CLOP: A Multi-stage Compiler to Seamlessly Embed Heterogeneous Code

Dmitri Makarov   Matthias Hauswirth

Università della Svizzera italiana, Lugano, Switzerland

{firstname}.{lastname}@usi.ch

Abstract

Heterogeneous programming complicates software development. We present CLOP, a platform that embeds code targeting heterogeneous compute devices in a convenient and clean way, allowing unobstructed data flow between the host code and the devices, reducing the amount of source code by an order of magnitude. The CLOP compiler uses the standard facilities of the D programming language to generate code strictly at compile-time. In this paper we describe the CLOP language and the CLOP compiler implementation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – compilers, code generation

General Terms Languages, Generative Programming

Keywords Heterogeneous Programming, Embedded Languages, Staging

1. Introduction

Most modern computer systems are heterogeneous: they combine general purpose host processors with various special purpose compute devices. Languages such as OpenCL [5] and CUDA [9] enable the development of programs for such systems, but they explicitly separate host and device code, require extensive glue code, complicate program understanding and testing, and make it difficult to tune performance without extensive refactoring.

In this paper we introduce CLOP, a platform that aims to address the above issues. CLOP allows the seamless embedding of compute kernels in heterogeneous applications. CLOP is implemented in the D programming language. It uses a multi-stage approach to compilation, based on D's support for string mixins, compile-time function execution, and compile-time reflection. CLOP is an embedding of an external language (OpenCL) based only on the standard facilities of a host language (D). CLOP provides high-level abstractions that it expands into low-level code of the host and embedded languages.

CLOP lifts the abstraction level by eliminating the considerable amount of boilerplate code usually necessary in OpenCL by seamlessly connecting embedded and host code. CLOP allows the use of host variables in embedded code, and it automatically generates the appropriate data transfers between host and device. It provides synchonization patterns that determine the correct generation of efficient parallel computations. CLOP seamlessly integrates error messages across embedded and host language. To improve maintainability of the CLOP platform, CLOP exploits D's unit-testing support to allow efficient debugging of CLOP components.

2. Background on D and OpenCL

D is a modern general-purpose systems programming language with C-like syntax and static type system. It combines features of imperative, object-oriented, and functional programming. Some of the features that D has had from the ground up, find their way into current and future versions of C++. Although there is no industry backing for D at the moment, several companies have started using D for their production code. There exist several open-source compilers for D that produce optimized native machine code. D programs can be linked with libraries of compiled C or C++ code. We chose D as a host language because of its support for embedded language development.

String mixins generate executable code from a string. The construct `mixin` (<*D code*>) takes as an argument a valid D code fragment that must evaluate at compile-time to a constant string. The text contents of the string must be *compilable* as a valid expression, a declaration, or a statement list also in D. The text is inserted in place of the `mixin` construct and the compilation is resumed from the beginning of the inserted code. The inserted code can contain other `mixin` constructs that will be evaluated at compile time as well. For example, the program

```
1 void main() {
2   import std.format, std.stdio;
3   const s = "auto x = ", v = `mixin (format("%d", 3 + 7));`;
4   mixin (s ~ v);
5   writeln("x = ", x);
6 }
```

generates the output "`x = 10`" and is equivalent to the program

```
1 void main() {
2   import std.format, std.stdio;
3   const s = "auto x = ", v = `mixin (format("%d", 3 + 7));`;
4   auto x = 10;
5   writeln("x = ", x);
6 }
```

In line 4 of the first version of the program we concatenate two strings, represented by the variables `s` and `v`. The concatenation is done by the D compiler at compile time and its result is a valid declaration `auto x = mixin (format("%d", 3 + 7));` The declaration replaces the `mixin` construct at line 4. It contains another `mixin` construct, which in turn is replaced by the result of calling the function `format("%d", 3 + 7)`, i.e. a string `"10"`, which when *mixed in* becomes an integer literal `10`. The function's

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3687-1/15/10...
http://dx.doi.org/10.1145/2814204.2814211

arguments are evaluated and the function call is executed at compile time. The `mixin` construct allows to invoke the CLOP compiler when an application, that embeds CLOP code, is being compiled.

Token strings are string literals that open with characters `q{`, close with `}`, and must contain only valid D tokens. In the previous code example instead of writing

```
const v = `mixin (format("%d", 3 + 7));`;
```

we could have written

```
const v = q{mixin (format("%d", 3 + 7));};
```

Both declarations are semantically equivalent, but the former initializer string is parsed as a single token and the latter is tokenized. Token strings allow to embed smoothly foreign syntax code in D program source, as long as the code's lexical tokens are consistent with the D language lexical rules. CLOP programmers can use token strings to make it more convenient to work with external (CLOP) code embedded in an application's source.

Compile-time function execution. Every D compiler must have a built-in interpreter that can evaluate D expressions and almost arbitrary functions at compile-time. To be eligible for compile-time execution a function must be side-effect free, its source code must be available to the compiler at the time it evaluates a call to the function, and the function cannot reference any global or local static variables. A function is executed at compile time if a call to the function appears in the context where its return value must be known at compile time, e.g. when it is a parameter of a `mixin` construct. The semantics of a function must be the same whether it is executed at compile-time or at run-time. CLOP exploits compile-time function execution to implement compile-time staging by parsing embedded CLOP code, and then evaluating the corresponding abstract syntax tree to generate the target code.

Compile-time reflection is realized through *traits*, special extensions to the language that allow to get information internal to the compiler. The syntax of traits is similar to `pragma` syntax available in other languages such as C and C++, as well as D. Thus, new traits can be added and implemented easily in future compiler versions if needed. The traits are effectively a set of entry-points into the D compiler internals.

The following code demonstrates how string mixins, compile-time function execution, and compile-time reflection allow to generate code, choosing a requested method of a struct at compile-time.

```
1  import std.stdio, std.traits;
2  struct Pattern {
3    const string method;
4    string generate() {
5      foreach (m; __traits(allMembers, Pattern))
6        if (method == m &&
7            isCallable!(typeof(__traits(getMember, this, m))))
8          return `writeln("applied `~mixin ("this."~m)~`.");`;
9      return `writeln("`~method~` pattern is not available.");`;
10   }
11   auto diagonal() { return "diagonal pattern"; }
12   auto horizontal() { return "horizontal pattern"; }
13 }
14 void main() {
15   mixin (Pattern("vertical").generate());
16   mixin (Pattern("horizontal").generate());
17   mixin (Pattern("method").generate());
18 }
```

When the above program has been compiled, its `main` function is equivalent to the following code

```
void main() {
  writeln("vertical pattern is not available.");
  writeln("applied horizontal pattern.");
  writeln("method pattern is not available.");
}
```

In the `main` function each `mixin` statement forces the D compiler to create an object of the struct `Pattern` at *compile-time* (lines 15–17). Each `Pattern` object is initialized with a string that is assigned to the only data member `method` of the struct (line 3). Immediately after constructing each `Pattern` object, the compiler executes on it the member function `generate`. The function finds a `Pattern` member with the same name as the string stored in the `method` field (line 6). If the found member is a function (line 7), the D compiler executes the member function after mixing in a method invocation expression (line 8). Of the three objects constructed in lines 15–17, the compile-time function execution on line 8 happens only for the object `Pattern("horizontal")`. The return value of `generate` is a string that represents a valid D statement generated at compile-time. The CLOP compiler uses the same mechanism to apply code rewriting patterns that we describe in Section 3.1. Also, it uses reflection to query the D compiler for information about the elements of the host application, e.g. the type of the elements of an array given the symbolic name bound to that array.

OpenCL. A typical OpenCL application consists of the *host* code, that usually runs on the CPU, and the compute *kernels*. The host code of an application invokes the OpenCL API functions to query the available devices, to create an OpenCL run-time *context* and one or more *command queues* in that context. The host uses a command queue to submit asynchronously commands to the device associated with the queue. The commands are either data transfers between the host and the device, or kernel invocation requests. An OpenCL API implementation comprises a vendor provided library and C header files that declare the API functions and data structures. The D package registry includes a package of bindings to the OpenCL API. With these bindings, heterogeneous programming with OpenCL in D is the same as in C or C++.

In OpenCL a *kernel* is a function written in the *OpenCL C* programming language. The host calls OpenCL API functions at run time to translate the kernel code to the machine code of a specific device. The kernel can call other non-kernel functions, but not the functions of the host application. In the OpenCL execution model a kernel is executed over a domain of points in an indexed N-dimensional space, called *ND-range*. There is one instance of the kernel running for every point of the domain. Such an instance is called a *work-item*. The work-items of a single kernel invocation can be thought of as running simultaneously. The entire domain of work-items is subdivided into *work-groups*. A work-group represents a subset of work-items that can share local memory and can synchronize on the barrier constructs provided by the OpenCL programming language.

3. CLOP Language and Compiler

3.1 Syntax and Semantics

CLOP follows the OpenCL programming model specifying an N-dimensional index space and the computations performed for each point in the space. CLOP augments the OpenCL language with additional constructs to specify the global synchronization patterns. Kernel code is designated by the `NDRange` construct, that defines the index space on which the kernel will be executed, and a compound statement that represents the computations to be performed on the index space. A CLOP fragment can contain any number of internal function declarations, but currently only one `NDRange` construct.

Listing 1 shows an implementation of the Needleman-Wunsch algorithm [8] embedded in the body of the D method `clop_nw`. Lines 11–19 are the CLOP code wrapped in a compile-time function call `compile` that returns a string of valid D source code. The string is mixed in place of the `mixin` statement at lines 10–20. The mixed-in code includes a constant string representing the generated OpenCL program and the D statements that invoke the OpenCL API to run the kernel on a computing device.

```
1  class Needleman_Wunsch {
2    NDArray!int F, S; // matrices of computed cost and scores
3    int rows, cols, penalty;
4    this(string[] args) {
5      ... // code to initialize rows, cols and penalty
6      S = new NDArray!int(rows, cols);
7      F = new NDArray!int(rows, cols);
8    }
9    void clop_nw() {
10     mixin (compile(q{
11       int max3(int a, int b, int c) {
12         int k = a > b ? a : b;
13         return k > c ? k : c;
14       }
15       Antidiagonal NDRange(r : 1 .. rows, c : 1 .. cols) {
16         F[r, c] = max3(F[r - 1, c - 1] + S[r, c],
17                        F[r, c - 1] - penalty,
18                        F[r - 1, c] - penalty);
19       }
20     }));
21   }
22 }
```

Listing 1: A fragment of a D program with embedded CLOP code. The function `clop_nw` is a complete implementation of the Needleman-Wunsch algorithm, that computes the cost matrix `F` of all possible alignments of two sequences. The matrix `S` contains the alignment scores for every pair of symbols. The variables `F`, `S`, `penalty`, `rows`, and `cols` are data members of the class `Needleman_Wunsch` and used transparently inside the CLOP code.

Line 15 specifies that the computations are to be performed over a 2-dimensional box $[1, rows) \times [1, cols)$, with variables `r` and `c` used to index the corresponding dimensions. The ND-range of CLOP is not always the same as the ND-range used to invoke the generated OpenCL kernel. CLOP provides global synchronization patterns, which define how the kernel will be invoked on the ND-range space. On line 15 we request the compiler to generate the `Antidiagonal` global synchronization pattern. The kernel will be invoked on each anti-diagonal of the box $[1, rows) \times [1, cols)$ with synchronization after each invocation. The first invocation will be for indices $\{1, 1\}$, the second—for indices $\{2, 1\}$ and $\{1, 2\}$, the third—for indices $\{3, 1\}$, $\{2, 2\}$, $\{1, 3\}$, and so on until the last invocation will be for indices $\{rows - 1, cols - 1\}$. The global synchronization pattern specifier is optional. If it is omitted, the CLOP compiler will generate the host code that invokes the kernel on the entire ND-range index space at once. Figure 1 shows the syntax of CLOP's `NDRange` block.

In addition to global synchronization patterns CLOP can generate code for frequently recurring patterns that require synchronization between threads executing the same kernel. For example, the statement `s = reduce!"a + b"(0, t);` adds the elements in the array `t` and stores the result in the variable `s`. The CLOP compiler generates OpenCL code in which threads cooperatively perform the reduction applying a function of two arguments (`a + b`) to the elements of the array `t`. Such operations require in-kernel synchronization. In OpenCL this is limited to threads belonging to the same *work group*. However, the operation is so common, that CLOP automates the generation of code for it.

3.2 Run-time Support

The CLOP library provides the `runtime` object that initializes the OpenCL context and command queue to interact with a specific device. The object maintains a list of the OpenCL resources for allocation and release. In general, within the scope of the CLOP compiler it is impossible to know when to release a device buffer. If we release it at the end of every CLOP instance, it is extremely inefficient when the kernel is invoked multiple times, because the

$KernelBlock$	$\leftarrow$	$(SynchronizationPattern)?$
		$RangeDecl\ CompoundStatement$
$RangeDecl$	$\leftarrow$	"`NDRange`" "`(`" $RangeList$ "`)`"
$RangeList$	$\leftarrow$	$RangeSpec$ ("`,`" $RangeSpec$)$*$
$RangeSpec$	$\leftarrow$	$Identifier$ "`:`"
		$Expression$ "`..`" $Expression$
		("`$`" $Expression$)?

Figure 1. An excerpt from the CLOP grammar. The kernel block specifies an optional synchronization pattern, an N-dimensional index space, and the compound statement of computations to be done on the index space. In addition to the kernel block a CLOP fragment can include any number of declarations of entities to be used in the kernel block. The range specification can include an optional part delimited by '`$`', which allows to specify the desired OpenCL work-group size in each dimension of the index space.

buffer must be created anew every time the kernel is invoked, and data must be transferred to the device unnecessarily. The `runtime` object keeps a list of allocated resources, and releases them if the runtime is either initialized for new device or destroyed. The CLOP compiler generates the code that interacts with the `runtime` object.

The CLOP compiler recognizes D arrays and generates the code that creates device buffers and transfers data to and from the device when 1-dimensional arrays are used in CLOP blocks. In order to use multidimensional arrays in CLOP we implemented the class `NDArray` that can be used to create arrays of up to 3 dimensions, but internally stores data in one continuous chunk appropriate for transferring the data to and from an OpenCL device. The CLOP compiler recognizes variables of `NDArray` type and rewrites the multidimensional indices for these variables to properly access the data in continuous memory pointed to by such variables. Along with the `runtime` object `NDArray` objects help to manage the OpenCL resources.

3.3 Compiler Implementation

The CLOP compiler generates code in stages. The staging is done via compile-time function execution. Each stage generates code for the next stage, until the last stage generates the OpenCL code, comprising the kernel and any supplemental declarations, and D code that manages the run-time resources related to the host-device interaction.

The first thing that the compiler creates is the code that invokes the CLOP parser. To implement the CLOP parser we use a parser expressing grammar (PEG) generator. PEGs [3] allow to describe the lexical and hierarchical syntax of a language in one concise grammar which is parseable in linear time.

The result of parsing, an abstract syntax tree (AST), is saved as an `enum` value, which is used by the subsequent stages. The next stage takes as parameters the AST for this CLOP program and the program's location in the source file. The purpose of this stage is to analyze the AST and extract the information about the variables in the enclosing lexical scope which are external to the CLOP program. The objects referred to by these variables will be passed as parameters to the generated OpenCL kernel. For extracting this information the CLOP compiler relies on the compile-time reflection facilities of the D programming language.

The information about the external variables is packaged in two lists. One list contains the types of the variables, the other contains their symbolic names. The list of types is contained in a special template `TypeTuple` available in the D standard library. `TypeTuple` holds arbitrary types in a single container, e.g. `TypeTuple!(typeof(F),typeof(S),typeof(penalty))` is a list of types for variables `F`, `S`, and `penalty` from the Listing 1. This container is passed to the final CLOP compiler stage along with the list of strings that represent names of parameters for the OpenCL kernel that the compiler will generate.

The final stage of the compiler takes the original AST, lowers it by simplifying the expressions, and applies the required synchronization pattern. While lowering, it replaces high-level constructs such as `reduce` by the normal OpenCL code that implements these constructs. The compiler implementation has a collection of code snippets and methods that know how to expand these snippets. When the compiler walks the AST and encounters a construct that needs to be replaced, it invokes the corresponding method that performs the appropriate snippet expansion so that the construct is replaced by semantically equivalent OpenCL code.

Diagnostics. The CLOP compiler generates error messages as any compiler is expected to do when errors are found in the input program. The CLOP compiler collects the errors that it has found in the input program and generates the code that contains the `pragma(msg, errors_string);` construct. When this code is mixed in the application code, after the CLOP compiler returns, the D compiler outputs the error messages, if any.

Debugging the compiler. Debugging a compiler (CLOP) that is executed only in another compiler (D) can be a daunting task. The CLOP compiler runs only when the D compiler compiles an application that embeds CLOP code. If the CLOP compiler generates incorrect D code, the D compiler will abort compilation with a cryptic error message. Fortunately, the compile-time function execution rules require that the semantics of a function are the same whether it is executed at compile-time or at run-time. This rule combined with the support for unit tests in D allows us to debug the CLOP compiler efficiently by writing unit tests for the CLOP compiler internal functions and running the tests as executable programs with full access to the debugging facilities one is used to.

3.4 Results

We implemented several applications to compare heterogeneous programming in CLOP versus programming directly in OpenCL. Since the CLOP compiler works at compile-time only there is no observable difference between the run-time performance of applications implemented in CLOP and in *bare* OpenCL. The following table compares the applications in the number of lines of code.

Application	CLOP	OpenCL
Needleman-Wunsch	11	169
Back propagation	38	142
Linear algebra	12	134
Stencils	23	226

4. Related Work

Many domain specific languages were created to abstract the gory details of heterogeneous programming and to generate optimized code for GPUs and other accelerators, e.g. ViennaCL [11], a DSL for linear algebra operations, or SafeGPU [7], a contract-based library for GPGPU. Often such DSLs are implemented as C++ template libraries. They hide the low-level details of OpenCL behind the data types and operations provided by the DSL, but limit the algorithms that a programmer can express in them.

Steuwer et al. [12] echo our goals, but they raise the level of abstraction for heterogeneous programming using a library of skeletons which are similar to CLOP synchronization patterns. Their DSL, SkelCL, provides a collection of algorithmic skeletons, parameterized by the user's functions. These functions are passed as character string parameters to the skeleton objects. This is inconvenient since there's no analysis of the user's functions until the code is compiled by the OpenCL compiler at run time. In addition, the programmers are limited by the existing skeletons.

Accelerate [2] was designed to allow Haskell programs to perform operations with multi-dimensional arrays on a GPU. Similarly to Accelerate, CLOP uses quasiquotations to generate code for higher-level constructs common in heterogeneous programming. In the realm of JVM languages Delite [1] is a framework for DSLs embedded in Scala. Delite uses multi-pass staging to generate optimized device-specific code at run-time, whereas CLOP is based on strictly compile-time metaprogramming. Firepile [10] is an example of a library-based approach to GPU programming. It translates Scala methods to OpenCL kernels relying on functions being first-class objects. At run-time the Firepile translator locates the bytecode of a function passed to it as a value. It uses the bytecode to construct an abstract syntax tree for the function and then translate it to OpenCL kernel code.

Yin-Yang [4] is a framework for DSL embedding. It allows programmers to express their algorithms in a directly embedded DSL, and then translate the code into the corresponding deeply embedded DSL. This simplifies the development and debugging of the code written in the DSL, since the development is not hindered by a level of intermediate code produced when deeply embedded DSL code is staged. At run-time the generated deeply embedded DSL code is further staged and optimized to machine code targeting either the host CPU or any of the available devices. Unlike Firepile and Yin-Yang, the CLOP compiler is executed by the host language compiler (the D compiler in our case) and in principle does not cause any run-time overhead.

Recently Khronos Group released the SYCL [6] specification that defines an abstraction layer to allow single-source style heterogeneous programming with OpenCL. CLOP is our effort in the same direction taking advantage of compile time code rewriting.

5. Conclusion

Today, all but the simplest programs involve a complex interplay of various technologies and programming languages, thus making the programs themselves heterogeneous. One of the goals of programming tools is to smooth out the edges between a program's heterogeneous parts and make it appear as a single source program. To that end we develop CLOP, which makes heterogeneous programming more efficient. CLOP is open-source and available for download at `http://dmakarov.github.io/clop/`.

References

[1] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *PACT*, 2011.

[2] R. Clifton-Everest, T. McDonell, M. Chakravarty, and G. Keller. Embedding Foreign Code. In *PADL*. 2014.

[3] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *POPL*, 2004.

[4] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-yang: Concealing the Deep Embedding of DSLs. In *GPCE*, 2014.

[5] Khronos Group. The OpenCL Specification, Version: 1.2, 2012. URL `https://www.khronos.org/registry/cl/`.

[6] Khronos Group. C++ Single-source Heterogeneous Programming for OpenCL, 2015. URL `https://www.khronos.org/sycl`.

[7] A. Kolesnichenko, C. M. Poskitt, S. Nanz, and B. Meyer. Contract-Based General-Purpose GPU Programming. In *GPCE*, 2015.

[8] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48, 1970.

[9] NVIDIA. CUDA Parallel Computing Platform, 2015. URL `http://www.nvidia.com/object/cuda_home_new.html`.

[10] N. Nystrom, D. White, and K. Das. Firepile: Run-time Compilation for GPUs in Scala. In *GPCE*, 2011.

[11] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *GPUScA*, 2010.

[12] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IPDPSW*, 2011.