MetaCL: Automated “Meta” OpenCL Code Generation for High-Level Synthesis on FPGA

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

Atharva Gondhalekar  
Dept. of ECE  
Virginia Tech  
athanval@vt.edu

Mohamed Hassan  
Dept. of ECE  
Virginia Tech  
mwasfy@vt.edu

Wu-chun Feng  
Depts. of CS & ECE  
Virginia Tech  
feng@cs.vt.edu

Abstract—Traditionally, FPGA programming has been done via a hardware description language (HDL). An HDL provides fine-grained control over reconfigurable hardware but with limited productivity due to a steep learning curve and tedious design cycle. Thus, high-level synthesis (HLS) approaches have been a significant boon to productivity, and in recent years, OpenCL has emerged as a vendor-agnostic HLS language that offers the added benefit of interoperation with other OpenCL platforms (e.g., CPU, GPU, DSP) and existing OpenCL software. However, OpenCL’s productivity can also suffer from tedious boilerplate code and the need to manually coordinate the host (i.e., CPU) and device (i.e., FPGA or other device). So, we present MetaCL, a compiler-assisted interface that takes OpenCL kernel functions as input and automatically generates OpenCL host-side code as output. MetaCL produces more efficient and readable host-side code, ensures portability, and introduces minimal additional runtime overhead compared to unassisted OpenCL development.

Index Terms—Code Generation, OpenCL, CPU, FPGA, GPU, HPC, Programmability, Productivity, Portability, Clang, LLVM, MetaCL

I. INTRODUCTION

FPGAs are gaining traction in the high-performance computing (HPC) community to accelerate a wide spectrum of applications via their configurable computing capability and superior power efficiency. However, poor programmability impedes the integration of FPGAs into HPC. Using hardware descriptive languages (HDLs) to customize an FPGA’s configurable fabric requires skill and knowledge of the underlying hardware architecture. While HDLs allow fine-grained control, they have a steep learning curve that limits their appeal and accessibility to software developers and domain scientists.

In contrast to programming a (fixed) hardware ASIC, such as a CPU or GPU, an FPGA programmer needs to optimize not only for performance (or power) but also for area, as a given computation must fit logic, routing, and internal storage into a limited silicon area. This complexity requires additional design iterations, which follow the laborious and time-consuming cycle of simulation, testing, functional verification, placement, and routing. High-level synthesis (HLS) tools partially address this complexity by allowing developers to write code at a higher level of abstraction. For example, the vendor-neutral OpenCL heterogeneous computing standard [1] has emerged as a portable C-based HLS approach, supported by FPGA-native compilers and runtime systems such as Xilinx’s SDAccel [2] and Intel’s FPGA SDK for OpenCL [3].

OpenCL has been used for FPGA acceleration across a range of applications [4]–[8], thus demonstrating the viability of such an HLS approach. The benefits of OpenCL to FPGA developers are multi-fold: (1) FPGA functions (device kernels in OpenCL terminology) can be written in a variant of C rather than a low-level HDL; (2) FPGA devices can be transparently reconfigured with new kernels; and (3) OpenCL applications are easy to port between supported devices.

However, the abstractions in the OpenCL programming environment also introduce challenges that limit ease of use. Specifically, OpenCL separates CPU-side “host” code from the FPGA’s “device” code and requires manually managing the interaction between them at run time, which is tedious and error-prone. Furthermore, to be portable to many architectures and software domains, the powerful OpenCL C host API is necessarily verbose, often requiring many lines of API interaction (commonly known as “boilerplate”) to realize even a small computation on a device. Fig. 1a provides an example of this boilerplate code for matrix multiplication in OpenCL, where after eliding device and kernel source management, 50+ lines of host code are still required to run just eight lines of code on the device, as shown in Fig. 1b. These OpenCL API calls are needed to configure the runtime environment and interact with the device kernels but impose development overhead to create and maintain.

To complicate matters, part of the OpenCL boilerplate code scales in size, relative to the number and complexity of device kernels; other parts must be rewritten for different devices; and all of this code should be error-checked, which is often elided for development efficiency. For example, Table I lists the typical OpenCL calls necessary to run code on a device and their relative frequency, where $D$ is the number of devices to utilize in a computation, $P$ is the number of OpenCL kernel programs (typically a .cl or .aocx file), $K$ is the number of kernels, and $A$ is the average number of arguments per kernel.

To simplify OpenCL development and further improve FPGA productivity, this paper presents MetaCL, a compiler-assisted code-generating tool that automatically generates the extensive OpenCL host boilerplate needed (e.g., Fig. 1a) to...
initialize and run a kernel on an FPGA or other device, directly from the OpenCL kernel implementation (e.g., Fig. 1b). The lines highlighted in gray in Table I correspond to the \(O(P + K \times A)\) lines of boilerplate code that MetaCL auto-generates. Using MetaCL’s auto-generated code eliminates human error and saves time by removing the burden of creating and managing boilerplate as kernels evolve. Moreover, by capturing boilerplate in auto-generated functions, MetaCL significantly reduces the total code that must be manually written, while increasing robustness with integrated error checking.

The contributions of this paper are summarized below.

- An automated “meta” OpenCL (MetaCL) code generator for host-side boilerplate code.
- A rigorous characterization of how MetaCL significantly improves developer productivity on two OpenCL applications: BabelStream (streaming memory benchmark) and SNAP (particle-transport proxy application).

A quantitative comparison that shows that the improved developer productivity via automated code generation of MetaCL delivers comparable performance to manually-written OpenCL host-side code.

II. RELATED WORK

The two major producers of FPGAs, Xilinx and Intel, now have their own respective OpenCL toolchains for FPGA [2], [3]. However, these vendor tools do not go much beyond the OpenCL standard and leave the burden of managing OpenCL state or invoking OpenCL kernels to the FPGA developer. Another prominent HLS tool, available as a software IDE for programming FPGAs, is LegUp [9]. Like MetaCL, LegUp targets efficient HLS for the FPGA, but it does so without regard to the complexity of host-code programming.

MetaCL stands apart from existing OpenCL for FPGA work by minimizing the distance between manually-written code and the standards-compliant OpenCL it produces. Developers write pure OpenCL kernels and interact with the generated static code via OpenCL data types. Our approach retains portability to compliant OpenCL implementations, thus supporting interoperaion with other OpenCL tools and libraries. This interoperaion is crucial to utilizing advanced features, such as primitives for CPU-to-device transfers of OpenCL device buffers over a Message Passing Interface (MPI), like those provided by the MetaMorph OpenCL backend [10].

III. AUTOMATED CODE GENERATION VIA METACL

MetaCL’s automated generation of code supports productive OpenCL development in three ways. First, to reduce developers’ manual effort, the \(O(P + K \times A)\) lines of boilerplate associated with OpenCL program/kernel management and invocations are hidden behind a robust host-to-device interface. This interface automatically utilizes the refined device management provided by the open-source MetaMorph OpenCL backend to further hide the \(O(D)\) device-management boilerplate. Figure 2 shows how the manual effort of OpenCL development is offloaded in a MetaCL-driven development approach. Second, by abstracting away this boilerplate code, a MetaCL-generated interface improves the clarity (and thus maintainability) of the host application. Finally, by using the device kernel specification to statically enforce the host-to-device kernel interface at host compile time (i.e., via make automation of MetaCL), runtime errors due to interface inconsistency
(like example, scalar kernel parameters are exposed directly to the
top wrapper’s prototype to utilize pass-by-value semantics.
This in turn supports static-compile time type checking of
arguments provided to those parameters that the void* pass-
by-reference semantics of clSetKernelArg do not.

(a) Traditional OpenCL development requires users to manually
manage device and kernel boilerplate, which can be substantial, in
addition to the actual application logic and kernel implementations.

(b) MetaCL allows OpenCL application developers to focus on
kernels and application logic by abstracting away kernel and device
management. MetaCL generates simple-to-use kernel wrappers and
robust initializers.

Fig. 2: A MetaCL-driven approach to OpenCL development
offers an alternative which typically no longer requires man-
ually managing the OpenCL runtime’s devices and kernels.

MetaCL is implemented as a “compiler-like” tool that
ingests kernel source files written in OpenCL C and relies
on the robust parsing and semantic analysis of a compiler to
produce the corresponding host-callable interface, which the
host code can then reference. By leveraging the compiler’s
semantic analysis, MetaCL ensures all elements of the kernel
interface are faithfully represented on the host. For example
when a user-defined type is used as a kernel parameter,
MetaCL recursively identifies all other user-defined data types
and imports them to the host code, mapping OpenCL data
types to their host API equivalents in the process (Fig. 3).

1/ ← myKernel.cl ← /
2 struct foo {
3 float4 pos;
4 char2 ids;
5 };
6 struct bar {
7 float data;
8 unsigned int flags;
9 };

(a) Device structs that use
OpenCL device types must be
mapped to the host
types are autogenerated

Fig. 3: MetaCL automatically imports user-defined data types
to ensure consistency.

A. Clang/LLVM

MetaCL is realized as a ClangTool, leveraging the source
analysis and manipulation components of the Clang/LLVM
compiler project [11], [12]. MetaCL is a standalone binary
that uses Clang as a library, requiring no modifications, which
makes MetaCL easy to package and deploy alongside Clang
installations. MetaCL heavily leverages the semantic
information that Clang/LLVM provides about the parsed OpenCL
kernel code to create a reliable and concise host interface.

B. MetaCL Kernel Parsing and Traversal

Like a compiler, MetaCL is invoked from the command line
and must be provided a set of kernel implementation files and
any typical preprocessor arguments they need. Clang prepro-
cesses, parses, and semantically analyzes the input file(s) to
generate abstract syntax trees (ASTs), which MetaCL traverses
to produce the host interface. MetaCL identifies all OpenCL
kernel functions via Clang’s ASTMatchers interface. For
each match, the kernel AST nodes are inspected for their
parameter lists and any attributes that may require special
enforcement on the host. If a parameter in the list has a user-
defined type, its definition and those of any other user-defined
types upon which it depends are also traversed recursively for
import.

C. Code Generation

The execution of MetaCL follows the AST traversal order
due to the use of ASTMatchers, but boilerplate code is
generated and cached according to where it will be placed in
the generated interface. Generally, these boilerplate elements
must be created either:

• Once per input file: O(P), corresponding to a
  cl_program
• Once per kernel function: O(K), corresponding to a
  cl_kernel, including both initialization/deconstruction
  and the wrapper
• Once per kernel parameter: O(K × A), corresponding to
  clSetKernelArg calls and the associated parameter
  in the generated wrapper
• Once per output file: O(1) to O(P), corresponding to
top-level elements of the generated interface, such as
internal management data structures, imported user types,
and initialization, registration, and deconstruction

MetaCL implements a simple code cache for generated el-
ments, and then, after all ASTs are completely traversed, it
serializes the final output. Depending on the options specified,
this output consists of one or more pairs of .c and .h files
which contain the generated host-to-device interface — these
pairs are hereafter referred to as a generated module.

D. MetaMorph OpenCL Backend for MetaCL

OpenCL applications must explicitly manage compute de-
vices and the OpenCL platforms they reside on. This boiler-
plate scales with the number of devices the application uses,
but by language design, it should be agnostic of the number
and interface of kernels. To support modular and plugin-based
software design patterns, we leave this boilerplate isolated
from that generated by MetaCL and leverage the existing
open-source implementation from MetaMorph’s [10] OpenCL.
MetaMorph is used as it implements a similarly thin C layer between user code and OpenCL, is open-source, and offers additional developer conveniences, such as transparent MPI exchanges of OpenCL buffers.

MetaMorph’s construction is already detailed in [10], but the existing OpenCL backend was improved and extended to support MetaCL-generated modules. These changes allow for:

- Dynamic Module Registration: MetaCL-generated modules need a mechanism to communicate state and pointers to auto-generated functions to MetaMorph.
- Automatic Module (Re-)Initialization and Deconstruction: MetaMorph automatically initializes known modules’ kernels when a device is selected and releases them when the device is retired.

With these changes, MetaCL-generated modules are capable of lazy self-registration and initialization at first wrapper call and often will not require any explicit initialization.

IV. PRODUCTIVITY AND PERFORMANCE EVALUATION

As a proxy for developing new FPGA OpenCL applications with the MetaCL-supported approach, we apply MetaCL to the kernels from two existing OpenCL applications. We then evaluate the performance of the original OpenCL code and the MetaCL-generated OpenCL code on an Intel Arria 10 FPGA.

First, we explain the applications and the experimental setup used for evaluation. We then discuss how the MetaCL-assisted development approach improves developers’ productivity with specific examples from the evaluated applications. Finally, both the existing OpenCL-only and new MetaCL-assisted versions of the applications are profiled on the Intel Arria 10 device in order to understand the performance impact of a MetaCL-assisted development process.

A. Applications Benchmarks: BabelStream and SNAP

BabelStream [13] is an evolution of the STREAM [14] memory bandwidth benchmark and provides numerous implementations to promote cross-platform analysis. The OpenCL version of BabelStream is written using the OpenCL C++ host-side API [15]; however, the MetaCL-generated interface expects the more common OpenCL C API’s data types. Utilizing the C++ API’s accessor operators, such as those in Table II, allows device state to be shared between the manually-written code and MetaMorph, while kernels were auto-managed by the MetaCL-generated interface and manual data management remained in C++.

<table>
<thead>
<tr>
<th>OpenCL C API Type</th>
<th>OpenCL C++ API Type</th>
<th>Accessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_command_queue</td>
<td>cl::CommandQueue myQueue</td>
<td>myQueue()</td>
</tr>
<tr>
<td>cl_context</td>
<td>cl::Context myContext</td>
<td>myContext()</td>
</tr>
<tr>
<td>cl_mem</td>
<td>cl::Buffer myBuffer</td>
<td>myBuffer()</td>
</tr>
</tbody>
</table>

SNAP [16] is a proxy application developed at Los Alamos National Laboratory. It is based on their discrete ordinate neutral particle transport code, PARTISN [17]. Particle-transport applications can be difficult to port to FPGAs due to the large memory footprint of a global timestep. The discretization used creates a Jacobi computational pattern with an upwind dependency that sweeps across the domain, thus the Deakin GPU implementation [18] used in this work utilizes wavefront parallelism to maximize the number of busy work items during each sweep. The host implementation is OpenCL C and invokes nine OpenCL kernels at various execution stages. Most kernels are allowed to run asynchronously, and a secondary cl_command_queue is used to synchronize via OpenCL markers and events. This motivates the inclusion of queue, async, and returned event pointer parameters to the generated API to support multi-queue applications, non-blocking kernels, and event-based synchronization.

B. Experimental Setup

The testing environment consists of dual hex-core Intel Xeon Gold 6128 CPUs operating at 3.40 GHz with 192-GB RAM, running Ubuntu 18.04.3 with the 4.15.18 Linux kernel. Attached via PCIe is an Intel Programmable Acceleration Card (PAC) with an Arria 10 FPGA (10AX115S2F451SGES) and OpenCL kernels are compiled using version 19.3 of Intel’s OpenCL SDK for FPGA. This device has 1.15 million logic elements, over 5 million memory bits, 1518 DSP blocks, and 2713 RAM blocks.

C. Productivity Improvement

MetaCL improves the productivity of OpenCL host-code development in three ways. First, it reduces the total host code that the developer must manually write by abstracting boilerplate behind a simple kernel interface. Second, the removal of distracting boilerplate improves code clarity, making the construction and logic of the application more apparent, and thus easier to maintain. Finally, by being lightweight and easily integrated with a build environment, MetaCL supports rapid regeneration of the host-to-device interface. This allows the kernel code’s interface to be enforced at host compile-time rather than only at runtime as is typical for OpenCL.

1) Code reduction: To evaluate MetaCL’s productivity improvement we use source lines of code (SLOC) as our primary quantitative metric and compare the manually-written SLOC required to implement equivalent applications both with and without MetaCL. Manually-written code is divided into two categories: (1) user logic and (2) OpenCL and/or MetaCL APIs. Table III provides an overview of how our MetaCL-assisted approach saves programming effort related to managing OpenCL kernels and devices. This approach does not address automation of buffer creation or transfer patterns, since they are typically application- or problem-specific, which make up 30.6% and 40.1% of the remaining boilerplate in BabelStream and SNAP, respectively, after MetaCL is applied. Despite the remaining data management, by simplifying kernel and device management, our approach eliminates 41% and 54.1% of all OpenCL-related boilerplate, respectively. This simplifies the resulting code structure, allowing removal of other now-redundant code elements such as macros, conditionals, and exception handling (i.e., “Non-API Host Lines” in Table III) contributing to code clarity.
a) **BabelStream**: The OpenCL C++ API reduces code bulk over the C API but in exchange requires more complicated C++ program elements. As BabelStream is originally written with the C++ API, there is less room for improvement from MetaCL, but it still significantly reduces total manually-written boilerplate and produces an API that is interoperable with the existing C++ code. These results show that MetaCL can provide a sizable reduction in SLOC even compared to the C++ API, without having to utilize more complicated C++ elements.

b) **SNAP**: SNAP makes use of OpenCL’s main C API, in particular utilizing 139 explicit calls to clSetKernelArgS. Further, as originally written, SNAP’s nine kernels were in a single cl_program that would not fit on the Arria 10 FPGA and had to be manually partitioned into two. While Intel’s OpenCL runtime transparently reconfigures the devices with these cl_programs as needed, the additional program requires its own boilerplate to load, create, and release. However, this additional bulk is entirely hidden by the MetaCL-generated API and did not need to be repeatedly hand-modified during the manual kernel-fitting process. After migration to the generated interface, the number of lines used to launch kernels is reduced from 184 to 69 and management lines from 67 to a mere 19. Overall, we observe a 54.1% reduction in lines spent on boilerplate and a 10.2% reduction in lines across the entire host code. These results highlight that MetaCL can substantially reduce the effort to accelerate a large code with OpenCL, while hiding extra complexities of the FPGA platform.

2) **Code Clarity**: One of the negative elements of unassisted OpenCL development is that the extensive boilerplate requirements distract from the the application’s logic by cluttering the code, which greatly complicates code maintenance. Thus a secondary benefit of MetaCL-assisted OpenCL development is that this distraction is dramatically reduced, as MetaCL’s host-like interface is simply more concise and reminiscent of a host function launch. Fig. 4 contrasts the sheer bulk typically required to run SNAP’s main waveform kernel (4a), with that required when using the MetaCL-generated interface (4b).

3) **Error Reduction**: Another productivity improvement offered by MetaCL is the reduction of host-to-device errors that are not discovered until during or after runtime. Such errors include mistyped, missing, and excessive kernel parameters, as well as inconsistent struct definitions between the host and the device. Rather than the host-compiler being oblivious to the types and kernel prototypes of the device code, as is typical of just-in-time-compiled OpenCL kernels, instead MetaCL faithfully represents these components in the host interface, making them available for static analysis during host compilation. As a further value, MetaCL also incorporates all of Clang’s semantic analysis, which is often more strict than vendor kernel compilers. Hence, kernel compilation errors are implicitly “double-checked” by adding MetaCL to the design cycle. This rapid error detection is reliant on integrating MetaCL with an application’s existing build system and runs in mere seconds. Further, with a typical Makefile or similar system, integration is as simple as creating additional build targets for and dependencies on the MetaCL-generated files.

### Table III: Reduction in manually-written host code using MetaCL-generated interfaces

<table>
<thead>
<tr>
<th>Host Task</th>
<th>Original ⇒ MetaCL</th>
<th>Original ⇒ MetaCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BabelStream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Mgmt.</td>
<td>45 ⇒ 21</td>
<td>67 ⇒ 19</td>
</tr>
<tr>
<td>Kernel Mgmt.</td>
<td>23 ⇒ 13</td>
<td>184 ⇒ 69</td>
</tr>
<tr>
<td>Data Mgmt.</td>
<td>15 ⇒ 15</td>
<td>59 ⇒ 59</td>
</tr>
<tr>
<td>Total</td>
<td>83 ⇒ 49</td>
<td>380 ⇒ 147</td>
</tr>
<tr>
<td>Savings</td>
<td>41%</td>
<td>54.1%</td>
</tr>
<tr>
<td>Non-API Host Lines Removed</td>
<td>19 ⇒ 0</td>
<td>43 ⇒ 0</td>
</tr>
<tr>
<td>Manually-written Host Lines</td>
<td>Entire Host Code</td>
<td>720 ⇒ 667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2120 ⇒ 1904</td>
</tr>
<tr>
<td></td>
<td>Savings</td>
<td>7.36%</td>
</tr>
</tbody>
</table>

(a) SNAP’s original sweep kernel invocation (manually-written). Note that 22 additional argument assignments are omitted and all arguments must have their position and type manually validated

<table>
<thead>
<tr>
<th>Host Task</th>
<th>cl_int err: cl_set_kernel err: 2 dimensional kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>BabelStream</td>
<td>cl_int err: 2 dimensional kernel</td>
</tr>
<tr>
<td>SNAP</td>
<td>cl_int err: 2 dimensional kernel</td>
</tr>
<tr>
<td>Device Mgmt.</td>
<td>cl_set_kernel err: first dimension: number of angles + number of groups</td>
</tr>
<tr>
<td>Kernel Mgmt.</td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
<tr>
<td>Data Mgmt.</td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
<tr>
<td>Total</td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
<tr>
<td>Savings</td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
<tr>
<td>Non-API Host Lines Removed</td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
<tr>
<td>Manually-written Host Lines</td>
<td>Entire Host Code</td>
</tr>
<tr>
<td></td>
<td>Savings</td>
</tr>
<tr>
<td></td>
<td>cl_set_kernel err: second dimension: number of cells in plane</td>
</tr>
</tbody>
</table>

(b) SNAP’s simplified sweep kernel invocation using MetaCL’s auto-generated wrapper, which eliminates the 27 lines used to manually define arguments and check the resulting return code. A single missing or mis-typed argument can now be automatically detected by the host compiler

Fig. 4: Contrast between the manually-written code necessary to run SNAP’s main waveform kernel, “sweep_plane.”

D. Performance and Analysis

While MetaCL significantly improves the productivity of OpenCL developers, it is critical that the additional convenience does not compromise application performance. So, we compare the FPGA’s achieved performance to itself — with...
and without MetaCL. To evaluate the performance effects of using a MetaCL-generated interface we compare three metrics:

1) The whole-program wall-clock time, measured by the `time` command
2) The device runtime of each kernel, queried from OpenCL's `cl_event` API
3) The wall-clock overhead for running each kernel, derived by finishing the OpenCL queue(s) and starting a wall clock before the argument assignment, enqueuing the kernel, immediately finishing the queue(s) again to force the kernel to execute, stopping the timer, and subtracting the `cl_event`-based device runtime.

The whole-program time is the first check to ensure that total time to solution is not significantly impacted by the introduction of MetaCL. This measurement is taken without any of the added flushing and profiling code of the latter two, and thus any kernels or transfers that are allowed to run asynchronously in the original code do so in both it and the MetaCL-assisted version. The `cl_event`-based timing ensures that we have not significantly tampered with the semantics of the kernel execution or invocation by utilizing the MetaCL-generated kernel wrappers. Finally, by analyzing the change in kernel launch overhead, we can examine any scaling cost of the MetaCL-generated interface and work to keep it lightweight.

a) BabelStream: BabelStream is invoked with the default problem size, corresponding to three arrays of approximately 268.4 MB of double-precision values. Each run performs a short initialization phase and then runs each of the benchmark kernels 10 times; for each trial, the arithmetic mean of their kernel runtime and overhead are reported. A comparison of the MetaCL-assisted whole-program runtime relative to the original OpenCL is shown in Fig. 5. The slight 1% speedup when using the MetaCL-generated interface is well within one standard deviation, so the total time to solution is unchanged. The kernel performance is also effectively unchanged; as Fig. 6a shows, performance is within ±0.006%. As expected, the average overhead of using the MetaCL-generated interface increased slightly (Fig 6b). However, in absolute terms, the overhead is only microseconds per launch and negligible to the overall application performance. For completeness, Tab. IV presents the FPGA utilization summary of BabelStream.

The simple streaming kernels easily co-occupy the device without necessitating reconfiguration.

b) SNAP: We evaluate SNAP using a dataset that simulates a 1×1×100 cell “pencil” region using a grid of 4×4×400 cells, up to 100 outer iterations per global timestep, and five inner iterations per outer1. The test case terminates in a single global timestep consisting of a total of four outer iterations and 334 inner iterations. Kernels are called a variable number of times, thus each trial reports the aggregate runtime and overhead as well as the number of calls so that the averages can be derived. As a complete proxy application rather than simple benchmark, SNAP’s kernels are significantly more complex, involving data-dependent branching and complex global memory indexing. Consequently, several have large RAM requirements that required partitioning the kernels into two `cl_programs`, as shown in Tab. IV. The host application was manually analyzed to determine the kernels that made up the inner “hot-loop” to try to minimize reconfiguration overhead. These kernels were then combined into one program with the remaining outer kernels collected in another program.

The total time to solution remains effectively unchanged, as shown in Fig. 5. However, when forcing synchronous execution to profile the kernels, the total time to solution increases by 30 – 40 seconds in both the MetaCL-assisted and original, indicating the importance of MetaCL’s support for these asynchronous execution patterns.

The aggregate kernel runtimes across the global iteration are unchanged (Fig. 7a) and dominated by the reduction kernels, 88k calls to the wavefront sweep and 17.7k buffer resets, which represent optimization targets for future work. As Fig. 7b shows, the only perturbation to average kernel runtime

1titan/gpu/00001-snap.in
Fig. 7: Kernel and overhead profile for SNAP, showing that MetaCL again has no effect on kernel performance, but introduces tens of microseconds of extra launch overhead that is dwarfed by the ∼4s reconfiguration cost to swap out cl_programs.

is 1% on the inconsequentially-fast `calc_dd_coeff` kernel.

Finally, the change in average launch overhead, shown in Fig. 7c is < 1%, which shows that with more complicated kernels, any overhead added by MetaCL is amortized by the increased cost of invocation within the OpenCL runtime itself. Further, average overheads in Fig. 7c spike to ∼4s during the reconfigurations when switching between the inner and outer `cl_programs`, which contribute about half of the profiling variants’ total time to solution. Re-designing the kernels to reduce RAM usage and allowing co-occupation of the device could significantly improve performance and will be explored as future work. Overall, MetaCL has helped to eliminate over 50% of the boilerplate writing effort with negligible impact on performance.

V. CONCLUSION

In this work we have debuted MetaCL, an automated “meta” OpenCL code generator for FPGA host-side code. Using MetaCL, programmer productivity is greatly enhanced, saving the developer from manually writing and maintaining significant portions of the required OpenCL host’s device and kernel management and invocation boilerplate. For the BabelStream memory-bandwidth benchmark and SNAP proxy application, MetaCL results in up to 54.1% less manually-written boilerplate, reducing total manually written host code by up to 10%. Moreover, this effort-saving scales with the number of kernels and sizes of their parameter lists. MetaCL achieves this while delivering comparable total time to solution and only minor changes to the performance and launch overhead of individual kernels. In addition, the generated code automatically detects and insulates the application developer from several runtime-discoverable errors and results in a cleaner and more concise user application. So, MetaCL provides a wholly more-productive approach (i.e., “software that writes itself”) to programming an OpenCL host application than manual implementation, and it does so without the compilation or development weight of a larger heterogeneous computing framework.

ACKNOWLEDGMENTS

Experimental system access was provided as part of the Intel OneAPI DevCloud public beta.

CODE REPOSITORIES

The software developed and evaluated for this work is available as open source. MetaCL is integrated with the MetaMorph framework available from https://github.com/vtsynergy/MetaMorph. Forks of BabelStream and SNAP used for evaluation can be found at https://github.com/vtsynergy/MetaCL-BabelStream and https://github.com/vtsynergy/MetaCL-SNAP, respectively.

REFERENCES


