

A Composable Workflow for Productive FPGA Computing via Whole-Program Analysis and Transformation (with Code Excerpts)

Paul Sathre
Dept. of CS
Virginia Tech
sath6220@cs.vt.edu

Ahmed Helal
Dept. of ECE
Virginia Tech
ammhelal@vt.edu

Wu Feng
Depts. of CS and ECE
Virginia Tech
feng@cs.vt.edu

Abstract

We present a composable workflow to enable highly-productive heterogeneous computing on FPGAs. The workflow consists of a trio of static analysis and transformation tools: (1) a *whole-program*, source-to-source translator to transform existing parallel code to OpenCL, (2) a set of OpenCL *kernel* linters, which target FPGAs to detect possible semantic errors and performance traps, and (3) a *whole-program* OpenCL linter to validate the host-to-device interface of OpenCL programs. The workflow promotes rapid realization of heterogeneous parallel code across a multitude of heterogeneous computing environments, particularly FPGAs, by providing complementary tools for automatic CUDA-to-OpenCL translation and compile-time OpenCL validation in advance of very expensive compilation, placement, and routing on FPGAs. The proposed tools perform *whole-program* analysis and transformation to tackle real-world, large-scale parallel applications. The efficacy of the workflow tools is demonstrated via a representative translation and analysis of a sizable CUDA finite automata processing engine as well as the analysis and validation of an additional 96 OpenCL benchmarks.

1 Introduction

Heterogeneous computing has become a frontrunner in the computational race for performance and power efficiency. FPGAs have gained increasing attention in the general-purpose computing community due to the emergence of high-level synthesis (HLS) tools based on OpenCL [17, 18]. Ideally, heterogeneous code should be "write-once, run-anywhere" regardless of the target parallel accelerator. In practice, however, productivity is limited by the community fragmentation between programming languages and the sunk cost in the existing heterogeneous code implemented in vendor-specific languages such as CUDA [20]. Thus, there is a need for automated tools to assist in the transition to the heterogeneous platforms accessible via the vendor-agnostic OpenCL language, which would allow devices to be procured on the basis of performance, power efficiency, and cost, rather than language compatibility only. Further, the FPGA is a new platform for many software developers, and such productivity tools can reduce the learning curve by providing automated translation to OpenCL and advisories about potential semantic and performance pitfalls.

The translation between programming languages should be done statically at the source-code level to support software maintainability, i.e., code modifications, adaptations, and extensions on the new target platforms. To automate such a daunting task, researchers have created several source-to-source translators with a specific focus on CUDA-to-OpenCL translation [9, 12, 15, 21], due to the

wealth of existing CUDA code that may benefit from the accessibility to heterogeneous platforms such as FPGAs. However, these tools work on a single translation unit (TU) at a time, which makes it impossible to propagate code transformations across TUs and limits their ability to holistically translate large-scale parallel codes.

Real-world applications follow a modular design methodology that separates the application's functionality into multiple software components (modules) to improve software maintainability and to promote code reuse. Thus, there is a compelling need for tools that examine applications holistically, spanning TUs to observe possible inconsistencies and effect wide-reaching global transformations.

As such, this paper presents a workflow of automated tools that supports whole-program analysis and transformation both within and across TUs for productive OpenCL programming on heterogeneous parallel systems with FPGA, GPU, and/or CPU. Whole-program analysis and transformation (e.g., refactoring, translation, etc.) is difficult in the context of separately compiled and linked languages (C/C++) and derivatives (CUDA and OpenCL) because of the traditionally hard delineation between TUs. However, by reducing this general cross-TU problem to the bounded instances of source-to-source translation between semantically similar languages and language- and platform-specific linter analyses, robust *cross-TU* tools can be realized using a unified framework that focuses on the interface between TUs.

In particular, we make the following research contributions:

- A *workflow of automated, composable tools to accelerate the development process of heterogeneous parallel code*. Specifically, CU2CL-MAST is a whole-program source-to-source translator that holistically transforms existing heterogeneous code from the vendor-specific CUDA to the vendor-agnostic OpenCL. FLOCL and FLOCL-MAST are intra- and inter-TU static code analyzers (i.e., linters), respectively, that detect possible semantic errors, runtime failures, and performance traps before investing time in expensive compilation and debugging/profiling. (§2 and §3).
- A *case study of transforming a finite automata code (iNFAnt) from CUDA to OpenCL via CU2CL-MAST and validation of the resulting code with the FLOCL and FLOCL-MAST linters*, thus enabling iNFAnt to run on other accelerators (e.g., FPGAs and AMD GPUs) in addition to NVIDIA GPUs (§4).
- A *suite of linter tools to identify potential performance issues and semantic faults in OpenCL codes*. Specifically, our proposed intra-TU tools identify over 153 potential kernel performance and semantic faults in five common OpenCL benchmark suites. In addition, our inter-TU linter detects inconsistent kernel calls between statically-compiled host code and JIT-compiled device code, which can result in undefined run-time behavior (§4).

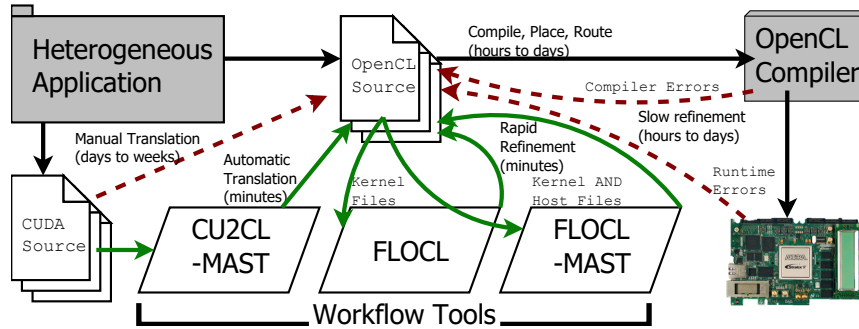


Figure 1. The proposed whole-program transformation and analysis workflow which accepts OpenCL or CUDA code (thanks to automated translation) and assists in rapid iterative refinement of device kernels and host runtime via linter advisories before ever compiling and running the application on an OpenCL platform.

2 Whole-Program Workflow

Figure 1 provides a sketch of the proposed workflow and explains how the tools interact. As a preliminary step for users with CUDA code, CU2CL-MAST (§ 3.4) expands an existing CUDA-to-OpenCL translator to the whole-program scope to tackle large-scale applications and enables these users to leverage FPGA architectures. These translated OpenCL *kernel files* (or preexisting ones) are individually analyzed with FLOCL (§ 3.3.1) to detect OpenCL and FPGA-specific semantic and performance faults. Next, the entire OpenCL application (i.e., all host and device files) is further inspected with FLOCL-MAST (§ 3.3.2) to ensure the consistency of the separately-compiled host-to-device interface. Finally, after these rapid refinement stages at the source-code level, the long kernel compilation process is performed to execute the OpenCL application on the FPGA device.

3 Workflow Tools

Here we present our suite of workflow tools, including our MAST (multi-abstract syntax tree) framework, libTooling management, FLOCL (FPGA linters for OpenCL), FLOCL-MAST, and CU2CL-MAST.

3.1 Multi-AST Framework

Figure 2 shows the typical uses of C, C++, and other parallel C-like languages (e.g., CUDA and OpenCL), where the source files (i.e., translation units or TUs) are separately compiled into individual object files, and then linked into a single binary/executable file. These TUs are only aware of each other based on explicit interfaces (i.e., declarations) provided by the programmer via shared header files. While the division of a software program into multiple TUs improves software maintainability, it complicates the design of any tool that operates on an entire application. Thus, we create a framework for transformation and analysis across multiple TUs that traverses the abstract syntax tree (ASTs) of each source file.

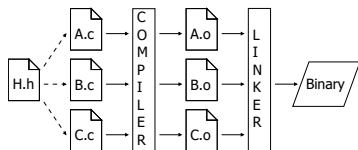


Figure 2. Typical separate compilation of object files before linking together.

Figure 3 shows the proposed framework that enables the creation of source-level inter-TU tools by providing several critical components. First, it leverages the Clang [6] compiler frontend to parse individual source files and translate them into navigable

ASTs for use in later stages. Second, the multi-AST framework uses the Clang *libTooling* library to control and modify the behavior of the compiler frontend, providing fine-grained control over the processing of individual TUs. Third, the framework relies on custom intra-TU logic to perform local analysis and transformation and to record interesting source features for later inter-TU stages. Finally, the inter-TU stage relies on the minimal shared interface between link units to connect across ASTs. The following subsections detail the different stages and compiler components that produce the multi-AST transformation and analysis tools.

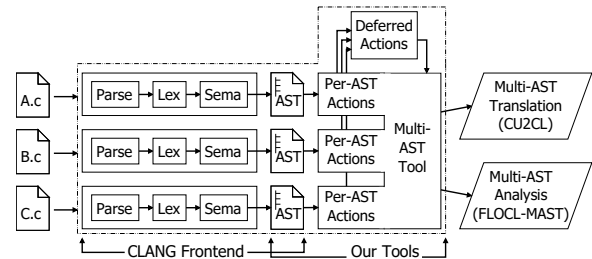


Figure 3. Multi-AST tools coordinate across invocations of the Clang Frontend to provide whole-program translation or analysis.

3.2 Clang libTooling Management

The Clang libTooling library provides the driver that translates source files into ASTs along with a convenient machinery to manage compilation tools via the invocation and processing of FrontendActions on source code. Algorithm 1 provides a skeleton of the execution phases of our multi-AST tools. The cross-TU tools customize the frontend behavior on a per-TU basis (i.e., to provide different compilation options for OpenCL device and host files) via pre-frontend callbacks. Custom FrontendActions implement the tool-specific intra-TU processing and are detailed further in subsequent sections. By having a tool scope *above* individual invocations of the frontend, the tool itself can maintain persistent data structures in memory across frontend invocations, which is critical to inter-TU analysis.

3.3 FLOCL and FLOCL-MAST

FLOCL provides OpenCL- and FPGA-specific linter passes to improve programmer productivity in developing OpenCL applications for FPGAs. The intra-TU portion is implemented as a series of modules for the *clang-tidy* tool. However, many of the complexities of OpenCL development arise because the OpenCL specification necessitates separate compilation of the host and device codes to support portability across OpenCL platforms. Therefore, the inter-TU FLOCL-MAST is devised as a companion tool to detect

```

1 struct misaligned {
2 char a;
3 double b;
4 char c;
5 };
6
7 struct packed_and_aligned {
8 char a;
9 double b;
10 char c;
11 } __attribute__((packed))
12   __attribute__((aligned(16)));

```

```

1 Finder->addMatcher(
2   recordDecl(
3     isStruct(),
4     ).bind("struct"), this);

```

```

1 Struct = Result.Nodes.getNodeAs<RecordDecl>("struct");
2 //Sum the bit width of all fields in the record
3 for each field in the record { minWidth += fieldWidth }
4 //Compute the minimum efficient member alignment (largest field member)
5 //Compute the minimum allowable width (even multiple of minAlignment <= sumWidth)
6 if (currWidth > minWidth) {
7   Emit "poorly packed" diagnostic
8 }
9 //Compute the minimum efficient struct alignment
10 // (smallest power of two up to the device's load width >= minWidth)
11 if (minAlign != currAlign) {
12   Emit "poorly aligned" diagnostic
13 }

```

(a) Example of a misaligned and unpacked struct that would use 24 bytes (aligned to 8 bytes) but only needs 10 as well as a packed and aligned struct that uses 16 bytes.

(b) The ASTMatcher used to find inefficient structs that may be misaligned or poorly packed.

(c) Skeleton of the Match Callback used to filter out only problematic structs.

Figure 4. Example code for the inefficient struct alignment and packing check.

Algorithm 1 Skeleton of multi-AST Tool execution.

```

START
Option processing
Pre-tool actions
for each source file from command line do
  Invoke Clang Frontend
  Pre-Frontend Callback Actions
  Frontend Parsing, Lexing, Semantic Analysis
  Post-Frontend, Intra-TU actions, Record deferred work
end for
Multi-AST Actions (consume deferred work)
END

```

inconsistencies in the separately-compiled interfaces between the OpenCL host and device codes.

3.3.1 FLOCL

The *clang-tidy* tool provides a robust framework for building linters that target C-like languages, and already supports a number of common linting passes (such as refactoring deprecated API usages, identifying dead code, etc.). However, general OpenCL kernels as well as FPGA-specific kernels require unique linting passes, due to the special C99 kernel dialect and platform restrictions, respectively. Thus, new linter modules for OpenCL and FPGA (i.e., “FLOCL”) are devised.

Modules to clang-tidy are written using the Clang *ASTMatchers* interface, which provides a way to target specific abstract syntax sub-trees (Sub-ASTs) that are defined in terms of node types and their relationships, and allow easy capture of snippets of structured code. Once matches are found, a user-defined callback is triggered to analyze each match. Hence, each FLOCL check is implemented as a matcher for a specific OpenCL code construct and a callback to diagnose it. These checks typically implement guidelines or restrictions from the OpenCL standard, OpenCL SDK documentations, and/or literature. A synthetic code example is constructed to exhibit the behavior, and then the AST representation of the code is manually examined for a matchable Sub-AST. A matcher to catch the symptomatic Sub-ASTs is designed, and then a callback provides further refinement and diagnostics.

We currently provide device checks for 1) inefficiently aligned/packed structs, 2) barriers in single-threaded kernels, 3) possibly-unreachable barriers inside conditionals, and 4) ID-dependent backward branching. These checks are derived from restrictions in the OpenCL specification and the Intel/Altera FPGA best practices. Additionally, a technique was developed to track variable assignments to ensure that a given variable never held a thread-dependent value, which is necessary to significantly reduce the false positive rate of checks (3) and (4).

1) Inefficient struct alignment and packing Ensuring efficient device memory access can improve the performance of an OpenCL kernel. When targeting Intel FPGAs, a minimum memory alignment of 4 bytes is desired and an optimal 128-bytes alignment is suggested [11]. Users who are unfamiliar with OpenCL may not be aware of the importance of memory alignment and packing, or the exposed attributes used to override the compiler defaults. Therefore, a linter is implemented to identify structs that do not use optimal packing and alignment, and suggest appropriate attributes for the developer to add. Figure 4 provides an example of a struct that could benefit from both explicit packing and alignment attributes, and shows a skeleton of the linter used to identify it.

The linter’s ASTMatcher detects every struct identified in the device code. Next, the match callback examines the default packing and alignment inferred by Clang, and computes the minimum efficient size and alignment based on the size of individual struct elements. Specifically, it computes the minimum size of the struct as if the element are stored contiguously, then computes the minimum alignment sufficient to fit the contiguous elements. If the Clang-generated size is larger than this computed minimum size, a diagnostic is emitted advising the use of a packed attribute. If the minimum computed alignment differs from the Clang-generated alignment, a similar diagnostic is emitted suggesting the proper alignment attribute to use. (Since we are targeting Intel FPGA devices, this value is bounded by a maximum 128-byte alignment.)

2) Single-work item barriers The Intel FPGA OpenCL platform detects if an OpenCL kernel is suited for execution across multiple work items (*NDRange*), or a single work item (*SWI*). This detection relies on calls to runtime APIs and/or compiler attributes, and also varies with the version of the device compiler. Older versions required calls to either `get_global_id` or `get_local_id` to be inferred as *NDRange*, whereas new versions trigger on additional functions [1, 11]. (Fig. 5a provides a trivial example of a *SWI* kernel and an *NDRange* kernel according to the original compiler.) If the user intends the kernel for *SWI* execution, any barrier synchronization can instead be relaxed to a `mem_fence`. Consequently, a linter was developed to address the presence of barriers in potential *SWI* kernels. The matcher shown in Fig. 5b simply detects any OpenCL kernel function with barriers, but no calls to a thread ID function. (Specifically the matcher binds on any function that 1) has the OpenCL `kernel` attribute, 2) calls either the OpenCL 1.X or 2.X work-group barrier function, 3) unless — logical NOT — it calls one of the ID functions. It also binds on the barrier call itself. A separate matcher will be generated for the modern compiler to support the additional ID functions.) The callback then provides a custom message, based on the target compiler version, to indicate

```

1 void __kernel singleWI(__global int * foo, int size) {
2   for (int j = 0; j < 256; j++) {
3     for (int i = 256; i < size; i+= 256) {
4       foo[j] += foo[j+i];
5     }
6   }
7   work_group_barrier(CLK_GLOBAL_MEM_FENCE);
8   for (int i = 1; i < 256; i++) {
9     foo[0] += foo[i];
10  }
11 }
12
13 void __kernel multiWI(__global int * foo, int size) {
14   int tid = get_global_id(0);
15   for (int i = get_global_size(0); i < size; i+= get_global_size(0)) {
16     foo[tid] += foo[tid + i];
17   }
18   work_group_barrier(CLK_GLOBAL_MEM_FENCE);
19   if (tid = 0) {
20     for (int i = 1; i < get_global_size(0); i++) {
21       foo[0] += foo[i];
22     }
23   }
24 }

```

(a) Example of a single work item kernel with a barrier (invalid) and a multiple work item kernel (that calls an ID function) with a barrier.

```

1 Finder->addMatcher(
2   //Find function declarations
3   functionDecl(allOf(
4     //That are OpenCL kernels
5     hasAttr(Attr::Kind::OpenCLKernel),
6     //And call a barrier function (either 1.x or 2.x version)
7     forEachDescendant(callExpr(callee(
8       functionDecl(anyOf(
9         hasName(" barrier "),
10        hasName(" work_group_barrier ")
11      ))
12    )), bind(" barrier ")),
13    //But do not call an ID function
14    unless(hasDescendant(callExpr(callee(
15      functionDecl(anyOf(
16        hasName(" get_global_id "),
17        hasName(" get_local_id ")
18      ))
19    ))))
20  )), bind(" function " ), this);

```

(b) The matcher used to find all OpenCL kernel functions that call a barrier, but do not call an ID function

```

1 MatchedDecl = Result.Nodes.getNodeAs<FunctionDecl>(" function ");
2 MatchedBarrier = Result.Nodes.getNodeAs<CallExpr>(" barrier ");
3 Emit "barrier in single work item kernel" diagnostic

```

(c) The simple Match Callback doesn't have to do any filtering, just emit a diagnostic since only invalid kernels are matched.

Figure 5. Example detection of a single-work-item kernel with a barrier.

that either the barrier will cause a compilation error or a `mem_fence` may be more appropriate for performance.

Thread-dependency tracking Several performance and semantic issues result from divergent threads within the OpenCL kernel. When threads take separate paths, performance suffers due to serialization and/or generation of less efficient FPGA hardware. Further, such divergence can result in deadlocks when barrier semantics are required. Therefore, a helper linter is developed to track variables which may contain thread-dependent values such that conditionals referencing them are flagged.

The helper linter detects all variable declarations within a kernel. When a variable is assigned the return value from a thread ID function, it is marked as *thread-dependent* by the match callback. Further, when a variable is assigned based on the value of some other variable(s), all members of the right hand side are checked for thread-dependency, and if such a dependency is found, the assignee is marked. Appendix Figure 10a shows the three matchers necessary to catch 1) declarations or assignments that directly call an ID function, 2) variable declarations that initialize to the value of some other variable or struct member, and 3) all assignments that reference a variable or struct member on the right hand side. Appendix Figure 10b demonstrates the callback portion that records

direct assignments (1), and propagates assignments from other ID-dependent references (2) and (3). Since Matches are performed consistent with a preorder traversal of the AST (and thus callbacks triggered in the same order), in almost all cases the dependency of a variable will be resolvable by the time it is matched — since use must occur after definition.¹ This particular tracking design is greedy and once a variable is flagged as ID-dependent, it cannot be unflagged; as a static analysis the approach is limited to what values the variable *may* contain, but without runtime information would have limited ability to definitively ascertain that such dependency was overwritten by all threads. This helper is used by the next two linters to determine if conditional expressions have a risk of being thread-dependent.

3) Possibly-unreachable barriers A barrier is a common and often necessary synchronization mechanism; however, it is also a potential source of deadlock errors that can be hard to debug at runtime. (On FPGA platforms with long kernel hardware generation time, the cost to resolve such a deadlock is further exacerbated.) A source of barrier deadlocks is the unintentional violation of the OpenCL specification by placing a barrier within a conditional region that is not reliably executed by all threads in a work-group. For example, Appendix Figure 11a shows two trivial for loops containing barriers that are executed an ID-dependent number of times, and will thus deadlock.

The helper linter for thread-dependency tracking, discussed above, is used to flag the “risky” conditional expressions — ones that either call a barrier directly or reference a variable or struct member. The rest of the matcher is otherwise fairly simple; it looks for any of the five primitive branch types (`if/else`, `switch/case`, `do/while`, `while` and `for`) that both contain a risky conditional expression and a call to a barrier function. Appendix Figure 11b provides the code for the branch matcher. The match callback (App. Fig. 11c) identifies what type of branch is present (for emitting a precise diagnostic), whether the conditional is in fact thread-dependent, and if so, whether it is due to a thread ID function call or an ID-dependent variable. However, barriers can still be used safely (if inefficiently) in a thread-dependent manner, if all paths reliably encounter the barrier. ID-dependency tracking provides false-positive reduction but alone does not address such valid in-branch barriers.

To address this case, the linter includes a refinement step to reduce false-positives even further in `if/else` and `switch` statements, where all possible paths execute the same number of barriers. Appendix Figure 12 provides the code that handles this refinement, which is structured similarly for both types of branches but accounts for the difference in semantics. (For example `switch` cases are terminated by breaks, but may run into one another without a break, and all possible entry points must be evaluated. `if/elses` have no terminator statement, but have well-defined “then” statements — usually compound.) Effectively both versions assemble a list of references to every possible entry point to their respective branches, and ensure an else or default case is present *otherwise it cannot statically guarantee that some threads don't skip the branch entirely*. Once this list is assembled, a simple recursive preorder traversal flattens the entire AST sub-tree for the branching structure,

¹The only corner case identified thus far is if a backward branch such as a loop or goto assigns a thread-dependent value to a previously-independent variable *after* a loop conditional that references the variable, but before the jump back to the start of the loop, i.e. in the loop body itself, but this is seldom performed in practice.

so that it may easily be traversed. Then this flattened tree is iterated from top to bottom to count barriers in each branch, skipping over most “uninteresting” AST nodes. When the beginning of a branch is encountered a barrier counter is initialized to zero. When the end of a branch is reached (either by encountering a break or by encountering the start of the next branch), the total number of barriers is checked. The refinement will abort to diagnostics if 1) the branch has zero barriers, since the matcher will only have matched if at least one branch has a barrier, 2) the branch has a different number of barriers than a previously-counted branch, or 3) a switch case has a non-zero number of barriers and proceeds directly into another case without breaking, since it is *guaranteed to encounter more barriers than the subsequent case*. Each time a function call is encountered, it is counted if and only if it is a barrier, otherwise, the refinement aborts since it cannot easily traverse the CFG and determine whether the called function contains a barrier. Finally, any time a jump or nested conditional is found, the refinement aborts.

4) ID-dependent backward branches In the context of FPGAs, OpenCL kernel instructions are generated as a hardware pipeline. For efficiency reasons, the Intel compiler attempts to collapse branch statements into a single bit indicating if a functional unit is active [11]. This is relatively easy for “forward” branches (without loops), resulting in flat control structures. However, looped (i.e. “backward”) branches are more difficult to implement which can significantly reduce performance, and thus should be avoided. While some algorithms may require backward branches, many can be removed with algorithmic refactoring. Therefore, a linter has been designed to recognize potential ID-dependent backward branching to advise users to consider refactoring their kernels.

ID-dependent backward branches, by definition, cannot be resolved and optimized at compile time, and thus result in a more complex hardware logic. Therefore, the linter needs to make use of the thread-dependency tracker we have previously detailed. As before, the linter matches on any loop with a risky conditional expression, and then the callback determines if the conditional may be thread-dependent and emits a diagnostic. Figure 6 outlines the otherwise simple remainder of the matcher and callback.

3.3.2 FLOCL-MAST

FLOCL-MAST expands on FLOCL by providing a standalone multi-AST framework for developing new linters that require simultaneous access to multiple TUs. An important example is linters that work across the host and device TUs to ensure semantic consistency. In large-scale applications, it can be difficult to ensure that modifications to the host-device interface are applied consistently in the whole program, which can lead to burdensome runtime bugs. For example, if the type, number, or position of kernel parameters changed, the application will crash and OpenCL’s runtime can report an error (granted that proper error checking is implemented in the code), wasting valuable compilation and execution time. Thus, the first Multi-AST linter for OpenCL implements a check to ensure the semantic consistency of each of the arguments provided to a device kernel invocation.

FLOCL-MAST operates in two distinct phases, an intra-TU phase that records information from each device or host AST, and an inter-TU phase that validates this information and reports potential problems. Figure 7 provides a conceptual overview of the flow from

```

1 // Omitted ID-dependency matchers
2
3 // Matcher for detecting branch statements inside of loops
4 // which only binds if they condition expression either calls an ID
5 // function or references a variable
6 // Sub-matcher to check if an expression contains an ID call or
7 // references a variable that may be ID-dependent
8 const auto COND_EXPR =
9   expr (anyOf(
10     hasDescendant (callExpr (
11       callee (functionDecl (anyOf(
12         hasName (" get_global_id "),
13         hasName (" get_local_id ")
14       )))
15     ).bind (" id_call ")),
16     hasDescendant (stmt (anyOf(
17       declRefExpr (to (varDecl ())),
18       memberExpr (member (fieldDecl ()))
19     )))
20   )).bind (" cond_expr ");
21 // Matcher that grabs any branch statements that have a potential
22 // ID-dependency in the conditional expression and are inside loops
23 Finder -> addMatcher (
24   stmt (anyOf(
25     forStmt (
26       hasCondition (COND_EXPR),
27     doStmt (
28       hasCondition (COND_EXPR),
29     whileStmt (
30       hasCondition (COND_EXPR)
31     )).bind (" backward_branch ")),
32   this);

```

(a) The ASTMatcher used to find all loop conditionals that need to be checked for ID-dependency.

```

1 // Omitted thread-dependency tracking code
2
3 // The second part of the callback just checks matched branches for
4 // an ID-dependent conditional expression
5 const auto *CondExpr = Result.Nodes.getNodeAs<Expr> (" cond_expr ");
6 const auto *IDCall = Result.Nodes.getNodeAs<CallExpr> (" id_call ");
7 const auto *Loop = Result.Nodes.getNodeAs<Stmt> (" backward_branch ");
8 int loop_type = -1;
9 if (Loop) {
10   if (isa<DoStmt>(Loop)) {loop_type = 0;}
11   else if (isa<WhileStmt>(Loop)) {loop_type = 1;}
12   else if (isa<ForStmt>(Loop)) {loop_type = 2;}
13 }
14 if (CondExpr) {
15   if (IDCall) {
16     // It calls one of the ID functions directly
17     // Emit a diagnostic that the conditional calls an ID function
18   } else {
19     // It has some DeclRefExpr(s), check for ID-dependency
20     const auto *retExpr = hasIDDepDeclRef(CondExpr);
21     const auto *retMemberExpr = hasIDDepMember(CondExpr);
22     if (retDeclExpr || retMemberExpr) {
23       // Warn that an ID-dependent variable is referenced
24     }
25   }
26 }

```

(b) The latter portion of the backward branch callback uses the ID-dependency information computed earlier to check the conditional expressions from branches that are inside loops.

Figure 6. Example code for checking whether backward branches (loops) may have inefficient ID-dependent conditional expressions.

Algorithm 2 Skeleton of FLOCL-MAST execution stages.

```

START
for each source file from command line do
  if Kernel File then
    Record Kernel Prototypes (Figs. 13e & 13f)
  else
    Record cl_kernel Decls (Figs. 13b & 13c)
    Record clCreateKernel Calls (Figs. 13b & 13c)
    Record clSetKernelArg Calls (Figs. 13b & 13c)
    Record Kernel Launches (Figs. 13b & 13c)
  end if
  Retain AST, SourceManager, etc.
end for
Bind cl_kernels to Device Prototypes (Fig. 14a)
Cluster clSetKernelArg calls to Launches (Fig. 14b)
Bind Launch Clusters to Prototypes (Fig. 14c)
Check Each Argument Against Prototype (Figs. 15a & 15b)
END

```

source code to diagnostics. Algorithm 2 shows the sequence of

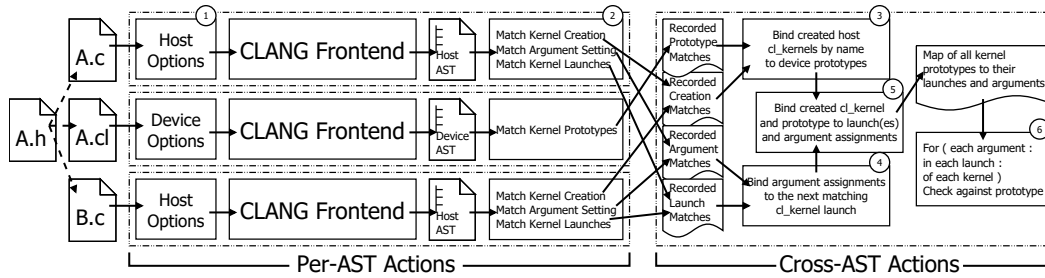


Figure 7. Conceptual overview of FLOCL-MAST execution. Host and device files are recognized by file type (1) and each go through a separate instance of the Clang Frontend to be processed to ASTs. After the ASTs are generated, they each use AST Matchers and callbacks to record relevant AST nodes for the inter-TU stages (2). After all per-AST actions are completed the cross-AST portion of the tool iterates over all the recorded information to pair kernel function prototypes to their host-side representation (3) and uses (4) & (5), then inspects them all for type consistency between arguments and parameters (6).

linter steps, and provides pointers to the Appendix source code listings for each processing step. Similar to FLOCL, the intra-TU recording phase is implemented as a set of ASTMatchers and callbacks which identify the code components of each stage of the OpenCL kernel invocation. It records `__kernel` function declarations from device ASTs, and `clCreateKernel`, `clSetKernelArg`, and `clEnqueueNDRangeKernel` calls from host ASTs. Further, each time a `cl_kernel` is declared, the storage space is checked; if it is global (and possibly extern) it is stored in a special inter-TU data structure that maps global externs to their real definition (i.e. the object file that will actually store the variable).

After the intra-TU analysis is completed on all device and host ASTs, the multi-AST portion of the tool combines information from each of the kernel invocation stages. First, each `clCreateKernel` call is checked for an exact match of its kernel name argument to a prototype from a device AST. When such a match is found, the device declaration is bound to the assigned `cl_kernel`. Next, all `clSetKernelArg` calls are “clustered” to the lexically-next kernel launch with a matching `cl_kernel`. Lexically-next refers to the next launch encountered when scanning the source code from beginning to end, since kernel invocations are usually preceded closely by the argument specification. Following argument clustering, the next stage iterates over every kernel launch and attaches them to a prototype with a matching `cl_kernel` object, if one exists.

Finally, each kernel launch is compared to the device prototype to ensure that 1) all arguments are provided and 2) each provided argument has a compatible type to that of the declared parameter in the same position. This comparison must account for the difference between the pass-by-reference semantics of `clSetKernelArg` and the pass-by-value semantics of the kernel. Hence, the void pointer-to-scalar variables in the host code must be resolved to a non-void qualified type, if possible. (Typically the address of variables is taken and cast to void within the call, which is easy to resolve.) Pointers on the device must be assigned as either `cl_mems` (for global and constant memory) or a NULL pointer with a custom size argument for local memory. The former currently uses a wildcard match of any approved pointer to a `cl_mem`, and the latter is currently unsupported; both will be refined as future work.

This inter-TU linter is useful as a demonstration of the complexity of multi-AST analysis in comparison to the intra-AST analysis used by basic FLOCL. While both FLOCL and FLOCL-MAT use the same underlying ASTMatcher and callback technology, the complexity of the analysis dramatically increases once the tool works across the ASTs of different translation units.

3.4 CU2CL-MAST

CU2CL [9] is an automated CUDA-to-OpenCL translator that performs *AST-driven, string-based* translation, i.e., CU2CL walks the AST generated by Clang to identify CUDA expressions and then translates the code by replacing the relevant text directly. This has the benefit of preserving preprocessor directives, comments, and format in the generated OpenCL source files. However, CU2CL is restricted to a single translation unit – it cannot deal with the extra complexity of a multi-file application without tedious manual pre- and post-processing (for example, to merge generated OpenCL boilerplate). Therefore, we provide a whole-program multi-AST expansion of CU2CL, dubbed CU2CL-MAST (Multi-AST).

The implementation of CU2CL-MAST consisted of four phases. First, a *libTooling* interface was built to bootstrap and contain multiple instances of the existing intra-TU translation functionality, as detailed in § 3.2. Second, to ensure completeness of translation, an implicit ban on processing non-`.cuh` header files was removed. Third, intra-TU boilerplate generation was modified such that common elements are produced by the inter-TU layer and generated in a shared set of `cu2cl_util.c/h/cl` utility files. Furthermore, within each TU the inter-TU layer generates extern declarations for `cl_programs` and `cl_kernels` originating in other TUs. The fourth and most novel stage tackled the longstanding incongruity between OpenCL and CUDA representations of pointers to device-side buffers: OpenCL stores the host-side pointer to a device buffer in a `cl_mem` object, whereas CUDA allows storage of such pointers in any arbitrary pointer type.

The original intra-TU CU2CL performs immediate depth-first translation of the device buffer used in a `cudaMalloc` expression by simply navigating to the buffer’s declaration and changing the type [9]. However, as each AST is only walked once, when this variable is aliased – either by pointer assignment or through a function call – propagation to all possible aliases cannot be guaranteed, causing a type inconsistency in other code regions that are not aware of the translation. (Figure 8a demonstrates a simple case of aliasing which confounds the original translator.) Thus, either the variable is not translated and instead explicitly casted to a standard pointer variable (as in [12]), or the translation must be fully propagated to all affected regions. The latter is substantially harder problem as potentially extends across scopes and TUs. While the casting solution is expedient, it creates a software maintainability issue in a broader codebase as the true type of the casted `cl_mem` variables is only apparent when used in OpenCL calls. Therefore, to ensure propagation of translated device pointers all translation

```

1 =====Contents of a.h=====
2 void create(float **, size_t);
3 void launch(float *, float *);
4 =====Contents of a.cu=====
5 #include "a.h"
6 __global__ void fooKern(float * inBuf, float * outBuf) {
7     outBuf[threadIdx.x] = inBuf[threadIdx.x];
8 }
9 void create(float ** buffer, size_t size) {
10     cudaMalloc(buffer, size);
11 }
12 void launch(float * inBuf, float * outBuf) {
13     fooKern<<<1,256>>(inBuf, outBuf);
14 }
15 =====Contents of b.c=====
16 #include "a.h"
17 float *A, *B;
18 int main(int argc, const char * argv[]) {
19     int condition = atoi(argv[1]);
20     create(&A, sizeof(float)*256); create(&B, sizeof(float)*256);
21     float *inBuf, *outBuf;
22     if (condition) { inBuf = A, outBuf = B; }
23     else { inBuf = B, outBuf = A; }
24     launch(inBuf, outBuf);
25 }

```

(a) CUDA device pointers used as arguments and aliased. The initial translation is triggered by the `cudaMalloc` call on line 10.

```

1 =====Contents of a.h-cl.h=====
2 void create(cl_mem *, size_t);
3 void launch(cl_mem, cl_mem);
4 =====Contents of a.cu-cl.cpp=====
5 ... (omitted boilerplate)
6 #include "a.h-cl.h"
7 void create(cl_mem * buffer, size_t size) {
8     *buffer = clCreateBuffer(..., size, ...);
9 }
10 void launch(cl_mem inBuf, cl_mem outBuf) {
11     clSetKernelArg(__cu2cl_Kernel_fooKern, 0, sizeof(cl_mem), &inBuf);
12     clSetKernelArg(__cu2cl_Kernel_fooKern, 1, sizeof(cl_mem), &outBuf);
13     localWorkSize[0] = 256;
14     globalWorkSize[0] = (1)*localWorkSize[0];
15     clEnqueueNDRangeKernel(..., __cu2cl_Kernel_fooKern, ...);
16 }
17 =====Contents of b.c-cl.cpp=====
18 ... (omitted boilerplate)
19     #include "a.h-cl.h"
20     cl_mem A, B;
21     int main(int argc, const char * argv[]) {
22         __cu2cl_Init();
23         int condition = atoi(argv[1]);
24         create(&A, sizeof(float)*256); create(&B, sizeof(float)*256);
25         cl_mem inBuf, outBuf;
26         if (condition) { inBuf = A, outBuf = B; }
27         else { inBuf = B, outBuf = A; }
28         launch(inBuf, outBuf);
29         __cu2cl_Cleanup();
30 }

```

(b) Example of Multi-AST CU2CL’s fully-propagated `cl_mem` translation

Figure 8. Propagation of the `cl_mem` type by CU2CL-MAST. Without the Multi-AST expansion only the declaration of “buffer” on line 9 in Fig. 8a is translated. With Multi-AST propagation, line 9 propagates horizontally to the forward declaration on line 2. By translating a header shared with `b.c`’s AST, the calls to create on line 20 propagate upward to the declarations of “A” and “B” on line 18. These propagate horizontally through the alias assignments in the `if/else` region on lines 22 and 23 to the declarations of “inBuf” and “outBuf” on line 21. Finally, translating `inBuf` and `outBuf` propagate downward through the call to “launch” on line 24 to launch’s forward declaration on line 3 and then horizontally to the definition on line 12.

was hoisted to the inter-TU layer which has access to the entirety of the codebase and can track aliasing use/def relationships.

Primarily, inter-TU translation relies on forward declarations from a shared header that resides on all ASTs and thus form a common anchor between them. Thus, we modified the intra-TU portion of CU2CL to defer all `cl_mem` translations, by storing them in an inter-TU list, and to generate a map from all declared variables and functions to their references such that the inter-TU layer can track their uses. Actual translation is then implemented as a consumer of the deferred translation list. While each translation is performed, the reference map is checked for necessary `cl_mem` translation propagation.

In particular, we address three cases in which a `cl_mem` type translation necessitates propagation to some other device pointer:

- Downward propagation: when a translated variable is a function argument, the type change must be extended **down** the control flow graph (CFG) to the callee’s parameters.
- Upward propagation: when a function parameter is translated, the type change must be propagated **up** the CFG to any variables supplied as arguments to that parameter.
- Horizontal propagation: when a function parameter is translated, it must be applied to any forward declaration in a shared header to be visible to other ASTs. Pointers aliased through assignment statements must ensure that both the left- and right-hand sides have the `cl_mem` type.

When a necessary propagation is found, the newly-affected variable is added to the translation list, if it is not already on the list. Once the list is consumed, all `cl_mem` translations are fully propagated. (Effectively, this performs a breadth-first propagation of translations up and down the control flow graph and across ASTs, starting at the initial translation site.) Figure 8 walks through a typical propagation cascade from a single initial translation site.

4 Case Studies

We demonstrate the efficacy of the proposed workflow using common OpenCL mini-apps and benchmarks as well as a CUDA finite automata processing engine. The following subsections detail the applications used to test each tool, and provide a discussion on the quality of their respective outputs — translated OpenCL for CU2CL-MAST, and linter warnings for FLOCL and FLOCL-MAST.

4.1 Target Workloads

4.1.1 iNFAnt

iNFAnt is a CUDA implementation of a non-deterministic finite automata (NFA) regex matching engine [3]. We used an optimized version of the algorithm, detailed in [23], as a case study for a CUDA-to-FPGA translation pipeline using both CU2CL-MAST and FLOCL/FLOCL-MAST. An NFA regex matching application was selected as it represents a common workload for FPGAs. In later sections, we discuss the improvement of CUDA-to-OpenCL translation using CU2CL-MAST, and analyze the feedback produced by FLOCL and FLOCL-MAST on the translated code.

4.1.2 Benchmarks

Five OpenCL benchmark suites were selected to cover a wide range of use cases in terms of both application domain and OpenCL development style. OpenDwarfs [8, 13], PolyBench-ACC [10], Rodinia [4, 5], and SHOC [7] come from a more traditional HPC and general-purpose GPU background. CHO [19] targets OpenCL-on-FPGA development and captures current limitations of offline-compiled FPGA development. Across the five suites, there are a total of 96 separate OpenCL applications.

4.2 CU2CL-MAST

4.2.1 Translation Analysis

To show how CU2CL-MAST simplified the translation of large-scale code with multiple TUs, such as optimized iNFAnt, we compare CU2CL-MAST’s translation to the original plugin-based, single-AST CU2CL (version 0.6.2b). (The original CU2CL had to be modified to ensure output files were generated despite possible errors.) CU2CL-MAST’s translation improvement was evidenced in three categories:

Improved header files processing As a prerequisite to multi-AST translations, CU2CL-MAST ensures that translations that occur within *any* file are faithfully reproduced as output. The original CU2CL translator took an overly conservative approach and limited translation of header files to non-system header files with either a `.cu` or `.cuh` extension, although many developers declare wrappers to CUDA kernel and utility functions in `.h`, `.hpp`, or other file types and *may pass device buffers through those functions*. By expanding translation to all non-system files regardless of file type, CU2CL-MAST performs translation in 14 additional header files of Optimized iNFAnt, which include portions of the interface that must have `c1_mems` propagated through them.

Multi-AST boilerplate generation CU2CL generates custom boilerplate for each AST, including many redundant features such as OpenCL emulation of CUDA functions like `cudaMemset`. Conversely, CU2CL-MAST defers the boilerplate generation to the inter-TU portion of the tool. As such, it can intelligently generate shared utility files that concisely provide all shared functionality. Furthermore, CU2CL-MAST coordinates all explicit initialization and finalization functionality from each TU, *eliminating the need for post-translation manual merging*. While Optimized iNFAnt uses a single kernel file, CU2CL-MAST generates the necessary extern declarations to make it visible program wide, which is necessary since creation and invocation happen in separate source files. In programs with more kernels from different source files, the value of this automatic propagation across source files is further increased.

c1_mem propagation Section 3.4 has already detailed the significance of ensuring propagation of type translations. The Optimized iNFAnt code uses wrapper functions around all CUDA allocation and deallocation functions. This necessitates `c1_mem` propagation, which CU2CL-MAST successfully provides.²

4.2.2 Translation Validation

The translated OpenCL application was validated by comparing its output against the the original CUDA application, when compiled with the built-in debugging and validation routines. The built-in test packet generator was used as synthetic network traffic, and the packaged “big-boy” transition graph was used to specify the finite automata. The OpenCL results were consistent with the original CUDA application; in particular, we successfully reproduced the original results using the translated OpenCL code on all tested OpenCL platforms (Nvidia K20Xm GPU, AMD S9150 GPU, Intel Arria10 FPGA).

4.3 FLOCL and FLOCL-MAST

4.3.1 FLOCL Results

The FLOCL linters produce warnings about possible semantic or performance faults when examining the 138 kernel files in the benchmarks. Each may have important performance issues or runtime problems that will only become apparent at runtime after

²A slight incongruence in the CU2CL and CU2CL-MAST implementation of `cudaMallocHost` was discovered. Rather than returning just the host buffer’s pointer, and internally managing the `c1_mem` as required to directly emulate the function, `__cu2c1_mallocHost` returns both the host and device pointers. A simple map (key: host pointer, value: device `c1_mem`) implementation is manually utilized to emulate the desired semantics. Automatic management of such host-allocated-and-mapped buffers remains for future work.

the developer has invested significant time in compilation, placement, and routing. The following is the detailed description of the warnings found:

- Possibly unreachable barrier: four kernel files in the benchmark suite have barriers which are thread-dependent and may result in deadlocks at runtime. Fig. 9 shows one of the hits in SHOC/SPMV.
- ID-dependent backwards branches: 90 instances of backward branches are observed in the benchmark suites, and another four occur in the optimized iNFAnt application.
- Inefficient struct alignment: in the benchmark suites, FLOCL identifies 59 opportunities to augment a struct with extra attributes for improved memory utilization. Specifically, 41 cases could benefit from an aligned attribute, and 18 from a packed attribute. An additional 2 structs from the Optimized iNFAnt application could benefit from both explicit packing and alignment.
- SWI-barrier: no instances of a barrier in a single-threaded kernel were found. (The benchmarks largely consist of applications that are explicit designed for multi-threaded NDRange execution, rather than single-thread execution.)

```

1 __kernel void
2 spmv_csr_vector_kernel(__global const FPTYPE * restrict val,
3                       __global const FPTYPE * restrict vec,
4                       __global const int * restrict cols,
5                       __global const int * restrict rowDelimiters,
6                       const int dim, const int vecWidth,
7                       __global FPTYPE * restrict out)
8 {
9     // Thread ID in block
10    int t = get_local_id(0);
11    // Thread ID within warp
12    int id = t & (vecWidth-1);
13    // One row per warp
14    int vecsPerBlock = get_local_size(0) / vecWidth;
15    int myRow = (get_group_id(0) * vecsPerBlock) + (t / vecWidth);
16    __local volatile FPTYPE partialSums[128];
17    partialSums[t] = 0;
18    if (myRow < dim)
19    {
20        // ... omitted partial sum loop
21        partialSums[t] = mySum;
22        barrier(CLK_LOCAL_MEM_FENCE);
23        // ... omitted reduction and write
24    }
25 }

```

Figure 9. Example of a possibly-unreachable barrier found in SHOC/SPMV. If a workgroup is configured such that some (but not all) work items’ `myRow` exceeds the `dim` parameter, the barrier after partial summation (line 22) will not be encountered by the entire workgroup. `myRow` is marked as thread-dependent due to the ID calls on lines 10 and 15.

4.3.2 FLOCL-MAST Results

We apply FLOCL-MAST to each OpenCL application, across the five benchmark suites, to detect any semantic inconsistency between the host and device code. As the benchmarks are largely production-grade and well-developed, 50 of the 97 applications found no inconsistencies between device and host argument types. The host-device validation linter is designed for in-development applications, where the interface may be in flux and more likely to exhibit inconsistencies. However, despite the overall quality of the benchmarks, FLOCL-MAST did raise warnings while validating the remaining 47 applications.

True Positives In a typical CPU-only compilation, the compiler emits warnings about mismatched types or missing arguments when calling user-defined functions it is aware of. Since OpenCL host and device codes are separately compiled using different compiler tool chains, neither tool chain is aware of the interface the other is using for invoking user-defined kernel functions. Therefore,

Table 1. Application behavior when linted with FLOCL-MAST

Benchmark Suite	Report True Positives	Report False Positives	Other Issues
CHO	0	2	0
OpenDwarfs	3	3	2
PolyBench-ACC	1	8	1
Rodinia	1	1	10
SHOC	5	7	7

FLOCL-MAST provides this same functionality across the host-to-device interface, and several examples are found in the benchmark suites.

- Possible loss of precision: nine applications change the size and/or signedness of an integer between host and device.
- PolyBench/Jacobi-2d-Imper has an unused wrapper function that calls two kernels without arguments. It should either be removed or documented as requiring arguments persisting from the other (correct) launches.

False Positives Generally, a linter should err on the side of caution when producing warnings. Currently FLOCL-MAST produces some false positive warnings about missing or mis-typed arguments, many of which can be resolved by respecting the equivalence of typedefs that refer to the same primitive type.

- Unrespected equivalent types: FLOCL-MAST currently uses the *declared type* to match arguments to parameters, but should be adapted to use the *primitive type* referred to by a possible typedef'd or dependent type. This causes the majority of false mismatches (17 out of 21 applications with false positives). Specifically, SHOC/BFS, Rodinia/Leukocyte, and CHO/{AES_fpga, AES} all mismatch to the `cl_int` OpenCL host runtime type; OpenDwarfs/TDM mismatches a typedef'd `ubyte` to an `unsigned char` on the device; OpenDwarfs/{TDM, CRC} mismatch a host `unsigned int` and a device-side `uint`; PolyBench/{Correlation, SYRK, 2mm, Gemver, gesummv, GEMM, SYR2K, Covariance} use a typedef inside a conditional compilation region for `DATA_TYPE`; and SHOC/{GEMM, MD and, S3d} all use a template-dependent type on the host which FLOCL-MAST should resolve (as it has all the compile-time template specialization information necessary to do so).
- Variable argument position: SHOC/BFS and OpenDwarfs/Kmeans, use an incremented integer to specify the argument position, rather than an integer literal. If the variable value cannot be statically deduced at compile time, the position cannot be determined for validation against the device prototype.
- SHOC/MD calls an iterative kernel in which the arguments are only set on the first launch, outside a loop, and then multiple subsequent calls invoke the kernel without reassigning the same arguments. In this case, both launches occur within the same function and FLOCL-MAST should be expanded to understand OpenCL's retained arguments; however, if the launches were to occur in separate functions, a static analysis tool would have limited ability to deduce what order they would be invoked at runtime, and thus whether any arguments were retained.
- SHOC/{Reduction, Scan, and Sort} use dynamic local memory allocation, which is reported as a type mismatch between a `NULL` pointer on the host, and the true `local` address-space pointer type on the device. FLOCL-MAST should be expanded to match such local memory regions to a host `NULL` pointer, but further

confirmation of the host-intended type would be limited to what can be heuristically deduced from the size argument to `clSetKernelArg` (such as through inspection of a possible `sizeof` operator).

Other Issues The remaining issues represent limitations of the static analysis approach when applied to a JIT-compiled (partially dynamically-compiled) language, or corner cases to improve in the tool's implementation.

- Dynamic kernel names and program strings: in all but the simplest runtime-invariant assignments, using dynamic string for device code or kernel name will be outside the scope of a static analysis tool or *offline FPGA compilation*. Eleven applications use some form of dynamic string value.
- Indirect `cl_kernel` references: eight applications alias their `cl_kernel` objects by passing them through functions or storing them within other data types. Some degenerate cases may be solvable with propagation techniques like CU2CL-MAST, but most are outside the scope of a static analysis tool, as following (possibly multiple) runtime-dependent `cl_kernels` through a CFG is more difficult than a static type change.

5 Related Work

Traditional whole-program analysis and transformation tools suffer from prohibitive run-time and memory cost due to the quadratic complexity of the required program representations such as Program Dependence Graph (PDG) [2, 16]. One approach for designing such tools is to construct the expensive program representations on demand using program slicing [2]. However, this demand-driven approach is limited to software debugging and comprehension and is not suitable for performing global source-code transformation or detecting inter-TU inconsistencies. Another approach for whole-program tools summarizes the critical program representations to reduce the super-linear run-time and memory complexity at the cost of less analysis precision, which hinders the ability of these tools to realize source-code translation and bug detection [22].

Therefore, this work introduces an open-source framework for whole-program analysis and transformation that utilizes the abstract syntax tree (AST) representation of software programs and leverages the Clang and LLVM infrastructure [6, 14]. Unlike other program representations, the size of the AST is linear in the size of the software program [2, 16], which makes it a practical representation for tackling several important problems. Specifically, we demonstrate the application of a novel multi-abstract syntax tree (multi-AST) approach to create tools that operate on entire programs composed of multiple separately compiled and linked C/C++-like source files.

6 Conclusion

In this work we have demonstrated a workflow for productive OpenCL programming and whole-program analysis on FPGA. The workflow is composed of three tools: a reconstructed Multi-AST variation of an existing CUDA-to-OpenCL translator, intra-TU linters for OpenCL and FPGA, and an inter-TU linter for validating the OpenCL host-to-device kernel launch interface. The entire workflow is demonstrated by translation and analysis of a research finite automata code, originally written in CUDA. Further analysis of the

linters is conducted on a suite of 96 OpenCL benchmarks, successfully identifying over 150 possible errors or performance faults, as well as opportunities to further refine the utilized heuristics.

York, NY, USA, Article 18, 10 pages. <https://doi.org/10.1145/2482767.2482791>

References

- [1] Altera. 2015. Altera FPGA SDK for OpenCL, Best practices guide. Online. (05.04 2015). https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
- [2] Darren C. Atkinson and William G. Griswold. 1996. The Design of Whole-program Analysis Tools. In *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*. IEEE Computer Society, Washington, DC, USA, 16–27. <http://dl.acm.org/citation.cfm?id=227726.227732>
- [3] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnT: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Comput. Commun. Rev.* 40, 5 (Oct. 2010), 20–26. <https://doi.org/10.1145/1880153.1880157>
- [4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [5] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. 1–11. <https://doi.org/10.1109/IISWC.2010.5650274>
- [6] Clang Clang. 2013. A C language family frontend for LLVM. (2013).
- [7] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 63–74. <https://doi.org/10.1145/1735688.1735702>
- [8] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. 2012. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. ACM, New York, NY, USA, 291–294. <https://doi.org/10.1145/2188286.2188341>
- [9] Mark Gardner, Paul Sathre, Wu chun Feng, and Gabriel Martinez. 2013. Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator. *Parallel Comput.* 39, 12 (2013), 769 – 786. <https://doi.org/10.1016/j.parco.2013.09.003> Programming models, systems software and tools for High-End Computing.
- [10] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. <https://doi.org/10.1109/InPar.2012.6339595>
- [11] Intel. 2017. Intel FPGA SDK for OpenCL, Best practices guide. Online. (12.08 2017). https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
- [12] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. 2015. Bridging OpenCL and CUDA: A Comparative Analysis and Translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 82, 12 pages. <https://doi.org/10.1145/2807591.2807621>
- [13] K. Krommydas, Wu-Chun Feng, M. Owaida, C. D. Antonopoulos, and N. Bellas. 2014. On the characterization of OpenCL dwarfs on fixed and reconfigurable platforms. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*. 153–160. <https://doi.org/10.1109/ASAP.2014.6868650>
- [14] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.
- [15] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. 2011. CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, 300–307.
- [16] S.S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers.
- [17] Aaftab Munshi. [n. d.]. The OpenCL Specification, 2008. *Chronos OpenCL Working Group* ([n. d.]).
- [18] Aaftab Munshi. 2009. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 1–314.
- [19] Geoffrey Ndu, Javier Navaridas, and Mikel Luján. 2015. CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators. In *Proceedings of the 3rd International Workshop on OpenCL (IWOC '15)*. ACM, New York, NY, USA, Article 10, 10 pages. <https://doi.org/10.1145/2791321.2791331>
- [20] CUDA Nvidia. 2007. Compute unified device architecture programming guide. (2007).
- [21] Paul Sathre, Mark Gardner, and Wu-chun Feng. 2012. Lost in translation: Challenges in automating cuda-to-opencl translation. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 89–96.
- [22] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis*. Technical Report. New York Univ. Comput. Sci. Dept.
- [23] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '13)*. ACM, New

A Additional Code Listings

```

1 //Sub-matcher to check if an expression calls an ID function
2 const auto TID_RHS =
3   expr(
4     hasDescendant(callExpr(
5       callee(functionDecl(anyOf(
6         hasName("get_global_id"),
7         hasName("get_local_id")
8       )))
9     ))
10 );
11 //Sub-matcher for any of the binary assignment operators
12 const auto ANY_ASSIGN =
13   anyOf(
14     hasOperatorName("="), hasOperatorName("!="),
15     hasOperatorName("/="), hasOperatorName("%="),
16     hasOperatorName("+="), hasOperatorName("-="),
17     hasOperatorName("< <="), hasOperatorName("> >="),
18     hasOperatorName("&="), hasOperatorName("^="),
19     hasOperatorName("|=")
20 );
21 //Matcher for any statement that either initializes a variable with
22 // an ID call, or assigns the results of an ID call to a variable
23 Finder->addMatcher(
24   compoundStmt(
25     //Bind on actual get_local/global_id calls
26     forEachDescendant(
27       stmt(anyOf(
28         //Declarations which initialize with an ID call
29         declStmt(hasDescendant(
30           varDecl(hasInitializer(TID_RHS)).bind("tid_dep_var")
31         )),
32         //Assignments that store an ID value
33         binaryOperator(allOf(
34           ANY_ASSIGN,
35           hasRHS(TID_RHS),
36           hasLHS(anyOf(
37             declRefExpr(to(
38               varDecl().bind("tid_dep_var")
39             )),
40             memberExpr(member(
41               fieldDecl().bind("tid_dep_field")
42             ))
43           ))
44         ))
45       ).bind("straight_assignment")
46     ), this);
47 //Matcher for all variable declarations that reference another
48 // variable in their initializer (in case it is ID-dependent)
49 Finder->addMatcher(
50   stmt(forEachDescendant(
51     varDecl(hasInitializer(
52       forEachDescendant(stmt(anyOf(
53         declRefExpr(
54           to(varDecl())
55         ).bind("assign_ref_var"),
56         memberExpr(
57           member(fieldDecl())
58         ).bind("assign_ref_field")
59       )))
60     )), this);
61 //Matcher for all variables that at some point are assigned a
62 // value from another variable (in case it is ID-dependent)
63 Finder->addMatcher(
64   stmt(forEachDescendant(
65     binaryOperator(allOf(
66       ANY_ASSIGN,
67       hasRHS(
68         forEachDescendant(stmt(anyOf(
69           declRefExpr(
70             to(varDecl())
71           ).bind("assign_ref_var"),
72           memberExpr(
73             member(fieldDecl())
74           ).bind("assign_ref_field")
75         )))
76       ),
77       hasLHS(anyOf(
78         declRefExpr(to(
79           varDecl().bind("pot_tid_var")
80         )),
81         memberExpr(member(
82           fieldDecl().bind("pot_tid_field")
83         ))
84       ))
85     ), this);
86 // (a) The complex set of matchers required to track thread-dependent values
87 // (b) Code used in callbacks to finish propagating thread-dependency information.
88 // Portions of the code related to handling struct members have been omitted for brevity
89 // since they are structurally isomorphic to their variable counterparts, but stored
90 // differently on the Clang AST.
91 // A recursive preorder traversal of an expression to check if any of the
92 // variables that might be referenced in it are marked as ID-dependent
93 const DeclRefExpr * MyCheck::hasIDDepDeclRef(const Expr * e) {
94   //If this expression is a variable reference
95   if (const DeclRefExpr * expr = dyn_cast<DeclRefExpr>(e)) {
96     //check if it's on the ID-dependent variable list
97     if (std::find(IDDepVars.begin(), IDDepVars.end(),
98       dyn_cast<VarDecl>(expr->getDecl()) != IDDepVars.end()) {
99       return expr;
100     } else {
101       return NULL;
102     }
103   } //Else it is either a non-reference leaf node,
104   // or it is an ancestor that needs to be recursed into
105   else {
106     //Iterate over its children and see if any of them are ID-dependent
107     for (auto i = e->child_begin(); i != e->child_end(); ++i) {
108       if (auto * newExpr = dyn_cast<Expr>(i)) {
109         auto * retExpr = hasIDDepDeclRef(newExpr);
110         if (retExpr) return retExpr;
111       }
112     }
113     //If none of the children force an early return with a match, return NULL
114     return NULL;
115   } //Should be unreachable
116 }
117 //Omitted hasIDDepMember function isomorphic to the above, but for
118 // references to field members
119 // The beginning portion of any Match Callback that has to deal with thread
120 // dependency must start by populating the ID dependency list
121 // The first half of the callback only deals with identifying and propagating
122 // ID-dependency information into the IDDepVars vector
123 Variable = Result.Nodes.getNodeAs<VarDecl>("tid_dep_var");
124 Field = Result.Nodes.getNodeAs<FieldDecl>("tid_dep_field");
125 Statement = Result.Nodes.getNodeAs<Stmt>("straight_assignment");
126 RefExpr = Result.Nodes.getNodeAs<DeclRefExpr>("assign_ref_var");
127 MemExpr = Result.Nodes.getNodeAs<MemberExpr>("assign_ref_field");
128 PotentialVar = Result.Nodes.getNodeAs<VarDecl>("pot_tid_var");
129 PotentialField = Result.Nodes.getNodeAs<FieldDecl>("pot_tid_field");
130 //If there's a statement we know is ID-dependent and has a variable on the LHS
131 if (Statement && (Variable || Field)) {
132   //Record that this variable is thread-dependent only if we haven't already
133   if (Variable) {
134     if (std::find(IDDepVars.begin(), IDDepVars.end(),
135       Variable) == IDDepVars.end()) {
136       IDDepVars.push_back(Variable);
137     } else if (Field) {
138       if (std::find(IDDepFields.begin(), IDDepFields.end(),
139         Field) == IDDepFields.end()) {
140         IDDepFields.push_back(Field);
141       }
142     }
143   } else if (Statement) {
144     //Warn that we found an ID-dependent assignment,
145     // but couldn't deduce the assignee variable
146   } else if (Variable) {
147     //Warn that an ID-dependent variable was found,
148     // but we can't make sense of the statement it's in
149   } else if (Field) {
150     //Warn that an ID-dependent field was found,
151     // but we can't make sense of the statement it's in
152   }
153   if (RefExpr && PotentialVar) {
154     //if we got a hit because of a variable in the RHS
155     const auto RefVar = dyn_cast<VarDecl>(RefExpr->getDecl());
156     //If the LHS variable isn't recorded as ID-dependent
157     // but the referenced variable is, the assignee should be recorded
158     if (std::find(IDDepVars.begin(), IDDepVars.end(),
159       PotentialVar) == IDDepVars.end() &&
160       std::find(IDDepVars.begin(), IDDepVars.end(),
161         RefVar) != IDDepVars.end()) {
162       //Record it
163       IDDepVars.push_back(PotentialVar);
164     }
165   }
166 }
167 //Omitted tracking for MemberExprs in LHS/RHS or both, as it
168 // is isomorphic to the above

```

Figure 10. Tracking thread dependency relies on a complex combination of Matcher and Callback functionality, that exploits the preorder AST traversal property of the AST Matchers interface.

```

1 __kernel void dependent() {
2   int tid = get_local_id(0); int gid = get_global_id(0);
3   int tx = tid + get_local_size(0)*gid;
4   for (int i = tx; i<256; i++) {
5     barrier(CLK_LOCAL_MEM_FENCE);
6   }
7   for (int i = 0; i<get_local_id(0); i++) {
8     barrier(CLK_LOCAL_MEM_FENCE);
9   }
10 }

```

```

11 __kernel void independent() {
12   int tid = get_local_id(0);
13   int gid = get_global_id(0);
14   int tx = tid + get_local_size(0)*gid;
15   for (int i = 0; i<256; i++) {
16     barrier(CLK_LOCAL_MEM_FENCE);
17   }
18 }
19 }

```

(a) Example of two for loops that are ID-dependent, one from referencing an ID-dependent variable (“i” via “tx” via “tid” and “gid”) and another due to calling an ID function, as well as a for loop that is not ID-dependent.

```

1 //Omitted ID-dependency matchers
2
3 //Sub-matcher to check if a statment contains
4 // a descendent that is a barrier call
5 const auto HAS_BAR_DESC =
6   hasDescendant(
7     callExpr( callee (
8       functionDecl( anyOf(
9         hasName(" barrier "), hasName(" work_group_barrier ")
10      )
11    ) ). bind(" barrier ")
12  );
13 //Sub-matcher to check if an expression either
14 // calls an ID function or references a variable
15 const auto COND_EXPR =
16   expr( anyOf(
17     hasDescendant( callExpr(
18       callee( functionDecl( anyOf(
19         hasName(" get_global_id "), hasName(" get_local_id ")
20       )
21     ) ). bind(" id_call " ),
22     hasDescendant( stmt( anyOf(
23       declRefExpr(
24         to( varDecl() )
25       ),
26       memberExpr(
27         member( fieldDecl() )
28       )
29     ) )
30   ) ). bind(" cond_expr ");
31 //Matcher that finds any branch statements that
32 // have a potential ID-dependent conditional
33 // either from a call or variable reference
34 Finder->addMatcher(
35   stmt( anyOf(
36     forStmt( allOf(
37       HAS_BAR_DESC, hasCondition( COND_EXPR ) ). bind(" for " ),
38     ifStmt( allOf(
39       HAS_BAR_DESC, hasCondition( COND_EXPR ) ). bind(" if " ),
40     doStmt( allOf(
41       HAS_BAR_DESC, hasCondition( COND_EXPR ) ). bind(" do " ),
42     whileStmt( allOf(
43       HAS_BAR_DESC, hasCondition( COND_EXPR ) ). bind(" while " ),
44     switchStmt( allOf(
45       HAS_BAR_DESC, hasCondition( COND_EXPR ) ). bind(" switch " )
46   ) ), this);

```

(b) The ASTMatcher used to find all conditionals that contain a barrier and need to be checked for ID-dependency that may make the barrier unreachable by some threads.

```

1 //Omitted thread-dependency tracking code
2
3 //We want the diagnostic to emit slightly different
4 // text so it binds on the five potential conditional
5 // types (of which only one will be true) a barrier
6 // and conditional expression, and possibly an id call
7 Barrier = Result.Nodes.getNodeAs<CallExpr>(" barrier ");
8 ForAn = Result.Nodes.getNodeAs<ForStmt>(" for ");
9 IfAn = Result.Nodes.getNodeAs<IfStmt>(" if ");
10 DoAn = Result.Nodes.getNodeAs<DoStmt>(" do ");
11 WhAn = Result.Nodes.getNodeAs<WhileStmt>(" while ");
12 SwAn = Result.Nodes.getNodeAs<SwitchStmt>(" switch ");
13 Cond = Result.Nodes.getNodeAs<Expr>(" cond_expr ");
14 IDCall = Result.Nodes.getNodeAs<CallExpr>(" id_call ");
15 //Figure out which type of conditional
16 int type = -1;
17 if (ForAn) { type = 0; }
18 else if (IfAn) { type = 1; }
19 else if (DoAn) { type = 2; }
20 else if (WhAn) { type = 3; }
21 else if (SwAn) { type = 4; }
22 //If there is a conditional expression, start the
23 // diagnostics (this code gets hit by all matches that
24 // only trigger ID-dependency tracking, so we may
25 // not need to run this code each time)
26 if (Cond) {
27   //Omitted Valid-Barrier-Refinement Code (Fig. 11)
28   //The conditional expression might call an ID function
29   // directly, handle that case
30   if (IDCall) {
31     //Emit a diagnostic warning that all threads may not reach
32     // the barrier due to the ID call in the branch condition
33   } else {
34     //The condition does not call an ID function, but may
35     // reference a variable that needs to be checked for an ID
36     const auto * retExpr = hasIDDepDeclRef(Cond);
37     if (retExpr) { //It has an ID-dependent reference
38       //Emit a diagnostic warning of which variable in
39       // the conditional expression may be ID-dependent
40       // and risk making the barrier unreachable
41     }
42   }
43 }

```

(c) The latter portion of the unreachable barrier callback uses the ID-dependency information computed earlier to check the conditional expression inside every branch that contains a call to a barrier function.

Figure 11. Example code for checking whether a barrier is potentially not reachable by all threads due to being inside a thread-dependent branch.

```

1 switch(type) {
2   case 1: { //Barrier inside if/else-if/else
3     //If this is an else-if, we have already checked it
4     // based on the initial IfStmt
5     //Get the parent to see if we're in an else branch
6     const auto parents = Result.Context->getParents>(*IfAnsc);
7     const Stmt * parent = NULL;
8     if(!parents.empty()) { parent = parents[0].get<Stmt>(); }
9     if (parent) {
10      const IfStmt * ifParent = dyn_cast<IfStmt>(parent);
11      if (ifParent != NULL && ifParent->getElse() == IfAnsc) {
12        //If we are in an else branch, return
13        return;
14      } }
15    //First collect all else/if branches to iterate over
16    std::list<const Stmt*> checks;
17    auto currIf = IfAnsc;
18    checks.push_back(currIf->getThen()); //Record initial Then
19    const Stmt * nextIf = IfAnsc->getElse();
20    //Iterate through else-if Thens
21    for (; nextIf != NULL && dyn_cast<IfStmt>(nextIf);
22         nextIf=currIf->getElse()) {
23      currIf = dyn_cast<IfStmt>(nextIf);
24      checks.push_back(currIf->getThen());
25    }
26    if (nextIf == NULL) { // There is no final else
27      //then we cannot prove all threads execute the barrier
28      break;
29    } else {
30      //nextIf holds the actual final else statement
31      checks.push_back(nextIf);
32    }
33    //The list of branches to check is complete
34    // make sure they all call a barrier the same number of times
35    //Serialize the if/else-if/else tree to make traversal easy
36    std::list<const Stmt*> flatIfElse;
37    preorderFlattenStmt(IfAnsc, &flatIfElse);
38    std::list<const Stmt*>::iterator currCase = checks.begin();
39    int maxBarriers = -1; int currBarriers = 0;
40    //Iterate over every AST node in the if/else-if/else tree
41    for (auto itr = flatIfElse.begin(); itr != flatIfElse.end() &&
42         currCase != checks.end(); itr++) {
43      //If starting the next case, or at the last element
44      if (*itr == *(std::next(currCase, 1)) ||
45          std::next(itr, 1) == flatIfElse.end()) {
46        //No barriers, abort to diagnostics
47        if (currBarriers == 0) { break; }
48        if (maxBarriers != -1 && currBarriers != maxBarriers) {
49          //This branch has a different number of barriers than
50          // a previously processed branch, abort to diagnostics
51          break;
52        }
53        //First case evaluated, set the expected barrier count
54        if (maxBarriers == -1) maxBarriers = currBarriers;
55        //Advance to the next branch
56        currCase++;
57        //Just processed the last branch and haven't failed yet,
58        // then assume the barrier is safe and return!
59        if (currCase == checks.end()) { return; }
60      }
61      //At the beginning of each branch, reset the counter
62      if (*itr == *currCase) { currBarriers = 0; }
63      //When we encounter a function call, check if it's a barrier
64      if (auto call = dyn_cast<CallExpr>(*itr)) {
65        if (auto callDecl = call->getDirectCallee()) {
66          std::string name = callDecl->getNameAsString();
67          if (name != "" && name != "barrier" &&
68              name != "work_group_barrier") {
69            // if it's a call to non-barrier function, we don't know
70            // whether it contains a barrier, conservatively abort
71            break;
72          } else {
73            currBarriers++; //count the barrier
74          }
75        } else {
76          //Can't deduce what is being called
77          break; // conservatively abort
78        }
79      }
80      //If we hit any nested branches, break
81      if (isa<IfStmt>(*itr) || isa<SwitchStmt>(*itr) ||
82          isa<ForStmt>(*itr) || isa<DoStmt>(*itr) ||
83          isa<WhileStmt>(*itr) || isa<ReturnStmt>(*itr) ||
84          isa<BreakStmt>(*itr) || isa<GotoStmt>(*itr)) {
85        // If parent of currCase, then it's the initial If
86        // If parent of currCase+1, then it's an Else-If
87        bool isOuterIf = false;
88
89        if (const IfStmt * par = dyn_cast<IfStmt>(*itr)) {
90          //Make sure it's not a nested If
91          for ( auto cItr = par->child_begin();
92               cItr != par->child_end(); cItr++) {
93            if ((cItr != par->child_end()) && (*cItr == *currCase ||
94                *cItr == *(std::next(currCase, 1)))) {
95              isOuterIf=true;
96              break; //Out of the child check loop
97            } }
98          //If it is neither, then it must be an interior branch, so abort
99          if (!isOuterIf) { break; }
100        } }
101      break; //End of if/else-if/else processing
102    case 4: { //Barrier inside switch/case
103      bool hasDefault = false;
104      const SwitchCase * cases = SwitchAnsc->getSwitchCaseList();
105      std::list<const Stmt*> checks;
106      //This iterates from the bottom of the case list, upwards
107      for (; cases != NULL; cases= cases->getNextSwitchCase()) {
108        if (dyn_cast<DefaultStmt>(cases)) {
109          hasDefault=true;
110        }
111        //Skip over cases with no body (immediate subsequent case)
112        if (!dyn_cast<CaseStmt>(cases->getSubStmt())) checks.push_back(cases);
113        //If no default case, can't guarantee all threads execute barrier
114        if (hasDefault == false) break;
115        //The list of cases with bodies is complete
116        // make sure they all call a barrier the same number of times
117        //Serialize the switch/case tree to make traversal easy
118        std::list<const Stmt*> flatSwitch;
119        preorderFlattenStmt(SwitchAnsc->getBody(), &flatSwitch);
120        checks.reverse(); //Check cases from top to bottom
121        std::list<const Stmt*>::iterator currCase = checks.begin();
122        int maxBarriers = -1; int currBarriers = 0;
123        for (auto itr = flatSwitch.begin(); itr != flatSwitch.end() &&
124             currCase != checks.end(); itr++) {
125          //Reset counters when a new case is reached
126          if (*itr == *currCase) { currBarriers = 0; }
127          //If we continue into a subsequent case without a break
128          if (*itr == *(std::next(currCase, 1))) {
129            //Abort if we have any accumulated barriers
130            // since we are guaranteed to have more than subsequent case
131            if (currBarriers > 0) { break; }
132            currCase++; //otherwise just advance to next case
133          }
134          //Cases are formally ended by breaks
135          if (dyn_cast<BreakStmt>(*itr)) {
136            //No barriers, abort to diagnostics
137            if (currBarriers == 0) { break; }
138            if (maxBarriers != -1 && currBarriers != maxBarriers) {
139              break; //Some other case has a different number of barriers
140            }
141            //This is the first case evaluated, set the expected count
142            if (maxBarriers == -1) maxBarriers = currBarriers;
143            currCase++; //Advance to the next case
144            //If we just processed our last case and haven't failed yet,
145            // then we can assume the barrier is safe and return!
146            if (currCase == checks.end()) { return; }
147          }
148          //When we encounter a function call, check if it's a barrier
149          if (auto call = dyn_cast<CallExpr>(*itr)) {
150            if (auto callDecl = call->getDirectCallee()) {
151              std::string name = callDecl->getNameAsString();
152              if (name != "" && name != "barrier" &&
153                  name != "work_group_barrier") {
154                // if it's a call to non-barrier function, we don't know
155                // whether it contains a barrier, conservatively abort
156                break;
157              } else {
158                currBarriers++; //count the barrier
159              }
160            } else {
161              //Can't deduce what is being called
162              break; //conservatively abort
163            }
164          }
165          //If we hit any nested branches or jumps, break
166          if (isa<IfStmt>(*itr) || isa<SwitchStmt>(*itr) ||
167              isa<ForStmt>(*itr) || isa<DoStmt>(*itr) ||
168              isa<WhileStmt>(*itr) || isa<ReturnStmt>(*itr) ||
169              isa<GotoStmt>(*itr)) {
170            break;
171          } }
172        break; //End of switch/case processing
173      }

```

Figure 12. Refinement step to search through if/else and switch/case statements to attempt to prove all branches execute the same number of barriers.

```

1 fooKern = clCreateKernel(..., "foo", ...);
2 ...
3 clSetKernelArg(fooKern, 0, sizeof(int), &myAInt);
4 clSetKernelArg(fooKern, 1, sizeof(cl_mem), &myBclMem);
5 clSetKernelArg(fooKern, 2, sizeof(float), &myCPtr);
6 clEnqueueNDRangeKernel(..., fooKern, ...);
7 ...
8 clSetKernelArg(fooKern, 0, sizeof(unsigned int), &myAUInt);
9 clSetKernelArg(fooKern, 1, sizeof(cl_mem), &myBclMem);
10 clSetKernelArg(fooKern, 2, sizeof(float), &myCPtr);
11 clEnqueueNDRangeKernel(..., fooKern, ...);

```

(a) Example of OpenCL host code to create and launch the kernel from Figure 13d. Note that the first launch sets arguments of the correct host types, and the second misuses an unsigned integer for the zeroth argument.

```

1 //Matcher to catch all declarations of cl_kernels
2 Matcher.addMatcher(
3   varDecl(
4     hasType(asString("cl_kernel"))
5     ).bind("kernel_decl"), new HostKernelDeclHandler());
6 //Matcher to catch all references to cl_kernels
7 Matcher.addMatcher(
8   declRefExpr(to(varDecl(
9     hasType(asString("cl_kernel"))
10    )), bind("kernel_ref"), new HostKernelDeclRefHandler());
11 //Sub-matcher to check whether a RHS calls clCreateKernel
12 const auto createKernelRHS =
13   callExpr(callee(
14     functionDecl(hasName("clCreateKernel"))
15     )), bind("create");
16 //Matcher for any assignment of the return value of clCreateKernel
17 Matcher.addMatcher(
18   stmt(anyOf(
19     // catch declarations that are also initializations
20     declStmt(hasDescendant(
21       varDecl(hasInitializer(createKernelRHS)).bind("assignee")
22     )),
23     // catch regular variable assignments
24     binaryOperator(
25       hasOperatorName("="),
26       hasRHS(createKernelRHS),
27       hasLHS(declRefExpr(to(
28         varDecl().bind("assigned_kernel")
29       )))
30     ),
31     // catch assignments to struct members
32     binaryOperator(
33       hasOperatorName("="),
34       hasRHS(createKernelRHS),
35       hasLHS(memberExpr(member(
36         valueDecl().bind("assigned_kernel")
37       )))
38     )
39   )), bind("assign_stmt"), new HostKernelCreateHandler());
40 //Matcher to catch all argument assignments
41 Matcher.addMatcher(
42   callExpr(callee(
43     functionDecl(hasName("clSetKernelArg"))
44     )), bind("setarg"), new HostKernelArgHandler());
45 //Matcher to catch all kernel launches
46 Matcher.addMatcher(
47   callExpr(callee(
48     functionDecl(anyOf(
49       hasName("clEnqueueNDRangeKernel"),
50       hasName("clEnqueueTask")
51     ))
52   )), bind("launch"), new HostKernelCallHandler());

```

(b) Examples of matchers for the host AST(s)

```

1 __kernel void foo(
2   int a,
3   __global float * b,
4   float c) {...}

```

(d) Example of a kernel prototype to match.

```

1 //Matcher for any function with the OpenCL kernel attribute
2 Matcher.addMatcher(
3   functionDecl(
4     hasAttr(attr::Kind::OpenCLKernel)
5     ).bind("kernelDecl"), new DevKernelHandler());

```

(e) Kernel Matcher

```

1 //Stores declarations of cl_kernels
2 //If global or extern global, adds to cross-AST
3 // extern management structure
4 void HostKernelDeclHandler::run(const MatchResult &Result) {
5   const auto *KernelDecl =
6     Result.Nodes.getNodeAs<VarDecl>("kernel_decl");
7   if (KernelDecl) {
8     if (KernelDecl->getType().getAsString() == "cl_kernel") {
9       //Store the declaration for deferred stages
10      if (KernelDecl->isGlobal() && !KernelDecl->isExtern()) {
11        //Store as canonical global cl_kernel decl by name
12      } else if (KernelDecl->isGlobal() && KernelDecl->isExtern()) {
13        //Store as extern ref to global cl_kernel decl by name
14      }
15    } } }
16 //Attaches every reference of a cl_kernel object to the
17 // declaration it is referencing
18 //Unused for now but will become part of a cl_kernel
19 // propagator for CFG calls and other references
20 void HostKernelDeclRefHandler::run(const MatchResult &Result) {
21   const auto *KernelDeclRef =
22     Result.Nodes.getNodeAs<DeclRefExpr>("kernel_ref");
23   if (KernelDeclRef) {
24     if (KernelDeclRef->getType().getAsString() == "cl_kernel") {
25       //Directly attach to a previously-declared and stored
26       // cl_kernel, since it must be defined before reference
27     }
28   }
29 }
30 //Record information about calls to clCreateKernel
31 // and the assignee cl_kernel that stores the host object
32 void HostKernelCreateHandler::run(const MatchResult &Result) {
33   const auto *CreateKernel =
34     Result.Nodes.getNodeAs<CallExpr>("create");
35   const auto *KernVar =
36     Result.Nodes.getNodeAs<ValueDecl>("assignee");
37   //If the function is called and it has an assignee
38   if (CreateKernel && KernVar) {
39     //First get the name of the kernel it is creating
40     const StringLiteral * kernStr =
41       dyn_cast<StringLiteral>(
42         CreateKernel->getArg(1)->IgnoreImplicit());
43     if (kernStr) { //easy, it's a string literal
44       //Force the SourceManager and Context to persist
45       //Create and populate a storage variable
46       //Record the created kernel in a list for later processing
47     } else {
48       //TODO: If it's a reference to a string we will have to attempt
49       // to propagate it to find the StringLiteral(s) it's made of
50     }
51   } else {
52     //Omitted code to emit diagnostics for partial matches
53   }
54 }
55 //Record information about calls to clSetKernelArg
56 void HostKernelArgHandler::run(const MatchResult &Result) {
57   const auto *SetArgs = Result.Nodes.getNodeAs<CallExpr>("setarg");
58   if (SetArgs) {
59     //Create and populate a storage variable
60     //Record the assigned argument in a list for later processing
61     SetArgsCalls.push_back(data);
62   }
63 }
64 //Record information about kernel launches
65 void HostKernelCallHandler::run(const MatchResult &Result) {
66   const auto *LaunchKernel = Result.Nodes.getNodeAs<CallExpr>("launch");
67   if (LaunchKernel) {
68     //Create and populate a storage variable
69     //Record the launched kernel in a set sorted by source
70     // manager and location for later processing
71     KernelLaunches.insert(data);
72   }
73 }

```

(c) Examples host AST callbacks to record information for deferred processing

```

1 //Record information about kernel prototypes
2 void DevKernelHandler::run(const MatchResult &Result) {
3   const auto *KernelDecl =
4     Result.Nodes.getNodeAs<FunctionDecl>("kernelDecl");
5   if (KernelDecl) {
6     //Force the SourceManager and ASTcontext to persist
7     //Create and populate a storage variable
8     //Record the kernel prototype by name in a map for later processing
9     KernelPrototypes[KernelDecl->getName()] = data;
10  }
11 }

```

(f) Example kernel AST callback

Figure 13. The per-AST phase of FLOCL-MAST has separate matchers and callbacks for host and device code. Both record information that is later used by the cross-AST phase detailed in Figures 14 and 15.

```

1 //The deferred step to attach cl_kernels from a clCreateKernel
2 // call to a kernel prototype from a device AST
3 void bindKernelProtos() {
4 //Iterate over every clCreateKernel call from every host AST
5 for (createData * create : CreateKernels) {
6   if (create != NULL) {
7     //Look up the prototype to associate to based on the name used
8     // in the clCreateKernel call (for now a StringLiteral)
9     kernelProtoData * proto = KernelPrototypes[create->protoName];
10    if (proto != NULL) {
11      //If a matching prototype exists, populate and record a
12      // storage structure on a map, using the assigned
13      // cl_kernel's declaration as the key
14      if (!assignee->isGlobal()) {
15        BoundKernels[create->assignee] = data;
16      } else {
17        //look up canonical global declaration
18        BoundKernels[canonicalDecl] = data;
19      }
20    } else {
21      //Warn that a matching name prototype can't be found
22    } } } }

```

(a) The cross-AST phase must bind `cl_kernel` objects from the host AST(s) to kernel prototype declarations from the device AST(s), using information from `clCreateKernel` calls on the host AST(s).

```

1 //The deferred step to attach each launch and its newly-associated
2 // arguments to a kernel prototype, via the ones bound to cl_kernels
3 //TODO: implement cross-function and cross-alias cl_kernel
4 // tracking to propagate launches correctly to cl_kernels that
5 // are not globally declared (or in declared and created in
6 // the same scope as the launch)
7 void propKernelsToLaunches() {
8 //Iterate over every launched kernel in every AST
9 for (every launch : KernelLaunches) {
10  const ValueDecl * calledValue = NULL;
11  const Expr * argExpr;
12  //Try to resolve the launch's cl_kernel argument
13  // to a variable or member declaration
14  //If global, use the canonical declaration not an extern
15  if (calledValue == NULL) {
16    //If the cl_kernel can't be resolved, warn that we can't
17    // deduce what cl_kernel is being launched
18  }
19  //Search for a bound kernel prototype with a matching host cl_kernel
20  auto boundKernel = BoundKernels.find(calledValue);
21  if (boundKernel != BoundKernels.end()) {
22    //If a prototype is found, record this launch and its arguments
23    // for later processing
24    (*boundKernel).second->launches->push_back(*launches);
25  } } }

```

(c) The cross-AST phase must attach launched kernels (with newly-attached arguments) to a prototype. This phase relies on the launched `cl_kernel` declaration matching one successfully bound to a prototype by a `clCreateKernel` call.

```

1 //The deferred step to attach clSetKernelArgs to the lexically-next
2 // kernel launch they will apply to
3 //Warning: Lexically-next is a heuristic that will not take into
4 // account intervening assignments that happen due to a call
5 //Warning: does not currently care about scope so an assignment
6 // in one function may apply to a launch in another function if
7 // they share the same cl_kernel object, regardless of the order
8 // the functions are invoked at runtime
9 void clusterArgsToLaunches() {
10 //Iterate over all clSetKernelArgs calls from all host ASTs
11 for (setArgsData * setArgs : SetArgsCalls) {
12 //Try to resolve the assignment's cl_kernel argument
13 // to a variable or member declaration
14 if (setValue == NULL) {
15 //If the cl_kernel can't be resolved, warn that we can't
16 // deduce what cl_kernel the argument is being assigned to
17 }
18 //Iterate over all kernel launches this assignment may apply to
19 //The launches set is sorted by the AST first, and the lexical
20 // position second and we assume the assignment will not apply
21 // to a launch that is lexically-before it as a simplifying
22 // heuristic, so we start iterating at the first launch after
23 // the assignment, and end if we ever leave the AST or data
24 // structure, or start looking at launches that are somehow
25 // before the assignment.
26 for (all such launches : KernelLaunches) {
27  const ValueDecl * calledValue = NULL;
28  const Expr * largExpr;
29  //Try to resolve the launch's cl_kernel argument
30  // to a variable or member declaration
31  if (calledValue == NULL) {
32    //If the cl_kernel can't be resolved, warn that we can't
33    // deduce what cl_kernel is being launched
34  }
35  //If the two cl_kernels match (are the same declaration)
36  if (setValue == calledValue) {
37    //Figure out which kernel argument position is assigned
38    // for simplicity assume index is IntegerLiteral (for now)
39    if (const IntegerLiteral * intVal =
40        dyn_cast<IntegerLiteral>(
41            setArgs->call->getArg(1)->IgnoreImplicit())
42        ) {
43      //Bind the argument assignment call to the launch, using
44      // the position value as the map key
45      (LaunchesWithArgs[*launches])
46      [intVal->getValue().getLimitedValue()] = setArgs;
47      //Warn if we are overwriting a previously-assigned value
48    } else {
49      //Emit a diagnostic if the positional argument is not
50      // an IntegerLiteral that can be statically determined
51    }
52  }
53  //A matching kernel has been found, don't attempt to apply
54  // it to other kernels. TODO: Relax to support OpenCL's
55  // argument retention if they are not re-assigned
56  break;
57  //TODO: Early terminate if we have left assignment scope
58 } } }

```

(b) The cross-AST phase must heuristically determine what `clSetKernelArg` assignments on the host AST(s) apply to which kernel launches. Currently, the arguments must be *lexically-before* the launch in the same AST, and only apply to the first following launch with the same `cl_kernel`.

Figure 14. The first stages of the deferred cross-AST check for type consistency between kernel and device marry information from separate ASTs.

```

1 //Check whether a host and device type are compatible
2 //Warning: For now dynamically-allocated local memory
3 // regions will emit a type-incompatibility because they use
4 // a NULL pointer for the host variable to copy
5 //Warning: cl_mem's are currently treated as a wild card which
6 // matches with any __global or __constant pointer on the device
7 //TODO: Attempt to resolve the real type stored in a cl_mem
8 bool checkH2DTypeCompat(QualType hostType, QualType devType) {
9     //Wildcard match cl_mem types on the host to any __global or
10    // __constant pointer on the device
11    if (hostType.getAsString() == "cl_mem *" ||
12        hostType.getAsString() == "const cl_mem *") {
13        if (const Type * type = devType.getTypePtrOrNull()) {
14            if (!type->isAnyPointerType()) {
15                //If the device type is not a pointer, warn of type mismatch
16                return false;
17            } else {
18                //If the device type is a pointer, ensure it points
19                // to an appropriate address space
20                unsigned addrSpace = type->getPointeeType().getAddressSpace();
21                if (addrSpace != LangAS::opencl_global &&
22                    addrSpace != LangAS::opencl_constant) {
23                    //Warn if address space is neither __global nor __constant
24                    return false;
25                } else {
26                    //TODO: improve this wildcard match with any resolvable info
27                    // about the real type stored in the cl_mem on the host
28                    return true;
29                }
30            }
31        } else {
32            //Error if for some reason the device type cannot be resolved
33            return false;
34        }
35    } //No need for else, all cl_mem branches return early
36    //Handle non-pointer types
37    if (const Type * hType = hostType.getTypePtrOrNull()) {
38        if (!hType->isAnyPointerType()) {
39            //Warn if host type is not a pointer to memory to be copied
40            //TODO Handle NULL pointers for dynamic __local memory
41            return false;
42        } else {
43            //Discard any extra qualifiers and check if the types match
44            if (hType->getPointeeType()
45                .withoutLocalFastQualifiers().getAsString()
46                == devType.withoutLocalFastQualifiers().getAsString()) {
47                return true;
48            } else {
49                return false;
50            }
51        }
52    } //Emit an error that the host type cannot be deduced
53    //Fallback on returning false
54    return false;
55 }
56
57 (a) Host and device ASTs may use different types and memory spaces to refer to the
58 semantically-same type of variable. A helper function is used to concisely check if a
59 host argument's type is compatible with a device parameter's.
60
61
62
63
64
65
66
67
68
69
70
71 //The deferred step to check every launch for conformance
72 // to the prototype specified in device code. For now this
73 // is limited to checking the parameters specified on the device
74 // against the arguments provided to them on the host.
75 //TODO: Check for excess arguments specified on the host
76 //TODO: Check workgroup sizes against potential kernel attributes
77 void checkLaunchesToProtos() {
78     //Iterate over all prototypes that have been bound to cl_kernels
79     for (auto boundPair : BoundKernels) {
80         BoundKernelData * boundKernel = boundPair.second;
81         //Iterate over all the launches of this prototype
82         for (launchData * launch : *(boundKernel->launches)) {
83             //First, ensure that if the prototype has zero
84             // parameters, that none have been set
85             if (boundKernel->protoData->decl->getNumParams() == 0
86                 && LaunchesWithArgs.find(launch) !=
87                 LaunchesWithArgs.end()) {
88                 //Warn of argument to a parameterless kernel
89                 break;
90             }
91             //Iterate over all parameters in the prototype
92             for (int i = 0;
93                 i < boundKernel->protoData->decl->getNumParams();
94                 i++) {
95                 //Search for an argument in this position for this launch
96                 auto setArg = LaunchesWithArgs[launch].find(i);
97                 if (setArg == LaunchesWithArgs[launch].end()) {
98                     //No argument specified, warn of a missing argument
99                 } else {
100                    //If it has an argument in this position,
101                    // get it and check it against the prototype
102                    setArgsData * data = (*(setArg).second);
103                    //Get the type of the argument from the host
104                    QualType argType =
105                        data->call->getArg(3)->IgnoreImplicit()->getType();
106                    //Try to strip away any explicit casts to (void *)
107                    const CastExpr * explicitCast =
108                        dyn_cast<CastExpr>(data->call->
109                            getArg(3)->IgnoreImplicit());
110                    if (explicitCast && (explicitCast->getSubExprAsWritten()
111                        ->getType().getAsString() != "void *")) {
112                        argType = explicitCast
113                            getSubExprAsWritten()->getType();
114                    }
115                    //Get the device parameter type
116                    QualType paramType = boundKernel->protoData
117                        ->decl->getParamDecl(i)->getType();
118                    //Check host and device types for compatibility
119                    if (checkH2DTypeCompat(argType, paramType)) {
120                        //The types match, either emit a success or do nothing
121                    } else {
122                        //Warn that the device and host types are incompatible
123                    }
124                }
125            }
126        }
127    }
128    //TODO check for arguments with out of bounds parameters
129    //TODO handle dynamic __local memory arguments
130 }
131 }
132 }

```

Figure 15. The final stages of the cross-AST type consistency check use the previously-married information to check every assigned launch parameter for consistency with the associated device kernel's prototype.

(b) The final cross-AST phase for checking host-side launches for existence and type-consistency of argument/parameters must iterate over all arguments assigned to every launch on every host AST and make sure they match the kernel prototype that is being invoked.