A Composable Workflow for Productive Heterogeneous Computing on FPGAs via Whole-Program Analysis and Transformation

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

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

Wu-chun 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 the very expensive compilation, placement, and routing on FPGAs, respectively. 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 and applications.**

I. Introduction

Heterogeneous computing has become a frontrunner in the computational race for performance and energy efficiency. FPGAs, in particular, have gained increasing attention in the general-purpose computing community due to the emergence of high-level synthesis (HLS) tools based on OpenCL [1], [2]. Ideally, heterogeneous code should be "write once, run anywhere" regardless of the target device. In practice, however, productivity is limited by the community fragmentation between programming languages and the significant investment in the existing heterogeneous code implemented in vendor-specific languages such as CUDA [3]. Therefore, FPGAs could significantly benefit from automated tools to port the vendor-specific code to the vendor-agnostic OpenCL, thus allowing devices to be procured on the basis of performance, energy efficiency, and cost, rather than language compatibility only. Further, the FPGA is a new platform for many GPGPU and CPU-only software developers, presenting an opportunity for productivity tools to reduce the learning curve by providing advisories about the unique semantic and performance pitfalls present in OpenCL-on-FPGA development.

More importantly, 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 [4]–[7] to leverage the wealth of the existing CUDA code on alternative heterogeneous platforms such as FPGAs. However, these tools only work on a single translation unit (TU) — i.e., a single source file and all the included headers — at a time, which limits their ability to propagate code transformations broadly across large, multi-file 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 exists a compelling need for tools that examine applications holistically (e.g., spanning all source files in the codebase at once) to observe possible inconsistencies and affect wide-reaching global transformations.

As such, this paper presents a workflow of automated tools that provides whole-program analysis and transformation both within and *across* source files in support of productive OpenCL programming on heterogeneous parallel systems with FPGAs, GPUs, and/or CPUs. Whole-program analysis and transformation (e.g., refactoring and translation) is difficult to perform in the context of separately compiled and linked languages, such as C/C++ and derivatives (CUDA and OpenCL), because of the traditionally hard separation between TUs at compile time. However, robust *cross*-TU tools can be realized by restricting the problem to only the necessary explicitly-shared interfaces that bridge between TUs at link time or runtime (i.e., via shared headers).

Specifically, we make the following research contributions:

- *A workflow of automated, composable tools to rapidly accelerate the development process of heterogeneous parallel code.* Specifically, CU2CL-MAST[1] is a whole-program source-to-source translator that holistically transforms existing heterogeneous applications from the vendor-specific CUDA to the vendor-agnostic OpenCL. FLOCL[2] and FLOCL-MAST[3] are static code analyzers (i.e., linters) for single-file, kernel code and cross-file, host-to-device code, respectively, which detect possible semantic errors, runtime failures, and performance traps before investing time in expensive compilation and debugging/profiling (see §II).
- *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 (see §III).
- *An analysis of common semantic faults detected by the workflow tools in OpenCL benchmark suites.* Specifically, FLOCL identifies over 155 potential kernel performance and semantic faults within the kernel files of five common OpenCL benchmark suites. In addition, FLOCL-MAST detects inconsistent kernel calls between statically-compiled host code and JIT-compiled device code, which can result in undefined run-time behavior (see §III).

II. Whole-Program Workflow

Figure 1 presents an overview of the proposed workflow and explains how the tools interact. To migrate an existing CUDA code to OpenCL for execution on an FPGA or alternative OpenCL platform, CU2CL-MAST (§II-D) expands an existing CUDA-to-OpenCL translator to the whole-program scope to tackle large-scale applications. OpenCL *kernel files* (handwritten or translated by CU2CL-MAST)

This work was supported in part by NSF I/UCRC IIP-1266245 via NSF CHREC.

[1]CUDA-to-OpenCL translator with Multi-Abstract Syntax Trees
[2]FPGA Linters for OpenCL
[3]FLOCL with Multi-Abstract Syntax Trees

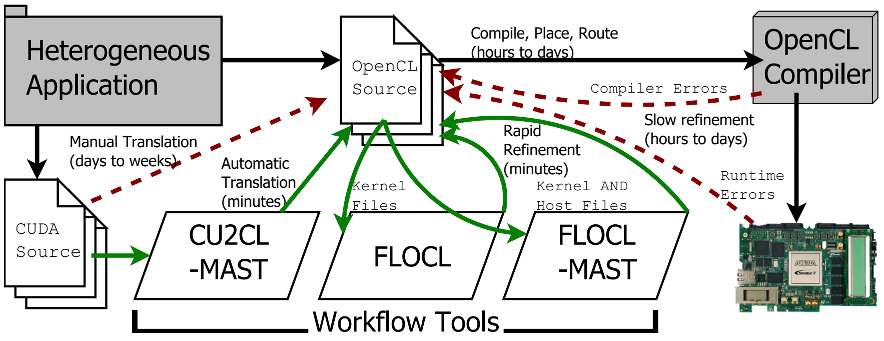
Heterogeneous Application
OpenCL Source
Compile, Place, Route (hours to days)
OpenCL Compiler
Compiler Errors
Slow refinement (hours to days)
Runtime Errors
Rapid Refinement (minutes)
Kernel AND Host Files
Kernel Files
Manual Translation (days to weeks)
Automatic Translation (minutes)
CUDA Source
CU2CL -MAST
FLOCL
FLOCL -MAST
Workflow Tools

Fig. 1: The workflow consists of three tools that can be used in concert or isolation: 1) CU2CL-MAST, to migrate existing CUDA codes to OpenCL, 2) FLOCL, to advise of possible flaws in OpenCL kernels, and 3) FLOCL-MAST, to ensure OpenCL kernels have a consistent interface between host and device.

are *individually* analyzed with FLOCL (§II-A) to detect OpenCL- and FPGA-specific semantic and performance faults. Next, the *entire* OpenCL application (i.e., all host and kernel files) is further inspected with FLOCL-MAST (§II-C2) to validate the separately-compiled, host-to-device interface. Changes suggested by the linter are then manually implemented by the developer. 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.

The following subsections detail the design and implementation of the workflow tools, including our multi-AST (abstract syntax tree) framework that enables cross-file operations, how we leverage Clang's libTooling for code management, as well as the FLOCL, FLOCL-MAST, and CU2CL-MAST tools themselves.

A. Multi-AST Framework

A secondary contribution underlying the CU2CL-MAST and FLOCL-MAST tools is our multi-AST framework that allows the tools to operate across all the source files in an application within a single invocation. This framework is necessitated by the design of C, C++, and other parallel C-like languages (e.g., CUDA and OpenCL), in which 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, as shown in Figure 2. 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 ASTs of all source files and enables the elements present in a given AST to affect actions elsewhere in other source files' ASTs.

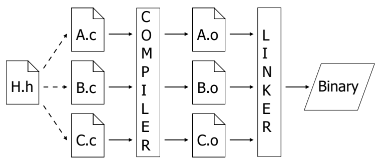

Fig. 2: Typical separate compilation of object files before linking together.

Figure 3 shows the design of the framework that provides several critical components to enable the creation of source-level, cross-file tools. First, it leverages the Clang [8] compiler's *libTooling* API and frontend to parse the individual source files and translate them into navigable ASTs for use in later stages. The libTooling library provides control and modification of the behavior of Clang's compiler frontend, providing fine-grained control over the processing of individual TUs into ASTs. Second, the framework uses custom *per-file* logic to perform local AST analysis and source transformation and to record interesting AST features for use in the following *cross-file* stage. Finally, the cross-file stage relies on the minimal shared interface between link units to connect across ASTs and to process the cross-file interactions recorded in the per-file stage. The following subsections detail the design of each framework stage and their roles in creating the multi-AST transformation and analysis tools.

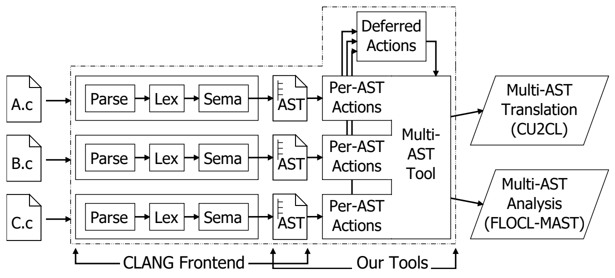

Fig. 3: Multi-AST tools coordinate across invocations of the Clang frontend to provide whole-program translation or analysis.

B. AST Generation and Management

The Clang compiler acts as the framework's driver that translates source files into ASTs, and the libTooling API provides a mechanism to manage multiple invocations of Clang, customize the `FrontendActions` they apply to process source code into ASTs, and access the resulting ASTs and auxiliary data structures.

Algorithm 1 Skeleton of multi-AST *Tool* execution.

```
START
Custom option processing
Pre-Clang actions
for each source file from command line do
    Customize frontend invocation via callback
    Invoke Clang frontend
    Frontend parsing, lexing, AST-generation
    Post-frontend, per-file actions, record deferred work
end for
Multi-AST actions (consume deferred work)
END
```

Algorithm 1 provides a skeleton of the execution phases of our multi-AST tools. The cross-file tools customize the frontend behavior on a per-file basis (i.e., to provide different compilation options for OpenCL device and host files) via pre-frontend callbacks. Next, Clang performs the parsing, lexing, semantic analysis, and AST generation for each file. The custom `FrontendActions` implement the tool-specific, per-file processing and then traverse the ASTs to record the potential cross-file work in a tool scope *above* the individual

```
struct misaligned {
char a;
double b;
char c;
};

struct packed_and_aligned {
char a;
double b;
char c;
} __attribute((packed))
  __attribute((aligned(16)));
```

(a) Example of a misaligned and unpacked struct that would use 24 bytes as well as an equivalent packed and aligned struct that uses 16 bytes.

```
Finder->addMatcher(
  recordDecl(
    isStruct()
  ).bind("struct"), this);
```

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

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

(c) Skeleton of the Match Callback used to filter out only problematic structs. Alignment and packed width calculations omitted for clarity.

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

Clang invocations. (The different `FrontendActions` are detailed further in subsequent sections.) After all Clang invocations, the tool utilizes the generated ASTs and deferred work to perform the cross-file analysis and/or transformation.

C. FLOCL and FLOCL-MAST

FLOCL and FLOCL-MAST improve the productivity of programmers in developing OpenCL applications for FPGAs by providing additional compile-time advisories about potential semantic or performance faults. The single-file, kernel linters are implemented as a set of modules for the *clang-tidy* [9] 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 platforms. Therefore, the additional cross-file FLOCL-MAST is devised to detect interface inconsistencies between the separately-compiled OpenCL host and device codes.

*1) **FLOCL**:* The *clang-tidy* [9] 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 and identifying dead code). However, as a sub-dialect of C, general OpenCL kernels as well as FPGA-specific kernels have special constraints that require unique linting passes. 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 — essentially restricted-scope subsets of the application code, such as a single function call, a nested `if/else` tree, or entire function definition) that are defined in terms of the node types and their relationships. 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 construct and a callback to diagnose it. These checks typically implement guidelines or restrictions from the OpenCL standard, OpenCL SDK documentations, and/or literature. To create a new linter, a synthetic code example is manually constructed to exhibit the behavior, and then the AST representation of the code is examined for the minimum matchable Sub-AST. A matcher to catch the symptomatic Sub-ASTs is constructed, and then a match callback is implemented to provide further refinement and diagnostics.

We currently provide checks for inefficiently aligned/packed structs (§II-C1a), barriers in single-threaded kernels (§II-C1b), possibly-unreachable barriers inside conditionals (§II-C1d), and ID-dependent backward branching (§II-C1e). These checks are derived from restrictions in the OpenCL specification and the Intel/Altera FPGA best practices guides. Additionally, a technique was developed (§II-C1c) 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 (§II-C1d) and (§II-C1e).

a) 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 four (4) bytes is desired, and an optimal 128-byte alignment is suggested [10]. 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; it 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 then 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 elements are stored contiguously; it 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.)

b) Single-work item barriers: The Intel FPGA OpenCL platform synthesizes hardware differently during kernel compilation if an OpenCL kernel is suited for execution across multiple work items (*NDRange*) versus a single work item (*SWI*). It determines suitability based on calls to OpenCL kernel functions and/or compiler attributes; it also varies with the version of the device compiler. For example, calls to either `get_global_id` or `get_local_id` result in compilation as an NDRange kernel [10], [11]. If the user intends the kernel for SWI execution, any barrier synchronization should instead be relaxed to a `mem_fence`. Consequently, we developed a linter to address the presence of barriers in potential SWI kernels. The matcher detects any OpenCL kernel function with barriers but without calls to a global or local ID function. The callback then provides a custom message, based on the target compiler version, to indicate that either the barrier must be removed or should be replaced with a `mem_fence` for performance.

c) Thread-dependency tracking: Several performance and semantic issues result from divergent threads within a 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

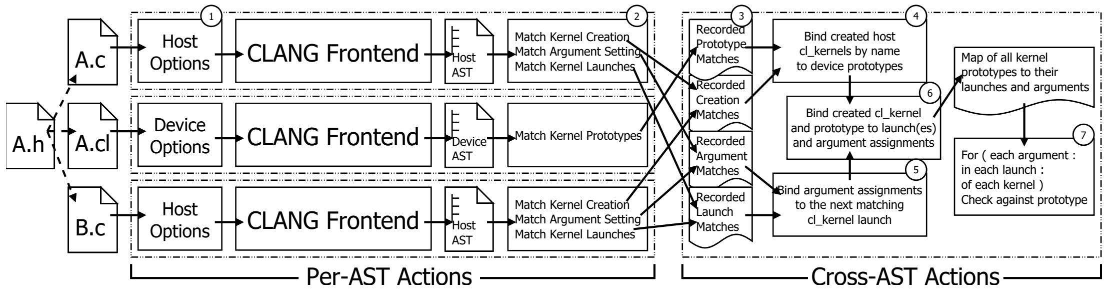
A.c
A.h
A.cl
B.c
1
Host Options
CLANG Frontend
Host AST
Device Options
Device AST
2
Match Kernel Creation
Match Argument Setting
Match Kernel Launches
Match Kernel Prototypes
3
Recorded Prototype Matches
Recorded Creation Matches
Recorded Argument Matches
Recorded Launch Matches
4
Bind created host cl_kernels by name to device prototypes
6
Bind created cl_kernel and prototype to launch(es) and argument assignments
5
Bind argument assignments to the next matching cl_kernel launch
Map of all kernel prototypes to their launches and arguments
7
For (each argument : in each launch : of each kernel) Check against prototype
Per-AST Actions
Cross-AST Actions

Fig. 5: 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 cross-file stages (2). After all per-file actions are completed the cross-AST portion of the tool iterates over saved information (3) to pair kernel function prototypes to their host-side representations (4) and their uses (5) & (6), then inspects them for type consistency between arguments and parameters (7).

required but improperly used. Therefore, we created a helper linter to track the variables that may contain thread-dependent values and to flag any conditionals that reference these variables.

The helper linter detects all variable declarations within a kernel. When a variable is assigned the return value from a global or local ID function, it is marked as *thread-dependent* by the match callback. Further, the linter tracks the flow of the thread-dependency through multiple assignments to mark all the *thread-dependent* variables. This helper is used by the next two linters to determine if conditional expressions have a risk of being thread-dependent.

d) Possibly-unreachable barriers: A barrier is a common and often necessary synchronization mechanism; however, it is also a potential source of deadlocks that can be hard to debug at runtime. (On FPGA platforms with long kernel compilation, the productivity 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.

The helper linter discussed in §II-C1c, is used to flag the "risky" (potentially thread-dependent) conditional expressions. It looks for any of the five primitive branch types (`if/else`, `switch/case`, `do/while`, `while` and `for`) that contain both a risky conditional expression and a call to a barrier function. The match callback 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 direct call to a global/local ID function or a reference to an ID-dependent variable. However, barriers can still be used safely (if inefficiently) in a thread-dependent manner, if all branches reliably encounter the barrier. ID-dependency tracking reduces the false-positive hits, but it does not address such valid in-branch barriers. To address this case, the linter includes a refinement step to reduce the false positives even further in `if/else` and `switch/case` statements, where all the possible paths execute the same number of barriers. (A full explanation is omitted for brevity.)

e) ID-dependent backward branches: In the context of FPGAs, OpenCL kernels are synthesized as hardware pipelines. For efficiency reasons, the Intel compiler tries to collapse branch statements into a single bit indicating if a functional unit is active [10]. This is straightforward for "forward" branches (without loops), resulting in flat control structures. However, looped (i.e., "backward") branches are more difficult to implement and can significantly reduce performance, and so they should be avoided. While some algorithms may require backward branches, many can be removed via algorithmic refactoring. Therefore, a linter has been designed to recognize the 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 more complex hardware-generated FPGA logic. Therefore, the linter needs to make use of the thread-dependency tracker that 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.

*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, our Multi-AST linter, the first to apply the multi-AST technique to 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: (1) a per-file phase that records information from each kernel or host file and (2) a cross-file phase that validates this information and reports potential problems. Figure 5 provides a conceptual overview of the flow from source code to diagnostics. Similar to FLOCL, the per-file recording phase is implemented as a set of ASTMatchers and callbacks that identify the code components of each stage of the OpenCL kernel invocation. It records kernel function declarations from device ASTs and the host-side kernel object (`cl_kernel`) binding, argument assignment, and launch functions 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 cross-file data structure that maps global externs to their real definition (i.e., the object file that will actually store the variable) for compatibility with CU2CL-MAST-generated applications.

After completing the per-file analysis on all device and host ASTs, the multi-AST portion of the tool combines information from each of the kernel invocation stages. First, each `cl_kernel` binding is checked for an exact match of the kernel name argument in its `clCreateKernel` call to the name of a prototype from a device AST. When such a match is found, FLOCL-MAST stores a "binding" of the prototype declaration from the device AST to the assigned `cl_kernel` in the host AST. Next, all kernel-argument assignment (`clSetKernelArg`) calls are "clustered" to kernel launches heuristically by finding the lexically-next kernel launch with a matching `cl_kernel` object. Lexically-next refers to the next launch encountered when scanning the source code from beginning to end, and it is highly effective as the argument specifications directly precede the kernel invocations in typical use cases. 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 its attached device prototype to ensure that (1) all arguments are provided and (2) each 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 copy-from void pointer arguments to `clSetKernelArg` must be resolved to a non-void pointee type, if possible. (Typically the address of variables to copy is taken and type-cast to void within the call, which is easy to resolve.) Pointer parameters 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 pointer to a `cl_mem`, and the latter is currently unsupported; both will be refined as future work.

This cross-file linter is useful as a demonstration of the complexity of multi-AST analysis in comparison to the per-file analysis provided by basic FLOCL. While both FLOCL and FLOCL-MAST use the same underlying ASTMatcher technology, the complexity of the analysis dramatically increases once the tool works across the ASTs of different translation units (TUs).

D. *CU2CL-MAST*

CU2CL [6] 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 formatting in the generated OpenCL source files. However, the original CU2CL was restricted to a single translation unit (TU) — it could not deal with the extra complexity of a multi-file application without tedious manual pre- and post-processing (e.g., merging the generated OpenCL boilerplate), and translations spanning files would be impossible to fully complete. Therefore, a primary contribution of this work is to apply the Multi-AST framework to CU2CL to support *whole-program* translation, resulting in the new "CU2CL-MAST" (Multi-AST).

Applying the multi-AST framework to CU2CL consisted of four stages. First, a *libTooling* interface was built to bootstrap and contain multiple instances of the existing per-file translation functionality, following the design detailed in §II-B. Second, to ensure completeness of translation, a restriction that translated files had either the `.cu` or `.cuh` extension was relaxed, since CUDA types and runtime calls can be used with and passed through `.c/.h` and `.cpp/.hpp` source files. Third, per-file boilerplate generation was modified such that common elements are produced by the cross-file layer and generated in a shared set of `cu2cl_util.c/h/cl` utility files. Furthermore, within each TU, the cross-file layer generates extern declarations for `cl_programs` and `cl_kernels` originating in other TUs to share newly-generated OpenCL state across object files. Together the first three stages provide a unified multi-file variant of CU2CL, which already simplifies post-translation development. However to demonstrate the new translation capability afforded by the multi-AST framework, a fourth stage providing a significant new translation was added. By providing a framework for translations to causally trigger further translations elsewhere in the codebase (in either the same or another AST), this novel stage tackles 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.

```
 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

Fig. 6: Propagation of the `cl_mem` type by CU2CL-MAST. Without the Multi-AST expansion, only the declaration of "buffer" on line 9 in Fig. 6a 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 17. These propagate horizontally through the alias assignments in 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.

The original per-file 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 from a pointer to a `cl_mem` [6]. However, as each AST was only walked once, if this variable was aliased — either by pointer assignment or through a function call — propagation to all possible aliases could not be guaranteed, causing a type inconsistency in other code regions that were not aware of the translation. (Figure 6a demonstrates a simple case of aliasing that confounded the original translator.) Thus, either the variable should not be translated and instead explicitly cast to a standard pointer variable (as in [7]), or the translation must be fully propagated to all affected regions. The latter is a substantially harder problem as it 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 types of the casted `cl_mem` variables are only apparent when used in OpenCL calls. Therefore, to fully propagate the type translation, all device pointer translation was raised to the cross-file layer, which has access to the entirety of the codebase and can track use/definition relationships to ensure propagation across aliases.

Primarily, cross-file translation relies on forward declarations from a shared header that resides on all ASTs and thus forms a common anchor between them. So, we modified the per-file portion of CU2CL to defer all `cl_mem` translations by storing them in a cross-file list. Further, as it traverse each AST it generates a map from all declared variables and functions to their uses for the cross-file layer to utilize. Actual translation is then implemented as a consumer of the deferred translation list. While each translation is performed, the reference map is checked for other uses to enqueue any further `cl_mem` translations that must then be performed (i.e., 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(s).) Figure 6 walks through a typical propagation cascade from a single initial translation site.

III. 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.

A. Target Workloads

1) ***iNFAnt****:* iNFAnt is a non-deterministic finite automata (NFA) regular-expression (regex) matching engine, realized in CUDA [12]. We used an optimized version of the algorithm [13] as a case study for a CUDA-to-FPGA translation pipeline using both CU2CL-MAST and FLOCL/FLOCL-MAST. We selected an NFA regex matching application 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.

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 [14], [15], PolyBench-ACC [16], Rodinia [17], [18], and SHOC [19] come from the high-performance computing (HPC) and general-purpose GPU community. CHO [20] targets OpenCL-on-FPGA development and captures current limitations of offline-compiled FPGA development. Across these five suites, there are 96 separate OpenCL applications.

B. CU2CL-MAST

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.) The viability of single-file translation was demonstrated in [6] and thus we focus only on multi-file translation here. CU2CL-MAST's translation improvement was evidenced in three categories:

a) Improved header file 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 these functions*. By expanding translation to all non-system files regardless of file type, CU2CL-MAST performs translation in 14 additional header files of the optimized iNFAnt, which includes portions of the interface that must have `cl_mem`s propagated through them.

b) 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 cross-file 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*. CU2CL-MAST generates the necessary extern declarations to make the optimized iNFAnt's sole kernel visible program-wide, which is necessary since it is written and used in separate source files. In programs with more kernels from several source files, the value of this automatic propagation across source files is further increased.

c) `cl_mem` propagation: §II-D 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 `cl_mem` propagation, which CU2CL-MAST successfully provides.

2) ***Translation Validation****:* To validate the auto-translated OpenCL application, we compared its output to the output from 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, and Intel Arria10 FPGA.

C. FLOCL and FLOCL-MAST

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,

TABLE I: FLOCL-MAST dramatically reduces the search space for finding mis-typed arguments in the separately compiled host-to-kernel interface.

Benchmark Suite	Evaluated SLOCs	Total Arg. Assignments	Checked and Correct	Checked and Flagged	Difficult False Positives	**True Positives**
CHO	12394	79 (100%)	73 (92%)	6 (8%)	0	**0**
OpenDwarfs	14574	336 (100%)	281 (84%)	10 (3%)	0	**4 (1%)**
PolyBench-ACC	9126	248 (100%)	229 (92%)	19 (8%)	0	**0**
Rodinia	34953	392 (100%)	215 (55%)	3 (1%)	0	**1 (<1%)**
SHOC	18829	349 (100%)	194 (56%)	37 (11%)	11 (3%)	**23 (7%)**
Total	89876	1418 (100%)	992 (70%)	75 (5%)	11 (1%)	**28 (2%)**

after the developer has invested significant time in compilation, placement, and routing. Table II provides a breakdown of how many hits each linter found across all the kernels in each suite. The following describes what each hit identifies:

Possibly unreachable barrier: Barriers that may not be encountered the same number of times by all threads because they lie in a thread-dependent conditional region. Figure 7 shows one of the hits in SHOC/SPMV.

ID-dependent backwards branches: Loop structures that may not be executed the same number of times by all threads and require inefficient pipelining on FPGAs.

Inefficient struct alignment: Structs that require multiple poorly-aligned or poorly-packed accesses to memory, reducing read/write efficiency in the kernel.

SWI-barrier: An unnecessary barrier in a single-threaded kernel. (The benchmarks largely consist of applications that are explicitly designed for multi-threaded NDRange execution rather than single-thread execution.)

TABLE II: Kernel semantic and performance risks identified by FLOCL

Benchmark Suite	Possibly Unreachable Barrier	ID-Dependent Backward Branch	Inefficient Packing / Alignment	Single Work-item Barrier
CHO	0	0	8/16	0
OpenDwarfs	2	13	0/4	0
PolyBench-ACC	0	2	0/0	0
Rodinia	3	62	10/21	0
SHOC	1	13	0/0	0
iNFAnt	0	4	2/2	0

```
 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 }
```

Fig. 7: 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.

*2) **FLOCL-MAST Results**:* We apply FLOCL-MAST to each OpenCL application, across the five benchmark suites, to detect any semantic inconsistencies between the host and device code. 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, even with production-ready benchmarks, FLOCL-MAST still found rare type-consistency bugs that *would not otherwise raise warnings at either compilation or runtime*.[4] The identified true positives represent potential loss of precision (of floating point or integer types) or signedness of integers, which would only exhibit themselves as broken data coming out of a kernel, which in turn, is laborious to both identify and address.

Table I provides some broad statistics on the benchmarks that we analyzed and shows how FLOCL-MAST pruned the search space to identify the type-consistency bugs. The benchmark suites contain almost 90k source lines of code (SLOC) across 256 primary source files (i.e., `.c`, `.cpp`, and `.cl` files that include OpenCL kernel or host/runtime code). Once ASTs are generated for each application (including all headers and macro/template expansions), FLOCL-MAST identifies 1418 host-side argument assignments via `clSetKernelArg` calls, and it safely eliminates 70% of all assignments from the search space as they have a compatible type between the host and device codes.

FLOCL-MAST flags around 5% of the total kernel argument assignments as potential host-device inconsistencies. A third of the inconsistencies represent true positives, where the type of the data passed to the kernel from the host is inconsistent with the type expected in the kernel and may result in data corruption due to the loss of precision or signedness. Such true positives are the "needles in the haystack" of 90k SLOC that FLOCL-MAST assists the developer in finding and resolving, thus demonstrating the viability of a static analysis approach to automate a significant part of validating the host-to-device interface of a large-scale codebase.

However, the remaining portion of the flagged inconsistencies are false positives. For example, SHOC makes use of dynamic `local` memory allocation, in which a kernel-side typed pointer parameter is mapped to a NULL pointer type on the host, preventing a consistency check and producing a difficult false positive to eliminate. In addition, runtime-dependent information that cannot be resolved at static-analysis time prevents a portion of the kernel arguments from being checked, a limitation of all static-analysis tools. Specifically, the necessary runtime information for checking such kernel arguments fits into one of the following categories: runtime-dependent `cl_kernel` launch, runtime-dependent `cl_kernel` creation, or runtime-dependent argument position. In many cases the runtime dependency is not necessary from a software engineering point of view, (e.g., a monotonically increasing variable storing the argument position for a single launch) and simple changes to the host code could remove the need for runtime-dependent information, and thus allow for static analysis.

[4]As the device code is separately-compiled from the host code, the traditional compiler warning for possible loss of precision cannot be applied during either host or device compilation. The OpenCL runtime copies data from the host to device via pass-by-reference from a void pointer, and thus cannot raise issues at runtime either.

IV. 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) [21], [22]. One approach for designing such tools is to construct the expensive program representations on demand using program slicing [22]. However, this demand-driven technique is limited to software debugging and comprehension tools, and it is not suitable for performing global source-code transformation or detecting cross-file 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 [23].

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 [8], [24]. Unlike other program representations, the size of the AST is linear in the size of the software program [21], [22], 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.

V. Conclusion

In this paper, we demonstrate a whole-program workflow for productive OpenCL programming and analysis on FPGAs and other heterogeneous OpenCL platforms. The workflow is composed of three tools: (1) a reconstructed and expanded Multi-AST variation of an existing CUDA-to-OpenCL translator, (2) *per-file* linters for OpenCL kernels on FPGAs, and (3) a *cross-file* linter for validating the OpenCL host-to-device kernel interface. The efficacy of the workflow is demonstrated by the translation and analysis of a substantial 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 uncovering opportunities to refine the proposed heuristics.

Previous work [25], [26] demonstrated the use of dynamically-linked libraries to hide the complexity of developing and running HPC applications on heterogeneous platforms. As such, the proposed workflow can be further expanded to improve the productivity of end users by generating host-callable, encapsulated modules for the OpenCL kernels to eliminate the need for developing the host-to-device interface or dealing with the interoperability across different devices.

References

[1] A. Munshi, "The OpenCL Specification, 2008," *Chronos OpenCL Working Group*.

[2] ——, "The OpenCL Specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 2009, pp. 1–314.

[3] NVIDIA, "Compute Unified Device Architecture Programming Guide," 2007.

[4] G. Martinez, M. Gardner, and W. Feng, "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, 2011, pp. 300–307.

[5] P. Sathre, M. Gardner, and W. Feng, "Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation," in *Parallel Processing Workshops (ICPPW), the 41st International Conference on*. IEEE, 2012.

[6] M. Gardner, P. Sathre, W. Feng, and G. Martinez, "Characterizing the Challenges and Evaluating the Efficacy of a CUDA-to-OpenCL Translator ," *Parallel Computing*, vol. 39, no. 12, pp. 769 – 786, 2013.

[7] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee, "Bridging OpenCL and CUDA: A Comparative Analysis and Translation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, pp. 82:1–82:12.

[8] Clang, "A C Language Family Frontend for LLVM," 2013.

[9] The Clang Team. Extra Clang Tools 8 Documentation. LLVM/Clang. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/

[10] Intel, "Intel FPGA SDK for OpenCL, Best Practices Guide." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en˙US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

[11] Altera, "Altera FPGA SDK for OpenCL, Best Practices Guide." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en˙US/pdfs/literature/hb/opencl-sdk/aocl-optimization-guide.pdf

[12] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA Pattern Matching on GPGPU Devices," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 20–26, Oct. 2010.

[13] X. Yu and M. Becchi, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," in *Proceedings of the ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, 2013, pp. 18:1–18:10.

[14] K. Krommydas, W. Feng, M. Owaida, C. D. Antonopoulos, and N. Bellas, "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*, June 2014, pp. 153–160.

[15] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 Dwarfs: A Work in Progress," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 291–294.

[16] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-level Language Targeted to GPU Codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[18] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.

[19] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74.

[20] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: ACM, 2015, pp. 10:1–10:10.

[21] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[22] D. C. Atkinson and W. G. Griswold, "The Design of Whole-program Analysis Tools," in *Proceedings of the 18th International Conference on Software Engineering*, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 16–27.

[23] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," New York Univ. Comput. Sci. Dept., Tech. Rep., 1978.

[24] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," in *The BSD Conference*, 2008, pp. 1–2.

[25] A. E. Helal, P. Sathre, and W. Feng, "MetaMorph: A Library Framework for Interoperable Kernels on Multi- and Many-core Clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, pp. 11:1–11:11.

[26] A. E. Helal, W. Feng, C. Jung, and Y. Y. Hanafy, "AutoMatch: An Automated Framework for Relative Performance Estimation and Workload Distribution on Heterogeneous HPC Systems," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 2017.