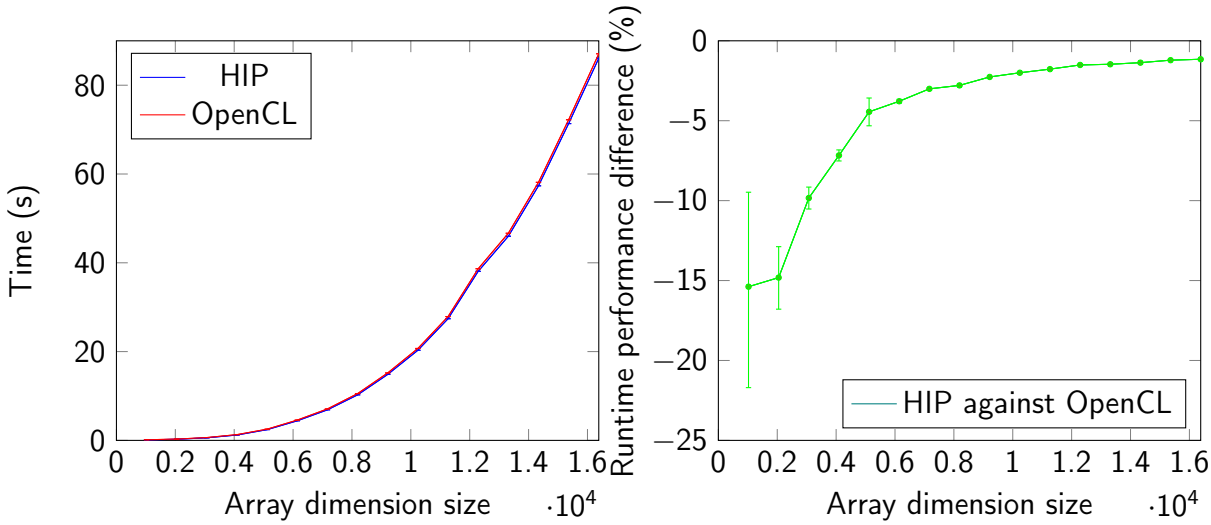

Figure 4.4: HIP matrix multiplication binary compared to OpenCL binary

By varying the number of iterations of the program and keeping the problem size constant (4096 x 4096), we observe a much greater deviation in runtime performance, shown in Figure 4.5.

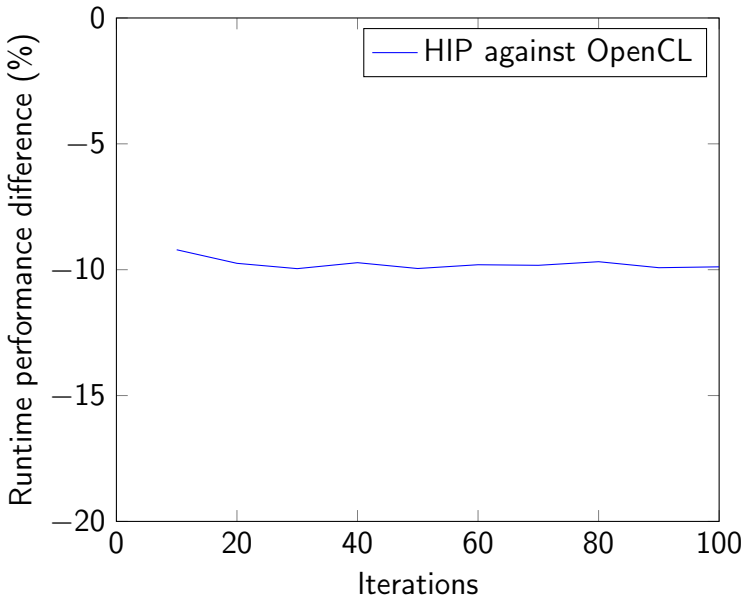

Figure 4.5: HIP vs OpenCL matrix multiplication performance (4096 x 4096)

We see a near-constant 10% decrease for the HIP binary completion time compared to OpenCL. We can attribute this much larger change to the extra runtime compilation overhead that OpenCL introduces; an overhead hoisted into compile time for HIP. Figure 4.6 shows a further breakdown of this statistic. Setup time refers to memory allocation and initialisation, and also kernel compilation. The kernel time is the physical time that the kernel is run on the GPU. We see much greater deviance in the setup time in comparison to the kernel time, with HIP showcasing a 17% performance increase. There is also a constant difference in kernel runtime, with HIP being 2% faster than OpenCL.

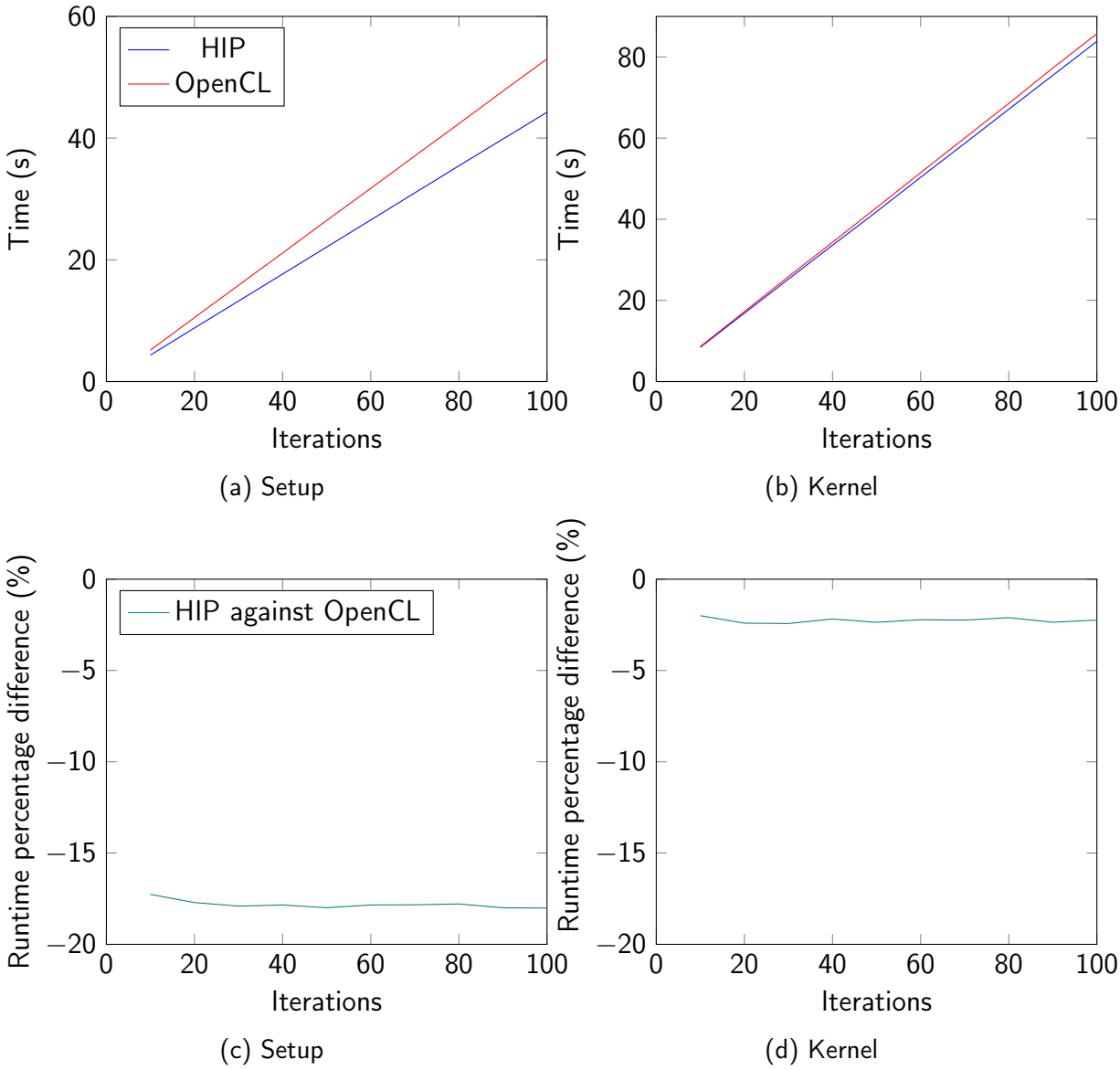

Figure 4.6: Runtime of different sections of HIP and OpenCL matrix multiplication (4096 x 4096)

We speculate the runtime performance increase is due to the general performance superiority of the CUDA runtime in comparison to the OpenCL implementation (discussed in section 2.1.1). A drawback of our investigation however is the lack of testing on AMD GPU hardware as we did not have access to an AMD card. We instead look at the findings of Kerscher [19], where OpenCL and HIP were investigated on an AMD platform. We see that for most test cases the OpenCL and HIP performance were roughly equivalent and that cases where one platform outperformed another were split evenly between them.

On average we see faster binaries when OpenCL is translated to HIP via Openhipify. Manual changes will still need to be made to the produced HIP code at this current stage to produce the final HIP binary, but we hope future development will minimise this added cost.

Chapter 5

Conclusion

5.1 Future Work

We believe we have developed a suitable base framework for Openhipify with the general translation pipeline being complete. Future work therefore primarily surrounds adding more translation features and cases. This has been driven by our evaluation section, namely from features preventing our test cases from compiling. As this is related to our core requirement we define the following improvements to be of high priority. Many of the OpenCL samples used multiple devices, something which we would have been unable to test so we chose to omit in Openhipify. This would be the first feature we would implement in the future. Other features include null kernel argument integration, kernel local memory space translation, and multi-dimensional kernel launch syntax support. With these implementations, our test cases would all pass the HIP compilation step. We could continue listing extra features to integrate, but this would stretch far beyond our set requirements for Openhipify; The OpenCL and HIP APIs are vast.

Taking a step back, a shift in development strategy would instead need to be taken. Modern transpilers surrounding GPGPU programming (and programming in general), are rarely created and maintained by a single developer. CU2CL was developed by a team of 4 [21], and has been deprecated as of 2017. HIPIFY is managed by AMD but developed via open-source contributions, with 42 contributors at the time of writing. SnuCL-Tr was developed by a group of 5 at the University of Seoul (since deprecated). For a project with a single developer, Openhipify was destined to be limited in scope and would require a larger team to cover a wider OpenCL vocabulary.

Improvements to the Openhipify framework are still something we aim to develop, with static control flow analysis improvements as our main aim. This would mainly take the form of dominator tree analysis. In graph theory, a node v dominates a node v' if every path from the entry node to v' intercepts v. A dominator tree encodes this information in a separate graph, where a node's children are dominated by it. This would be particularly useful for us when analysing control flow graphs in order to generate accurate kernel launch parameters. Consider the following OpenCL pseudocode:

```
1  ChangeArg3();
2
3  if (Cond1()) {
4    if (Cond2()) {
5      ChangeArg2();
6    }
7
8    ChangeArg1();
9  }
10
11 LaunchKernel();
```

This showcases a kernel with 3 parameters, all being initially set. Parameters are changed conditionally and the kernel is launched. Currently, Openhipify will show a warning message,

and skip the translation of such a case (equivalent error showcased in Figure 3.5). We are currently unable to evaluate what parameters should be set, we require a more complex analysis method. This would be possible if we undertook dominator tree analysis. We compile our input program to LLVM IR, run all compiler optimisations, and generate a control flow graph via LLVM, Figure 5.1, which encodes the different branches through our program. We can construct a dominator tree from this, shown in Figure 5.2.

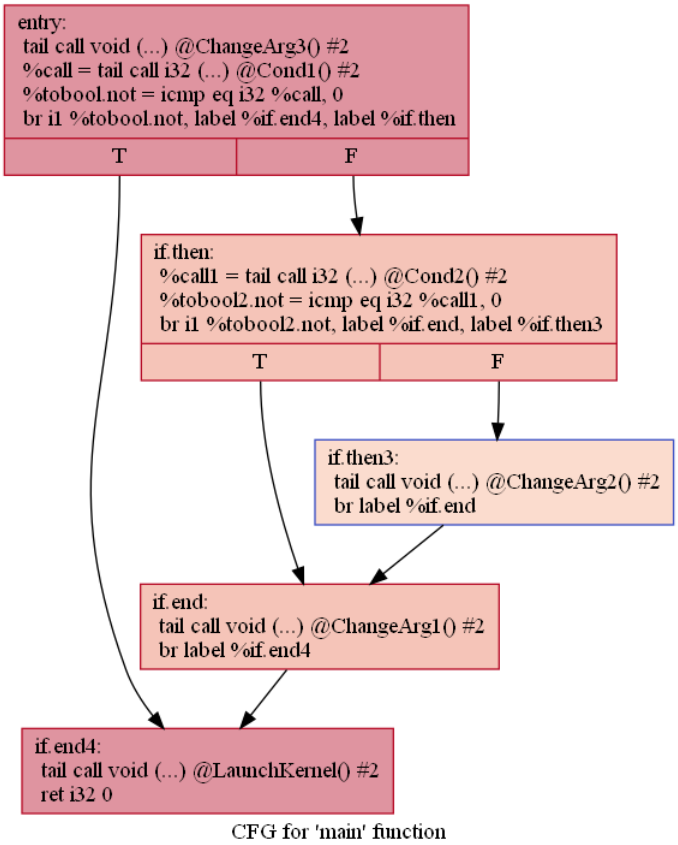

Figure 5.1: Control Flow Graph

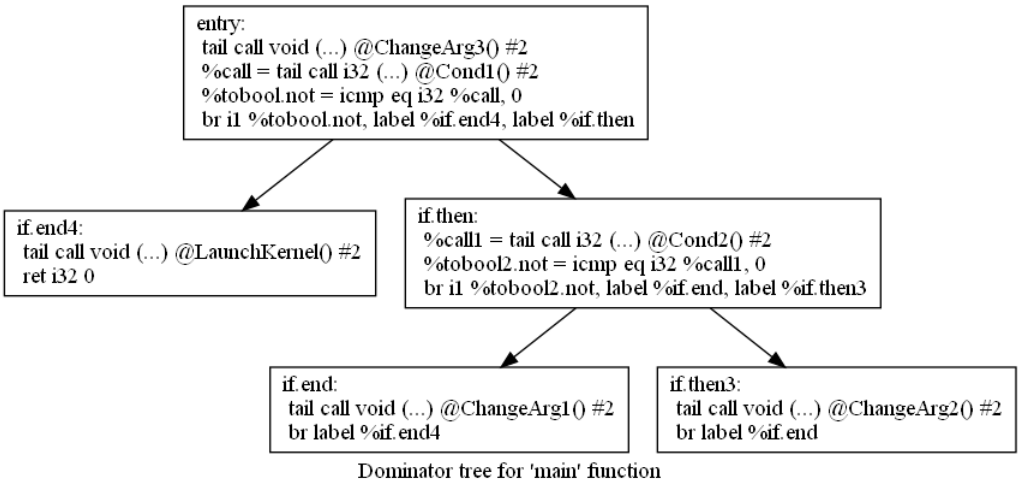

Figure 5.2: Dominator tree

We can now start making useful insights. Firstly, we can see that the call to `LaunchKernel` is dominated by a node containing the `ChangeArg3` call. This means that all routes through our program to launching the kernel pass through this call, therefore this argument will always be modified. Taking a look at our pseudocode this is obvious, but using a dominator tree provides analytical certainty. We can also see that the calls to change the other arguments are not dominating the launch call i.e. these calls do not happen for certain, they rely on conditional branches.

In order to produce a HIP translation we need all `ChangeArg` calls to dominate a launch instruction. We therefore must modify our control flow graph to produce this. By duplicating certain basic blocks and changing their branch targets, we produce 5.3.

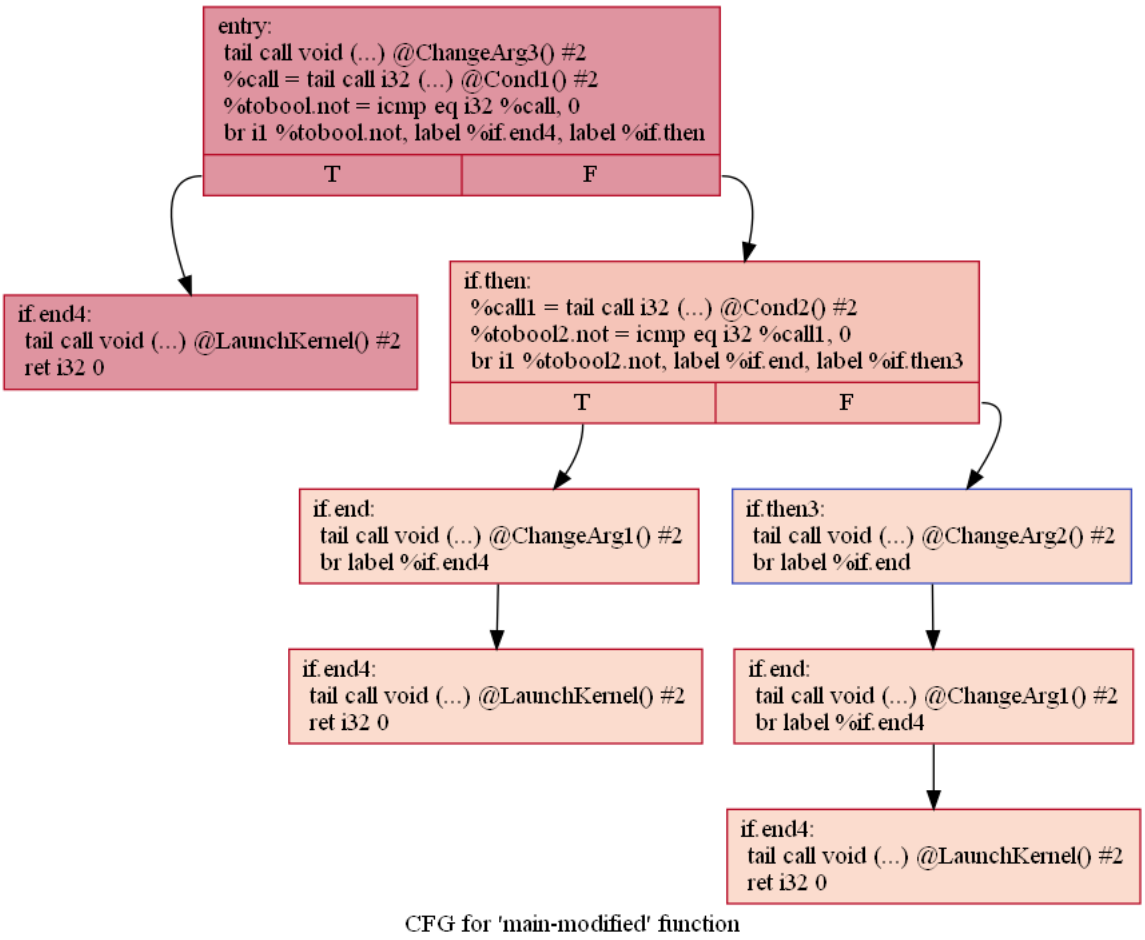

Figure 5.3: Modified CFG

This graph ends up being its own dominator tree by chance. We can now see that all calls to `ChangeArg` dominate a call to `LaunchKernel`. We can traverse up this dominator tree from each call to `LaunchKernel` to the entry node, recording branches taken and `ChangeArg` calls passed in order to generate a HIP equivalent. After doing this we produce the following:

```
if (Cond1()) {
  if (Cond2()) {
    LaunchKernel(modified, modified, modified);
  } else {
    LaunchKernel(modified, original, modified);
  }
} else {
  LaunchKernel(original, original, modified);
}
```

By cross-referencing to the original OpenCL pseudocode we can see this is isomorphic. After future implementation, Openhipify could be much more confident in kernel launch translation which would enlarge the set of possible OpenCL programs that can be translated correctly.

5.2 Requirements Evaluation

We are generally very pleased with our development efforts. Our core requirement states that we need a defined pipeline that has the potential to produce compileable and isomorphic HIP code. In section 4.2 we can see for certain cases this is indeed true, however this occurs infrequently; more work is needed to increase the size of the set of translatable programs. We believe we have met our core requirement by definition, but do wish in the future to produce more favourable results.

Our primary requirements stated the importance of developing a modular pipeline with separate host and kernel side passes that can work independently of each other. We have firmly met this requirement: kernel and host code can be individually translated as our Openhipify driver will select the individual pass for each. When used together we make use of the whole pipeline and introduce extra steps, for example post-translation actions detailed in Section 3.5.

We believe we have firmly met our secondary requirements which state that we should undergo extra syntactic analysis of the OpenCL input program, and also provide professional user messages. Examples of these messages can be seen in Section 3.4, where we take inspiration from the Clang and Rust compiler error messages (Examples in Appendix B.4), which are industry standard tools. In terms of additional syntactic analysis, while we do perform small amounts we hoped for more, and have laid out our future plans in the previous section.

5.3 Summary

We proposed the creation of a transpiler that can translate OpenCL to HIP. We developed a kernel and host pipeline for the two different sections of OpenCL code which converge to a HIP program. We utilised AST traversal to determine the initial intention of the program and utilised LLVM to generate suitable HIP replacements for OpenCL semantics. Care was taken for programmer usability, with concise and useful error and warning messages being propagated through the tool to the user to aid in translation.

While we are disappointed with the compilation ability of the produced HIP code, we recognise what next steps should be taken for improvement in this area. We are pleased with the improvements seen in the performance of the translated code, and the readability and structure of it. We have published this tool on Github [26] (README in Appendix B.1), along with

our test suite and results [25]. We hope for the wider OpenCL and HIP community to take note and join us in continued development. We are optimistic about the direction of GPGPU programming and hope enthusiasm stays prominent in this field, along with the continued development of HIP, OpenCL, and CUDA.

Word count: 10,639

Bibliography

[1] 19.0.0 clang, 2024. *clang::declrefexpr class reference* [Online]. Available from: https://clang.llvm.org/doxygen/classclang_1_1DeclRefExpr.html [Accessed 25/03/2024].

[2] AMD, 2024. *Hip porting guide* [Online]. Available from: https://github.com/ROCm/HIP/blob/develop/docs/old/user_guide/hip_porting_guide.md [Accessed 28/04/2024].

[3] AMD, 2024. *Hipify* [Online]. Available from: https://github.com/ROCm/HIPIFY [Accessed 21/03/2024].

[4] AMD, 2024. *Rocm* [Online]. Available from: https://github.com/ROCm/ROCm [Accessed 21/03/2024].

[5] avinabadasgupta, 2017. *Opencl graph theory* [Online]. Available from: https://github.com/avinabadasgupta/OpenCL-Graph-Theory [Accessed 10/04/2024].

[6] Clemente, C., Di Bisceglie, M., Di Santo, M., Ranaldo, N. and Spinelli, M., 2009. Processing of synthetic aperture radar data with gpgpu. *2009 ieee workshop on signal processing systems*. IEEE, pp.309–314.

[7] Dakkers, 2019. *Opencl examples* [Online]. Available from: https://github.com/Dakkers/OpenCL-examples [Accessed 25/03/2024].

[8] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. and Dongarra, J., 2012. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel computing*, 38(8), pp.391–407.

[9] ecatue, 2017. *Matrix multiplication opencl* [Online]. Available from: https://ecatue.gitlab.io/GPU2016/cookbook/matrix_multiplication_opencl/ [Accessed 10/04/2024].

[10] Eklind, R., 2018. *Llvm ir and go* [Online]. Available from: https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/ [Accessed 19/02/2024].

[11] Fang, J., Varbanescu, A.L. and Sips, H., 2011. A comprehensive performance comparison of cuda and opencl. *2011 international conference on parallel processing*. IEEE, pp.216–225.

[12] Gardner, M., Sathre, P., Feng, W.c. and Martinez, G., 2013. Characterizing the challenges and evaluating the efficacy of a cuda-to-opencl translator. *Parallel computing*, 39(12), pp.769–786.

[13] Group, K., 2020. *Offline compilation of opencl kernels* [Online]. Available from: https://www.khronos.org/blog/offline-compilation-of-opencl-kernels-into-spir-v-using-open-source-tooling [Accessed 21/03/2024].

[14] Group, K., 2024. *About the khronos group* [Online]. Available from: https://www.khronos.org/about [Accessed 21/03/2024].

[15] Group, K., 2024. *Khronos members* [Online]. Available from: https://www.khronos.org/members/list [Accessed 17/04/2024].

[16] Group, K., 2024. *Opencl overview* [Online]. Available from: https://www.khronos.org/opencl/ [Accessed 21/03/2024].

[17] Hernández, M., Guerrero, G.D., Cecilia, J.M., García, J.M., Inuggi, A., Jbabdi, S., Behrens, T.E. and Sotiropoulos, S.N., 2013. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. *Plos one*, 8(4), p.e61892.

[18] jeremyong, 2013. *opencl in action* [Online]. Available from: `https://github.com/jeremyong/opencl_in_action` [Accessed 25/03/2024].

[19] Kerscher, N., 2022. Investigating the hip programming model with regards to portability and performance portability. *Seminar on performance portable programming of hpcapplications*.

[20] Kim, J., Dao, T.T., Jung, J., Joo, J. and Lee, J., 2015. Bridging opencl and cuda: a comparative analysis and translation. *Proceedings of the international conference for high performance computing, networking, storage and analysis*. pp.1–12.

[21] Lab, S., 2017. *Cu2cl* [Online]. Available from: `https://chrec.cs.vt.edu/cu2cl/` [Accessed 10/04/2024].

[22] Larsen, E.S. and McAllister, D., 2001. Fast matrix multiplies using graphics hardware. *Proceedings of the 2001 acm/ieee conference on supercomputing*. pp.55–55.

[23] LLVM, 2024. *Llvm website* [Online]. Available from: `https://llvm.org/` [Accessed 10/04/2024].

[24] man7, 2024. *cmp(1) — linux manual page* [Online]. Available from: `https://man7.org/linux/man-pages/man1/cmp.1.html` [Accessed 26/04/2024].

[25] Mudd, B., 2024. *Openhipify* [Online]. Available from: `https://github.com/BenJMudd/openhipify-test-suite` [Accessed 11/04/2024].

[26] Mudd, B., 2024. *Openhipify* [Online]. Available from: `https://github.com/BenJMudd/openhipify` [Accessed 11/04/2024].

[27] Mudd, B., 2024. *Openhipify auxiliary functions* [Online]. Available from: `https://github.com/BenJMudd/openhipify/tree/master/docs/auxiliary-functions` [Accessed 19/04/2024].

[28] Mudd, B., 2024. *Openhipify test suite* [Online]. Available from: `https://github.com/BenJMudd/openhipify-test-suite` [Accessed 25/03/2024].

[29] NVIDIA, 2023. *Cuda zone* [Online]. Available from: `https://developer.nvidia.com/cuda-zone` [Accessed 21/03/2024].

[30] NVIDIA, 2024. *Cuda samples* [Online]. Available from: `https://github.com/nvidia/cuda-samples` [Accessed 05/04/2024].

[31] Nvidia, 2024. *Parallel thread execution isa version 8.4* [Online]. Available from: `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html` [Accessed 26/04/2024].

[32] Peddia, J., 2018. *Add-in board report* [Online]. Available from: `https://www.jonpeddie.com/news/jon-peddie-research-releases-its-q3-2018-add-in-board-report/` [Accessed 21/03/2024].

[33] Petrovič, F., Střelák, D., Hozzová, J., Ol'ha, J., Trembeckỳ, R., Benkner, S. and Filipovič, J., 2020. A benchmark set of highly-efficient cuda and opencl kernels and its

dynamic autotuning with kernel tuning toolkit. *Future generation computer systems*, 108, pp.161–177.

[34] rsnemmen, 2021. *Opencl examples* [Online]. Available from: `https://github.com/rsnemmen/OpenCL-examples` [Accessed 25/03/2024].

[35] Steam, 2023. *Steam hardware survey* [Online]. Available from: `https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam` [Accessed 21/03/2024].

[36] Sun, Y., 2022. *Accelerated computing with hip*. Advanced Micro Devices, Inc.

[37] Top500, 2023. *Top 500* [Online]. Available from: `https://www.top500.org/lists/top500/2023/11/` [Accessed 21/03/2024].

[38] Tsai, Y.M., Cojean, T., Ribizel, T. and Anzt, H., 2020. Preparing ginkgo for amd gpus–a testimonial on porting cuda code to hip. *European conference on parallel processing*. Springer, pp.109–121.

[39] ysh3299, 2022. *Opencl101* [Online]. Available from: `https://github.com/ysh329/OpenCL-101` [Accessed 25/03/2024].

Appendix A

Device Specifications

```
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          12.3 / 12.3
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6073 MBytes (6367543296 bytes)
  (010) Multiprocessors, (128) CUDA Cores/MP:    1280 CUDA Cores
  GPU Max Clock rate:                            1759 MHz (1.76 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.3, CUDA Runtime Version = 12.3, NumDevs = 1
Result = PASS
```

Figure A.1: deviceQuery results

```
benmudd@benmudd-B360M-DS3H:~$ inxi -Fxz
System:
  Kernel: 5.15.0-102-generic x86_64 bits: 64 compiler: N/A
  Desktop: Gnome 3.36.9 Distro: Ubuntu 20.04.6 LTS (Focal Fossa)
  Machine:
  Type: Desktop System: Gigabyte product: B360M-DS3H v: N/A serial: <filter>
  Mobo: Gigabyte model: B360M DS3H v: x.x serial: <filter>
  UEFI: American Megatrends v: F3 date: 03/01/2018
CPU:
  Topology: 6-Core model: Intel Core i7-8700K bits: 64 type: MT MCP
  arch: Kaby Lake rev: A L2 cache: 12.0 MiB
  flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx
  bogomips: 88796
  Speed: 800 MHz min/max: 800/4700 MHz Core speeds (MHz): 1: 800 2: 4214
  3: 3229 4: 1968 5: 802 6: 801 7: 800 8: 800 9: 800 10: 800 11: 800 12: 800
Graphics:
  Device-1: NVIDIA GP106 [GeForce GTX 1060 6GB] vendor: Micro-Star MSI
  driver: nvidia v: 545.23.08 bus ID: 01:00.0
  Display: x11 server: X.Org 1.20.13 driver: nvidia
  unloaded: fbdev,modesetting,nouveau,vesa resolution: 1920x1080~60Hz
  OpenGL: renderer: NVIDIA GeForce GTX 1060 6GB/PCIe/SSE2
  v: 4.6.0 NVIDIA 545.23.08 direct render: Yes
Audio:
  Device-1: Intel Cannon Lake PCH cAVS vendor: Gigabyte
  driver: snd_hda_intel v: kernel bus ID: 00:1f.3
  Device-2: NVIDIA GP106 High Definition Audio vendor: Micro-Star MSI
  driver: snd_hda_intel v: kernel bus ID: 01:00.1
  Sound Server: ALSA v: k5.15.0-102-generic
Network:
  Device-1: Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet
  vendor: Gigabyte driver: r8169 v: kernel port: 3000 bus ID: 03:00.0
  IF: enp3s0 state: up speed: 1000 Mbps duplex: full mac: <filter>
Drives:
  Local Storage: total: 2.73 TiB used: 248.16 GiB (8.9%)
  ID-1: /dev/nvme0n1 vendor: Samsung model: SSD 970 EVO Plus 2TB
  size: 1.82 TiB
  ID-2: /dev/sda vendor: Samsung model: SSD 870 EVO 1TB size: 931.51 GiB
  Partition:
  ID-1: / size: 671.79 GiB used: 248.13 GiB (36.9%) fs: ext4
  dev: /dev/nvme0n1p5
Sensors:
  System Temperatures: cpu: 41.0 C mobo: 27.8 C gpu: nvidia temp: 38 C
  Fan Speeds (RPM): N/A gpu: nvidia fan: 35%
Info:
  Processes: 367 Uptime: 2h 38m Memory: 31.28 GiB used: 2.94 GiB (9.4%)
  Init: systemd runlevel: 5 Compilers: gcc: 9.4.0 Shell: bash v: 5.0.17
  inxi: 3.0.38
```

Figure A.2: inxi results

Appendix B

Openhipify Resources

OpenHipify

Openhipify is an OpenCL to HIP transpiler, currently still in development. This can be used for OpenCL C translation (i.e. OpenCL kernels), and for the OpenCL 3.0 C API (C++ API is currently not supported).

Installation

Openhipify uses LibTooling in clang, and requires LLVM to build and use.

1. Download the LLVM source from the LLVM github.
2. Clone the openhipify repository into the tools section:

```
cd /llvm_project/clang/tools
git clone https://github.com/benjmudd/openhipify
```

3. Proceed with standard installation process (Found on LibTooling documentation)

```
cd /llvm_project
echo "add_clang_subdirectory(openhipify)" > clang/tools/CMakeLists.txt
mkdir build && cd build
cmake -G Ninja ../llvm-project/llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -
DCMAKE_BUILD_TYPE=Release -DLLVM_BUILD_TESTS=ON
ninja openhipify
```

The binary will be produced in `/llvm_project/build/bin`

Usage

Openhipify can be invoked on OpenCL kernel files (.cl extentions) and OpenCL host files using the OpenCL 3.0 C API.

```
# Kernel example
openhipify kernel.cl --
#host example
openhipify host.c --
#dual example
openhipify host.c kernel.cl --
```

Multiple host and kernel files can be inputted in any order into the tool. Each will be translated into a HIP file with the same name, with an extra .cpp extention (i.e. kernel.cl -> kernel.cl.cpp)

Examples of full translation can be found at the Openipify test suite.

Figure B.1: Openhipify README

```
#include "defs.h"
#include <CL/cl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_SOURCE_SIZE (0x100000)
int main() {
  /////////////// SECTION START ///////////////
  cl_mem Matrix, Visited, Min, Index, Array, Path;
  Matrix = clCreateBuffer(context, CL_MEM_READ_WRITE,
                          sizeof(float) * n * n, NULL, &err);
  Visited = clCreateBuffer(context, CL_MEM_READ_WRITE,
                           sizeof(float) * n, NULL, &err);
  Min = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * n,
                       NULL, &err);
  Index = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(int) * n,
                         NULL, &err);
  Array = clCreateBuffer(context, CL_MEM_READ_WRITE,
                         sizeof(float) * n, NULL, &err);
  Path = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(int) * n,
                        NULL, &err);
  clEnqueueWriteBuffer(command_queue, Matrix, CL_TRUE, 0,
                       sizeof(float) * n * n, matrix, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Visited, CL_TRUE, 0,
                       sizeof(float) * n, visited, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Min, CL_TRUE, 0,
                       sizeof(float) * n, minimum, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Index, CL_TRUE, 0,
                       sizeof(int) * n, index, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Array, CL_TRUE, 0,
                       sizeof(float) * n, array, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Path, CL_TRUE, 0,
                       sizeof(int) * n, path, 0, NULL, NULL);
  size_t globalsize[1];
  globalsize[0] = 452;
  size_t localsize[1];
  localsize[0] = 113;
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &Matrix);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &Visited);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &Min);
  clSetKernelArg(kernel, 3, sizeof(cl_mem), &Index);
  clSetKernelArg(kernel, 4, sizeof(cl_mem), &Array);
  clSetKernelArg(kernel, 5, sizeof(cl_mem), &Path);
  clSetKernelArg(kernel, 6, sizeof(float) * 113, NULL);
  clSetKernelArg(kernel, 7, sizeof(float) * 113, NULL);
  clock_t tic = clock();

  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalsize,
                         localsize, 0, NULL, &event);
  clock_t toc = clock();
  clEnqueueReadBuffer(command_queue, Matrix, CL_TRUE, 0,
                      sizeof(float) * n * n, matrix, 0, NULL, NULL);
  /////////////// SECTION END ///////////////
```

```
////////////////// Generated by Openhipify //////////////////
#include "hip/hip_runtime.h"

#include "dijkstragpu_kernel.cl.hpp"
/////////////////////////////////////////////////////////////

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_SOURCE_SIZE (0x100000)
int main() {
  /////////////// SECTION START ///////////////
  void *Matrix, *Visited, *Min, *Index, *Array, *Path;
  err = hipMalloc((void **)&Matrix, sizeof(float) * n * n);
  err = hipMalloc((void **)&Visited, sizeof(float) * n);
  err = hipMalloc((void **)&Min, sizeof(float) * n);
  err = hipMalloc((void **)&Index, sizeof(int) * n);
  err = hipMalloc((void **)&Array, sizeof(float) * n);
  err = hipMalloc((void **)&Path, sizeof(int) * n);
  hipMemcpy(Matrix, matrix, sizeof(float) * n * n,
            hipMemcpyHostToDevice);
  hipMemcpy(Visited, visited, sizeof(float) * n,
            hipMemcpyHostToDevice);
  hipMemcpy(Min, minimum, sizeof(float) * n, hipMemcpyHostToDevice);
  hipMemcpy(Index, index, sizeof(int) * n, hipMemcpyHostToDevice);
  hipMemcpy(Array, array, sizeof(float) * n, hipMemcpyHostToDevice);
  hipMemcpy(Path, path, sizeof(int) * n, hipMemcpyHostToDevice);
  size_t globalsize[1];
  globalsize[0] = 452;
  size_t localsize[1];
  localsize[0] = 113;

  clock_t tic = clock();

  hipLaunchKernelGGL(dijkstragpu, dim3(*(globalsize)),
                     dim3(*(localsize)), 0, 0, (float *)Matrix,
                     (float *)Visited, (float *)Min, (int *)Index,
                     (float *)Array, (int *)Path, (NULL), (NULL));
  clock_t toc = clock();
  hipMemcpy(matrix, Matrix, sizeof(float) * n * n,
            hipMemcpyDeviceToHost);
  /////////////// SECTION END ///////////////
```

Figure B.2: 'dijkstra' test case translation

```
diff --git a/assets/code/examples/difference/original.cpp b/assets/code/examples/difference/modified.cpp
index d9cec73..7dfca3d 100644
--- a/assets/code/examples/difference/original.cpp
+++ b/assets/code/examples/difference/modified.cpp
@@ -4,8 +4,6 @@
 #include "dijkstragpu_kernel.cl.hpp"
 ////////////////////////////////////////////////////////////

-#include "defs.h"
-#include <CL/cl.h>
 #include <math.h>
 #include <stdio.h>
 #include <stdlib.h>
@@ -14,12 +12,12 @@
 int main() {
   srand(time(NULL));
   int n = 452;
-  float *matrix = malloc(sizeof(float) * n * n);
-  float *visited = malloc(sizeof(float) * n);
-  float *minimum = malloc(sizeof(float) * n);
-  int *index = malloc(sizeof(int) * n);
-  int *path = malloc(sizeof(int) * n);
-  float *array = malloc(sizeof(float) * n);
+  float *matrix = (float *)malloc(sizeof(float) * n * n);
+  float *visited = (float *)malloc(sizeof(float) * n);
+  float *minimum = (float *)malloc(sizeof(float) * n);
+  int *index = (int *)malloc(sizeof(int) * n);
+  int *path = (int *)malloc(sizeof(int) * n);
+  float *array = (float *)malloc(sizeof(float) * n);
   int i, j, k, p, temp;
   int count = 0;
   FILE *fk = fopen("node452.txt", "r");
@@ -54,12 +52,8 @@ int main() {
     return 1;
   }

-  cl_context_properties properties[3];
   hipError_t err = hipSuccess;
   hipError_t event = hipSuccess;
-  properties[0] = CL_CONTEXT_PLATFORM;
-  properties[1] = (cl_context_properties)platform_id;
-  properties[2] = 0;

   void *Matrix, *Visited, *Min, *Index, *Array, *Path;
   err = hipMalloc((void **)&Matrix, sizeof(float) * n * n);
@@ -86,7 +80,7 @@ int main() {
   hipLaunchKernelGGL(dijkstragpu, dim3(*(globalsize)),
                      dim3(*(localsize)), 0, 0, (float *)Matrix,
                      (float *)Visited, (float *)Min, (int *)Index,
-                     (float *)Array, (int *)Path, *(NULL), *(NULL));
+                     (float *)Array, (int *)Path, (NULL), (NULL));
   clock_t toc = clock();
   hipMemcpy(matrix, Matrix, sizeof(float) * n * n,
             hipMemcpyDeviceToHost);
```

Figure B.3: 'dijkstra' diff file between original translation and necessary modifications

```
benmudd@benmudd-B360M-DS3H:~/testing$ clang test.c
test.c:2:3: error: use of undeclared identifier 'hello'
    2 |   hello = 2;
      |   ^
1 error generated.
benmudd@benmudd-B360M-DS3H:~/testing$ rustc test.rs
error[E0425]: cannot find value `hello` in this scope
  --> test.rs:12:5
   |
12 |     hello = 2;
   |     ^^^^^
   |
help: you might have meant to introduce a new binding
   |
12 |     let hello = 2;
   |     +++

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0425`.
```

Figure B.4: Existing compiler messages