The ARES High-level Intermediate Representation

Nick Moss
Los Alamos National Laboratory
Email: nickm@lanl.gov

Kei Davis
Los Alamos National Laboratory
Email: kei@lanl.gov

Patrick McCormick
Los Alamos National Laboratory
Email: pat@lanl.gov

***Abstract*—The LLVM intermediate representation (IR) lacks semantic constructs for denoting common high-performance operations such as parallel and concurrent execution, communication, and synchronization. Currently, representing such semantics in LLVM requires either extending the intermediate form (a significant undertaking) or the use of ad hoc indirect means such as encoding them as intrinsics and/or the use of metadata constructs. In this paper we discuss a work in progress to explore the design and implementation of a new compilation stage and associated high-level intermediate form that is situated between the abstract syntax tree and LLVM's IR. This high-level representation is a superset of LLVM IR and supports the direct representation of these common parallel computing constructs, together with the infrastructure for supporting analysis and transformation passes on this representation.**

I. Introduction

The LLVM intermediate representation (LLVM IR) is a low-level, *assembly-like* language, static single assignment form that encodes purely sequential semantics [11]. As is the general case with intermediate forms, it is intended to be a universal and architecture-independent target. With the end of Moore's Law and the growing importance of parallel programming to achieve high performance, the purely sequential nature of the IR places limits on the ability to reason about parallel regions of code in both analysis and optimization passes. The most common alternative that is used today is to perform the high-level analysis and/or code transformations within the front-end (as is the case with Clang's support for OpenMP). In our view, this largely defeats the design goals of universalness and independence—it would be preferable to write a suite of analysis and optimization passes once rather than multiple times in various front-end implementations.

While it seems plausible to modify LLVM IR and then adapt the overall infrastructure to incorporate these parallel semantics, they encompass higher-level details that would add complexity and nuances into the core of the LLVM infrastructure. From one perspective, LLVM IR is too low-level of a representation for these goals. Additionally, this approach has the disadvantage of disrupting the infrastructure's wide adoption and thus many of the leverage points across any number of different marketplaces.

Ideally, we'd like to enable the ability to support universal, front-end-independent analysis and optimization of parallel operations without adversely affecting the core features and capabilities of LLVM. However, we also want to achieve this in such a way that we can leverage and benefit from LLVM's broad capabilities and infrastructure. The ARES project (Abstract Representations for the Extreme-Scale Stack) aims to create a set of inter-operable tools and approaches for high performance computing to enable the exploration and incremental movement towards more effective methods for programming next-generation architectures. This paper presents our initial efforts to explore these goals with the design and implementation of the ARES high-level intermediate representation (HLIR). The HLIR is extensible and is a superset of, and lowers to, LLVM IR for subsequent processing by standardized LLVM passes.

In the remainder of this paper we discuss the design and implementation of the ARES HLIR, the supporting runtime components, an example of its usage, and finally related and future work.

II. Design and Implementation

The HLIR was designed to capture both conventional sequential semantics (like LLVM IR) as well as the higher-level parallel semantics mentioned previously. This was a conscious design decision that allowed us to make the high-level IR a superset of LLVM's constructs. This also allowed us to directly leverage the underlying LLVM implementation as the building blocks for the HLIR.

More precisely, like LLVM IR, the HLIR is hierarchical with LLVM IR appearing only at the lowest level, i.e., in leaf nodes in the ARES data structures. The high-level representation is implemented as a set of C++ classes designed to provide in-memory manipulation.[1] The HLIR contains recursive or nested HLIR-specific constructs with LLVM IR at key leaf nodes such as a task body. The HLIR defines a number of node types: leaf nodes include symbols, strings, numeric types, an LLVM IR function (which can represent an arbitrary section of code but is packaged into a proper function, encapsulating its dependencies). Recursive nodes allow multiple nesting within the representation, for example, mapping a symbol to another node (a symbol key/value pair), or a sequence of nodes, referenced by position instead of a named symbol. Both maps and sequences allow heterogeneous node value types. HLIR modules are designed with the capability to be easily merged. For example, one HLIR module containing definitions for a parallel HLIR construct can be merged into the scope of another HLIR module.

Our current implementation supports tasks, parallel for/reduce, communication, and synchronization. We have chosen

[1]Like LLVM IR, the HLIR also has a convenient textual representation but we have not yet defined and implemented a bitcode representation.

978-1-5090-3878-7/16 $31.00 © 2016 IEEE
DOI 10.1109/LLVM-HPC.2016.9

this initial set of functionality because it closely addresses our application requirements, but we plan to expand beyond this in the future and have designed the HLIR to be highly extensible. Remaining language-independent, multiple compiler front-ends could target HLIR to encode additional concepts such as memory locality and placement, data layout, data parallelism, and more. The HLIR adds a flexible and expressive hierarchical high-level representation to LLVM IR that is capable of capturing recursive definitions and attributes that cannot be readily represented in a traditional IR. A major difficulty we encountered in prior work with Clang is that its ASTs are not designed to be modified or successively transformed like LLVM IR. The HLIR is intended to bridge the gap between such high-level but rigid ASTs and sequential assembly-level code, to provide both low-level sequential instruction semantics as well as the ability to readily represent and modify nested AST-like structures such as loops and other high-level representations needed to express parallel constructs, and like LLVM IR, to be mutable.

One of the key advantages of our system is the ease with which parallel operations can be created and how the system inter-operates with LLVM. For example, HLIR will set up an LLVM IR function for the body of a parallel for or reduce and provides entry points for the compiler to specify the IR for this body and will automatically take care of the details of capturing any data dependencies. In this way, the HLIR system is highly customizable while abstracting many of the necessary details needed to finalize and lower parallel for/reduce, task, or communication directives including parallel launching and synchronization.

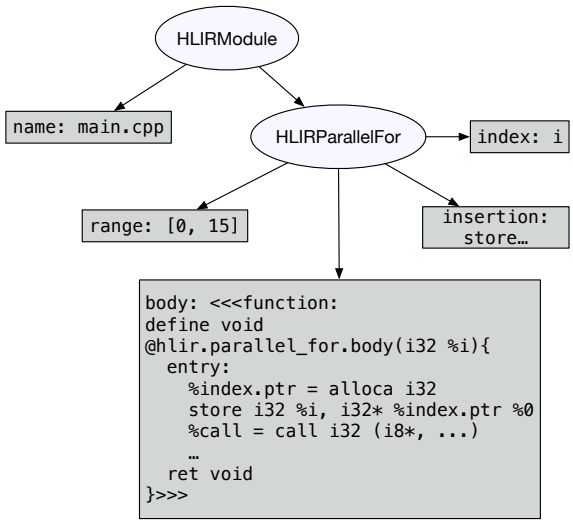

Fig. 1: A portion of an HLIR module containing a simple parallel for.

A. HLIR Structure

In this section we give a qualitative description of the major structural features of the HLIR.

a) Modules: As in LLVM IR, the top-level HLIR structure is called a module. Module attributes capture top-level information describing the module as a whole, for example, original source language, HLIR version number, etc.

b) LLVM IR sections: HLIR allows for the representation of arbitrary sequences of LLVM IR, but we require that they be properly wrapped in an LLVM IR function which notionally receives its dependencies as function parameters, i.e., does not reference any global variables or functions. It should be noted that this is merely a convenient way to package LLVM IR sequences. An LLVM value or type in isolation can be captured directly as a property of an HLIR construct, for example, an induction variable, or the reduction variable in a parallel reduce.

Next we describe how specific HLIR constructs are realized in our system.

c) Thread Pooling and Data Capturing: Outlining is a technique for isolating code and its data dependencies into a function so that it can be run in parallel in conjunction with a thread pool, or other runtime mechanisms. Our tasking and parallel for/reduce mechanism relies on outlining to bundle up the body of such constructs into an IR function which is then code generated. At runtime a pointer to this function is queued on a shared thread pool together with any external data that the body uses, and also with priority and synchronization objects. From the front-end's perspective this capturing of external data happens automatically—the actual transformation is deferred to the HLIR lowering pass. This means that when performing the code generation for a parallel construct the front-end can conveniently neglect that such values were introduced in a separate scope.

d) Tasks: A *task* is much like an ordinary function in LLVM IR, but is restricted in how it uses its inputs/outputs and global state because tasks run asynchronously and in parallel. HLIR currently supports read-only inputs and return value outputs. Implicitly, a task has a future associated with its return value. Within the HLIR transformation pass, when a value associated with a task future is subsequently used in IR, code is generated to block until that future has been released. Similarly, what appears in LLVM IR as ordinary calls to task-marked functions are transformed to proper parallel task launches by the same HLIR pass.

e) Parallel For: The body of a parallel for is lifted into an LLVM function and an HLIR value attribute can be retrieved from the HLIR representation giving the insertion point where a front-end can begin code generating this body. Further outlining is performed on this function by the HLIR pass to transform it so that it can be executed by our runtime. Here the HLIR performs dependency and variable usage analysis similar to what is done for a task.

f) Parallel Reduce: From the perspective of code generation and in terms of its HLIR representation, parallel reduce resembles parallel for. However, a parallel reduce includes a single associative reduction operation that acts on the left-hand-side of a variable we have explicitly specified as an HLIR attribute—the reduction result value. We generate an LLVM

IR function corresponding to the body of the parallel reduce and perform capturing of any external data dependencies. We have implemented a divide-and-conquer reduction algorithm, which is code-generated in IR for best performance and gets enqueued on M threads in our thread pool and each of these calls into the body function and applies the associative operator to a slice of the index space of size n. After computing the partial results, the reduction algorithm computes the final result in $O(\lg n)$ steps.

g) Communication: HLIR includes *channels* as an abstract means of communication and we have implemented socket-based channels and FIFO-based channels for testing purposes. We implemented a message buffering and message handling system that can be used in combination with channels to perform point-to-point communication and barrier synchronization. Currently, we have done more work on the runtime implementation of communication constructs than on their HLIR representations.

B. Runtime

For prototyping purposes we use a very simple runtime library to support parallel execution on a conventional CPU. The runtime is a C-based ABI but is implemented in C++ using concurrency features in the C++14 STL, Pthreads, and the Argobots light-weight user-level thread library [16]. The runtime consists of memory allocators, a thread pool, and synchronization classes using semaphores implemented in terms of condition variables. Internally, LLVM IR functions are created such that they are called with a single pointer to code generated structs that bundle up runtime arguments and application-level data so they can be queued to and executed by the thread pool.

Multiple nestings of parallel for/reduce and recursive task launches posed a challenge initially for our execution system because subtasks or nested parallel for loops have an associated future or synchronization object that must be waited on by their parent, thus occupying a thread. We solved this problem with Argobots which allowed us to code generate each task or nested parallel for/reduce such that it performs a non-blocking call to check the futures of its subtasks where threads can yield in the event that a future is not yet ready.

Multiple nesting of parallel for/reduce add additional complexity to our code generation scheme in the HLIR pass, a few details of which are worth noting. A nested parallel for/reduce contains multiple levels of data dependencies which must captured and passed in at the top-level and propagated as they are modified and passed to sub-launches. HLIR assumes responsibility for this to make it significantly less difficult for a front-end to specify the body of a parallel for/reduce. To handle nested values, from the HLIR client's perspectives, values appearing at different levels are the same, but the HLIR pass needs to take care of properly queueing and restoring them at each level.

III. Usage

In the preceding sections we described the overall design of HLIR. In this section, we briefly cover various HLIR implementation details and how HLIR is used by a front-end to target parallel constructs.

A. Usage Overview

HLIR is implemented using C++ 14, taking advantage of modern C++ features and is designed to be used by compilers using LLVM for code generation. After linking with the HLIR library and including the appropriate header, a front-end can then create an HLIR module, and call a number of methods on it to create HLIR parallel constructs, for instance, `createParallelFor()`, `createTask()`, etc.

B. HLIR nodes

HLIR's representation is provided by a hierarchy of *nodes* which are briefly described here. One should consult `HLIR.h` for a complete reference.

- `HLIRNode`—the base class of all other HLIR nodes. An HLIR node may be a child of at most one other node. A node is either a leaf or recursive. Recursive nodes can hold a heterogeneous collection of children nodes.
- `HLIRScalar`—the base class for simple scalar nodes such as `HLIRString`, `HLIRInteger`, etc. These are simple wrappers for scalars such as string, integer, floating point values, etc.
- `HLIRSymbol`—a lexical symbol, internally stored as a string.
- `HLIRVector`—a vector of heterogeneous nodes; used for storing positional information.
- `HLIRMap`—a recursive key/value map where keys are symbols and values may be heterogeneous.
- `HLIRFunction`—an LLVM IR `Function` and HLIR-specific convenience methods.
- `HLIRValue`—an LLVM IR `Value` and HLIR-specific convenience methods.
- `HLIRInstruction`—an LLVM IR `Instruction` and HLIR-specific convenience methods.
- `HLIRModule`—corresponds to an LLVM IR module and holds HLIR parallel constructs and convenience methods and HLIR meta-data describing the module as a whole.
- `HLIRTask`—a task tied to an ordinary LLVM IR function as its body.
- `HLIRParallelFor`—a parallel for loop referencing an ordinary LLVM IR function as its body, instruction insertion point for producing code for this body, and parallel iteration variable and ranges.
- `HLIRParallelReduce`—a parallel reduce loop similar to `HLIRParallelFor` but also holds the reduce result variable and reduction operator type.

C. HLIR Lowering Process

The HLIR module pass is responsible for transforming an HLIR module and associated LLVM IR instrumented with HLIR specific metadata and intrinsics into ordinary LLVM IR with calls to our runtime. The HLIR module pass executes first, before other LLVM passes, and HLIR specific code can then benefit from existing LLVM transformations—although

we haven't yet introduced any targeted optimizations that are intended to take advantage of specific LLVM passes. We plan to investigate if existing LLVM analyses could be used in this process or future extensions of HLIR.

The lowering process is nearly automatic, from the front-end's perspective. A front-end targeting HLIR need not be aware of the internal details, only a familiarity with the HLIR interface we provide, as summarized in the preceding section. For instance, creating a front-end that targets tasks is as simple as specifying that a certain LLVM IR function is a task by creating an `HLIRTask` on the `HLIRModule`, then the HLIR pass handles the details of transforming calls to that function into task launches and awaiting futures when that future's value is required in LLVM IR.

D. Front-end

We have implemented a Clang-based C++ front-end with HLIR-specific extensions that currently supports parallel for, parallel reduce, and tasks. The following code sections briefly demonstrate various usages of these constructs.

1) Tasks: The `task` keyword is placed at the beginning of a function declaration to designate it as a task. This example uses tasks to compute the n-the Fibonacci number in parallel.

```
task int fib(int i){
  if(i <= 1){
    return i;
  }

  return fib(i - 1) + fib(i - 2);
}
```

Note that a task is then invoked as if it were a normal function.

2) Parallel For: A simple example of parallel for.

```
float A[SIZE];

for(auto i : Forall(0, SIZE)){
  A[i] = i;
}
```

3) Parallel Reduce: A simple example of parallel reduce.

```
float sum = 0.0;

for(auto i : ReduceAll(0, SIZE, sum)){
  sum += 1.0;
}
```

E. Comparing HLIR to OpenMP

Appendix A contains a listing showing the generated IR outlining for OpenMP compared to HLIR for the simple parallel for example listed previously. As shown, outlined code in HLIR is considerably more succinct and makes fewer runtime calls. However, runtime performance is also determined by how this code is queued and executed by the runtime. In our system, we queue one function for each indexed item executed whereas it may be preferable in the future to aggregate multiple executions per queued function.

IV. Potential Future Work

We have demonstrated the feasibility of our approach in our backend and front-end prototype, including support for tasks, parallel for/reduce, and communication. There is high potential for future development in multiple areas. We will continue formalizing the semantics of HLIR and extending our coverage of HLIR to include support for additional parallel constructs including integrating the work we have done with communication in the runtime with tasks to provide distributed functionality. We will investigate the possibility of read/write dependency analysis in HLIR to determine read/write attributes of task parameters, perhaps as gathered by an HLIR-specific analysis pass that would run during our lowering process. Such infrastructure would allow us to create a dependency graph of the data and read/write usage of tasks in order to coordinate the asynchronous launches across multiple tasks that operate on shared data. In addition, this type of analysis could aid in multiple stages of our code generation process.

There remains a number of potential optimizations that we are considering for our code generation and runtime. One possibility, related to tasking and parallel for/reduce, as it applies to nesting, is to give higher priority to execution units, i.e., tasks of greater depth to speed up the execution of yielded execution units. Another broader area of potential effort is investigating the possibility of encoding OpenMP semantics in our system and making HLIR targetable through an OpenMP interface.

V. Related Work

The idea of using multiple levels of IR between source code and machine code is not new. Indeed, if we regard the AST and assembly language as IRs then the practice is ubiquitous. Here we consider extant examples of various levels of IR between the AST and assembly language, and also approaches that circumvent this approach.

1) Open64: Perhaps the best known example is the Open64 compiler's WHIRL IR. Despite now being effectively discontinued, Open64 has at least until recently used as both a research platform in compiler and computer-architecture research [3], and as the basis of proprietary compilers [1], [15], [18], and so remains relevant today. WHIRL encompasses five distinct IRs, ranging from the Very High WHIRL that is very much like an AST; through target-independent High WHIRL and largely target-independent Mid WHIRL; to Low and Very Low WHIRL that roughly correspond to conventional target-dependent assembly language and machine code, respectively.

2) Diderot: Another example is the Diderot language design and implementation [4]. Diderot is a high-level DSL for parallel methods of biomedical image analysis and visualization. It achieves portability via back-ends targeting C with gcc vector extensions, parallel C code, and OpenCL. It is an example of a system using multiple levels of IR to span the semantic range between AST and output code, and that performs program analysis and optimization that can only be performed because of the constraints imposed by the

domain specificity of the language, i.e., in terms of higher-level semantics than the target.

3) Non-imperative paradigms: The use of multiple IRs spans programming paradigms. In the case of Prolog, a so-called logic language, this was used to good effect to ease the task of global program compilation via incremental compilation [9]. For functional languages it has been used since the very first such compiler [2] to the most modern—for GHC these are *Core*, *STG*, and *C--*, the latter from which LLVM is generated [17], [19]—arguably because the semantic gap between such languages and machine code is much greater than for conventional imperative languages.

4) GCC Gimple: The Gnu Compiler Collection (GCC) primarily uses a single IR, *gimple*, as a universal target for multiple language front-ends [5]. While a single representation, in GCC there are differing contraints on what are legal forms between various different passes of the compiler. These constraints are not well documented and therefore may be thought of as defining implicit sub-languages. One could argue that a cleaner design would take the approach of LLVM, where passes can generally be performed in arbitrary order, or use distinct IRs that by design enforce the implicit constraints.

5) OpenCL SPIR: The Khronos Group™ has published a sequence of specifications for SPIR™, their Standard Portable Intermediate Representation. SPIR evolved from a specification of translation of OpenCL to LLVM [8], to the current SPIR-V, a stand-alone binary intermediate language for graphical shaders and compute kernels [7]. It is intended to be an extensible, universal target for multiple front-ends. A notable design goal is to allow (some) optimizations to be performed offline, i.e., at compile time, in terms of the semantics represented by SPIR.

6) Scout: In a prior project Scout, we extended Clang and C++ to add first-class extensions for computational meshes, parallel operations on meshes, and in situ visualization and plotting. We modified Clang's front-end including extending the lexer to add new keywords, added new AST types to represent meshes and parallel for, extended the parser and semantic analyzer to support our new statements, declarations, and expressions, and implemented their associated code generation to IR and interface to our runtime. We created an extension of DWARF to recognize Scout constructs and modified Clang and LLDB to support debugging of Scout statements and expressions [6], [12]. Scout, like the current OpenMP functionality in Clang, differs from the previous approaches in that there is no new intermediate representation: the functionality is wired into the Clang front-end and generates LLVM IR and runtime library calls directly.

7) Kokkos Clang: In a recent project (*Kokkos Clang*) we developed a specialized compiler by extending Clang in order to perform targeted code generation for Kokkos-specific constructs. This compiler generates highly optimized code for GPU/Nvidia targets and also provides semantic (domain) awareness throughout the compilation toolchain of these constructs such as *parallel for* and *parallel reduce*. In addition to runtime performance improvements, we achieved significant reductions in compile times and executable binary sizes by taking a more direct code generation path and by bypassing C++ template expansions which Kokkos relies on heavily.

8) OpenARC: A parallel effort in the ARES project is the further development of the Open Accelerator Research Compiler (OpenARC) [10]. Here parallelism in C source programs is encoded with OpenACC directives [13], and the IR takes the form of Java classes that provide an AST-like view of the source code [14]. Their goal is to target a diversity of heterogeneous architectures including GPUs and FPGAs.

VI. Conclusion

We described how LLVM's lack of direct support for parallel constructs and the need for a representation that is capable of capturing recursive and AST-like semantics has led us to create the HLIR. We described the design, implementation, and usage of HLIR. So far, we have implemented tasks, parallel for, parallel reduce, and communication/synchronization. We demonstrated the practicality of HLIR by implementing an experimental front-end where we added first-class support for these constructs to C++ in approximately a hundred lines of code each.

The current implementation of ARES is available as open-source under a BSD-style license and can be found at: https://github.com/losalamos/ares.

VII. Acknowledgements

The majority of the work presented in this paper was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the United States Department of Energy, under the guidance of Dr. Sonia Sachs. Additional support was provided by the Department of Energy's National Security Administration, Advanced Simulation and Computing Program. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S. Department of Energy under contract DE-AC52-06NA25396. The authors would like to thank the reviewers for their constructive feedback and suggestions.

References

[1] AMD. x86 Open64 Compiler Suite. http://developer.amd.com/tools-and-sdks/cpu-development/x86-open64-compiler-suite/, 5 2007.

[2] L. Augustsson. A compiler for lazy ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 218–227, New York, NY, USA, 1984. ACM.

[3] B. Chapman, D. Eachempati, and O. Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *International Journal of Parallel Programming*, 41(6):825–854, Dec. 2013.

[4] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM.

[5] GIMPLE - GNU Compiler Collection (GCC) Internals. http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html, Aug. 2013.

[6] J. A. Jablin, P. McCormick, and M. Herlihy. Scout: High-performance heterogeneous computing made simple. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 0:2093–2096, 2012.

[7] Khronos Group. SPIR-V Specification Provisional, Version 1.1, Revision 3, August 11, 2016. https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.pdf.

[8] Khronos Group. The SPIR Specification, Standard Portable Intermediate Representation, Version 2.0 – Provisional, Revision Date: 2014-06-05. https://www.khronos.org/registry/spir/specs/spir_spec-2.0.pdf.

[9] A. Krall and T. Berger. Incremental global compilation of prolog with the vienna abstract machine. In *In International Conference on Logic Programming*, pages 333–347. MIT Press, 1995.

[10] S. Lee and J. S. Vetter. Openarc: Extensible openacc compiler framework for directive-based accelerator programming study. In *Proceedings of the 2014 First Workshop on Accelerator Programming using Directives*, 2014.

[11] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, Aug 2016.

[12] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 1–10, Piscataway, NJ, USA, 2014. IEEE Press.

[13] OpenACC: Directives for Accelerators. http://www.openacc.org/.

[14] OpenARC: Open Accelerator Research Compiler. https://ft.ornl.gov/research/openarc.

[15] PathScale. EKOPath. http://www.pathscale.com/ekopath.html, 8 2013.

[16] S. Seo et al. Argobots: A Lightweight Low-level Threading/Tasking Framework. https://collab.cels.anl.gov/display/ARGOBOTS/.

[17] T. T. GHC (STG, Cmm, asm) illustrated. https://github.com/takenobu-hs/haskell-ghc-illustrated, 2016.

[18] Tensilica. Extensa C/C++ Compiler. http://www.tensilica.com/uploads/pdf/XCC~Compiler~Overview.pdf, 5 2007.

[19] D. Terei. A Haskell Compiler. http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html, 2016.

APPENDIX

A. OpenMP Outlining

```
; Function Attrs: nounwind uwtable
define internal void @.omp_outlined.(i32*
    noalias %.global_tid., i32* noalias
   %.bound_tid., [1000000 x float]*
   dereferenceable(4000000) %A) #1 {
  %1 = alloca i32*, align 8
  %2 = alloca i32*, align 8
  %3 = alloca [1000000 x float]*, align 8
  %.omp.iv = alloca i64, align 8
  %.omp.lb = alloca i64, align 8
  %.omp.ub = alloca i64, align 8
  %.omp.stride = alloca i64, align 8
  %.omp.is_last = alloca i32, align 4
  %i = alloca i64, align 8
  store i32* %.global_tid., i32** %1,
     align 8
  store i32* %.bound_tid., i32** %2,
     align 8
  store [1000000 x float]* %A, [1000000 x
     float]** %3, align 8
  %4 = load [1000000 x float]*, [1000000
     x float]** %3, align 8
  store i64 0, i64* %.omp.lb, align 8
  store i64 999999, i64* %.omp.ub, align
     8
  store i64 1, i64* %.omp.stride, align 8
  store i32 0, i32* %.omp.is_last, align
     4
  %5 = load i32*, i32** %1, align 8
  %6 = load i32, i32* %5, align 4
  call void @__kmpc_for_static_init_8u(%
     ident_t* @0, i32 %6, i32 34, i32* %.
     omp.is_last, i64* %.omp.lb, i64* %.
     omp.ub, i64* %.omp.stride, i64 1,
     i64 1)
  %7 = load i64, i64* %.omp.ub, align 8
  %8 = icmp ugt i64 %7, 999999
  br i1 %8, label %9, label %10

; <label>:9
  br label %12

; <label>:10
  %11 = load i64, i64* %.omp.ub, align 8
  br label %12

; <label>:12
  %13 = phi i64 [ 999999, %9 ], [ %11,
     %10 ]
  store i64 %13, i64* %.omp.ub, align 8
  %14 = load i64, i64* %.omp.lb, align 8
  store i64 %14, i64* %.omp.iv, align 8
  br label %15

; <label>:15
  %16 = load i64, i64* %.omp.iv, align 8
  %17 = load i64, i64* %.omp.ub, align 8
  %18 = icmp ule i64 %16, %17
  br i1 %18, label %19, label %31

; <label>:19
  %20 = load i64, i64* %.omp.iv, align 8
  %21 = mul i64 %20, 1
  %22 = add i64 0, %21
  store i64 %22, i64* %i, align 8
  %23 = load i64, i64* %i, align 8
  %24 = uitofp i64 %23 to float
  %25 = load i64, i64* %i, align 8
  %26 = getelementptr inbounds [1000000 x
      float], [1000000 x float]* %4, i64
     0, i64 %25
  store volatile float %24, float* %26,
     align 4
  br label %27

; <label>:27
  br label %28

; <label>:28
  %29 = load i64, i64* %.omp.iv, align 8
  %30 = add i64 %29, 1
  store i64 %30, i64* %.omp.iv, align 8
  br label %15

; <label>:31
  br label %32

; <label>:32
  call void @__kmpc_for_static_fini(%
     ident_t* @0, i32 %6)
  ret void
}
```

B. ARES Outlining

```
define void @hlir.parallel_for.body(i8* %
   args.ptr) {
entry:
  %args.ptr1 = bitcast i8* %args.ptr to %
     struct.func_args*
  %0 = getelementptr inbounds %struct.
     func_args, %struct.func_args* %args.
     ptr1, i32 0, i32 0
  %synch.ptr = load i8*, i8** %0
  %index.ptr = getelementptr inbounds %
     struct.func_args, %struct.func_args*
      %args.ptr1, i32 0, i32 1
  %funcArgs.ptr = getelementptr inbounds
     %struct.func_args, %struct.func_args
     * %args.ptr1, i32 0, i32 2
```

```
  %1 = load i8*, i8** %funcArgs.ptr
  %2 = bitcast i8* %1 to %struct.
     func_args.0*
  %3 = getelementptr inbounds %struct.
     func_args.0, %struct.func_args.0*
     %2, i32 0, i32 0
  %A = load [1000000 x float]*, [1000000
     x float]** %3
  %4 = alloca i1
  %5 = load i32, i32* %index.ptr, align 8
  %conv = uitofp i32 %5 to float
  %6 = load i32, i32* %index.ptr, align 8
  %idxprom = zext i32 %6 to i64
  %arrayidx = getelementptr inbounds
     [1000000 x float], [1000000 x float
     ]* %A, i64 0, i64 %idxprom
  store float %conv, float* %arrayidx,
     align 4
  br label %exit.block

exit.block:

   ; preds = %entry
  call void @__ares_finish_func(i8* %args
     .ptr)
  ret void
}
```