OpenMP Support for the Quack Programming Language

Jacob Lambert

***Abstract*—OpenMP is a high-level, directive-based standard designed to allow programmers to implement parallel programs without relying on low-level parallel programming APIs or threading libraries. Many compilers for popular languages such as C, C++, and FORTRAN support OpenMP directives.**

We explore the potential for implementing OpenMP support in our Quack compiler, which currently takes a Quack program as input and generates LLVM IR. We discuss our previous efforts in this area, and evaluate why these efforts were unsuccessful. We then survey two papers detailing the implementation of two research-oriented OpenMP compilers. We conclude by presenting new strategies for implementing OpenMP support in our current Quack compiler.

Keywords*—*Quack, OpenMP, OpenUH, OpenARC, OpenACC

I. Introduction

A. Quack

Quack [1] is a simple object-oriented, type-inferenced programming language, loosely based on Java. Quack was designed with a limited set of features for the University of Oregon CIS 461/561 course. Although Quack is considered a simplistic language, it supports integer and string literals, control flow, classes with single inheritance, class method definitions and calls, and dynamic dispatching.

As part of the course, we design and implement a Quack compiler that performs lexing, parsing, type-checking and code generation. Our implementation takes Quack code as input and produces LLVM code as an alternative to x86 or other assembly languages. Our compiler utilizes the C-based flex [2] and bison [3] for lexing and parsing, respectively. We implement creation of the Abstract Syntax Tree (AST) and type-checking in pure C++. Finally we implement code generation that produces LLVM [4] code using C++ and the LLVM C++ API [5]. We compile the C++ code using the Clang [6] compiler, and we compile the generated LLVM code with LLVM's *lli* compiler.

B. LLVM

Low-Level Virtual Machine (LLVM) [4] is a programming language designed to serve as an intermediate representation (IR) between high-level programming languages like C, C++, and FOTRAN, and low level assembly codes like x86.

Targeting LLVM in code generation has several advantages over targeting more traditional assembly languages. Because LLVM is not architecture specific, the representation is cross-platform, allowing programs and optimizations written in LLVM to be executed across different systems. Additionally because LLVM abstracts many of the more tedious programming steps required in other assembly programming, LLVM code is more human readable and human writable.

Even though LLVM offers several abstractions over assembly, because of its location in the compilation stack and representation, many of the optimizations normally performed at the assembly stage can still be performed at the LLVM stage, making LLVM ideal for implementing optimization passes.

C. LLVM C++ API

The LLVM C++ API is an extensive code base created to simplify the process of LLVM code generation for C++ compilers. Due to the size of the API, its complexity, the level of abstraction it provides, our unfamiliarity with code generation and its associated vocabulary, and the API's shortage of documentation, we found the LLVM API difficult to use as beginners. However, as we became more familiar with the interface, we understood how it could greatly simplify code generation for experienced users.

The API has four driving concepts or abstractions: The Context, The Module, Basic Blocks, and a Builder.

The Context is used to store meta-information concerning current code generation, and is a function parameter for many of the API calls. The Module is the top-level construct for keeping track of generated code. Basic Blocks are analogous to basic blocks in LLVM and other assembly languages. Functions, methods, and statement blocks are all represented by Basic Blocks. The Module consists of a sequential set of Basic Blocks. Finally, the Builder is used to append code statements to basic blocks, such as arithmetic, function calls, or return statements. Basic Blocks consists of lists of objects created by the Builder class.

D. OpenMP

OpenMP [7] is an open-source, cross-platform standard used for writing high-level, directive-based parallel programs. Several commercial C, C++, and FORTRAN compilers currently support OpenMP, along with a few research-based compilers.

OpenMP offers a high-level abstraction over lower-level parallel programming interfaces like the pthreads library and MPI. New users can create parallel programs using simple pragmas, while advanced OpenMP users can fine-tune parallelization by using pragmas with additional parameters.

Because of its numerous advantages, OpenMP has been one of the main tools used in parallel programming since its inception. For this reason, we felt that supporting OpenMP would be an interesting and feasible project extension for our Quack compiler.

E. Paper Outline

In section II we discuss our previous work and our attempts at implementing OpenMP support in our Quack compiler. In section III, we look at three open-source research-based compilers that support OpenMP in an attempt to understand different approaches to OpenMP support. In section IV we conclude by proposing possible solutions for implementing OpenMP in our Quack compiler, using knowledge gained by previous attempts and the paper survey.

II. Previous Work

We originally chose to implement OpenMP support as our compiler extension because both compiler writers have previous experience in parallel programming, and supporting simple OpenMP pragmas seemed interesting yet feasible. Because the Quack language does not have for loops, the typical target for OpenMP, we chose to implement simple OpenMP parallel sections. For example,

```
#pragma omp parallel
{
  f.foo();
}
```

Once annotated with the OpenMP pragma, the above code would be executed N times across all processing units, where N is the number of OpenMP threads specified by an environment variable.

We first thoroughly search through the LLVM C++ API for any calls specifically designed for creating parallel regions that we could use to handle the OpenMP sections. However, the API lacked any support for creating parallel regions.

To better understand how other compilers handle OpenMP constructs, we developed some simple C++ code with OpenMP pragmas similar to the functionality we wished to replicate in Quack. We then compiled this code using Clang++, with the option to emit LLVM code enabled. This gave us insight into how Clang++ generates LLVM code from C++ with OpenMP. We researched the function calls made in the generated LLVM code, which led us to Clang's internal API for generating parallel-enabled LLVM.

However, even after considerable searching, we could not find a way to interface Clang's internal API with the LLVM C++ API, which now forces us to consider other options, detailed in section IV.

III. Survey of OpenMP Compilers

A. OpenUH

OpenUH [8] is an open-sourced C, C++, and FORTRAN compiler with OpenMP support. OpenUH was developed by researchers at the University of Houston. OpenUH is built as an extension to the Open64 compiler. Open64 began as a C, C++, and FORTRAN compiler for Intel Itanium processors. OpenUH extends Open64 by adding OpenMP support and compiling for not only Itanium processors but any device with C or FORTRAN back-end compilers as well. Although there have been many advances in OpenUH since its inception, here we discuss the original compiler.

Most research compilers act as a front-end source-to-source compiler. That is, they translate input code into output code that can then be compiled to machine code by a different back-end compiler. This allows the compiler to be widely used across different platforms. However, there is normally no way to coordinate the front-end and back-end compilers, which can negatively affect potential for optimization.

Most commercial and industry compilers take an integrated approach, where the input program is compiled directly to device-specific machine code. In contrast to the front-end and back-end approach, with the integrated approach the different phases of the compiler can work tightly together when optimizing, which leads to better overall performance.

OpenUH adopts a hybrid approach. When targeting specific architectures, OpenUH acts as an integrated compiler, and compiles the input program directly to machine codes. The specific architecture mentioned in the paper is Intel Itanium processors.

When targeting other architectures, OpenUH acts as a source-to-source compiler. For C and C++ input programs, the output language is C. For FORTRAN programs, the output language is FORTRAN.

The OpenUH works in several stages.

- Similar to the Quack compiler, OpenUH first parses the input program and creates an intermediate representation (IR), WHIRL OpenMP. In the initial transformation, the OpenMP directives are left unchanged, only pushed down into the IR.
- Next, using the WHIRL IR, OpenUH performs the inter-procedural analysis (IPA) and loop-nest optimizer (LNO). IPA carries out optimizations such as dead function and variable elimination and inlining, while LNO performs optimizations like loop fission and fusion.
- After IPA and LNO, OpenUH preprocesses the OpenMP directives, and then performs the *lower_mp* phase. This phase transforms the WHIRL with OpenMP directives into WHIRL with mutli-threaded code with OpenMP runtime calls. The OpenMP is a portable library designed by the paper authors.
- Next, OpenUH transforms WHIRL into an static single assignment (SSA) form, performs additional optimizations, and then transforms the SSA back into WHIRL IR.
- After all optimizations have been performed, OpenUH performs one of the following transformations.
    - If targeting Itanium architectures, OpenUH generates the appropriate device code.
    - If targeting other devices, OpenUH generates C or FORTRAN code, depending on the input language, using *whirl2c* or *whirl2f* respectively.

B. *OpenARC*

OpenARC [9] is a C compiler that supports OpenACC, a directive-based parallel programming language similar to OpenMP. OpenARC is developed by ORNL, and is based on the Cetus C compiler.

Like OpenUH, OpenARC parses the input into a formal intermediate representation High-Level Intermediate Representation (HIR), the OpenARC analog of WHIRL.

HIR has some nice features to simplify optimizations. Every class in the IR extends a traversable class, which provides the functionality to easily traverse IR objects using built-in iterators. Also, the IR can easily be extended to include new types of nodes, which makes OpenARC ideal for prototyping new language extensions and libraries.

Although OpenARC doesn't provide an integrated compiler to target a specific device like OpenUH with Itanuium, OpenARC does support several different outputs for source-to-source compilation. While OpenARC only accepts C with OpenACC annotations as input, the compiler can produce several different target languages, including CUDA, OpenCL, OpenCL specific to Xeon-Phi architectures, and OpenCL specific to Altera OpenCL architectures. This allows OpenARC to be widely used across several different types of systems and devices.

The remainder of the paper discusses various types of optimizations and compiler passes performed by OpenARC, and compares the performance of OpenACC programs compiled with OpenARC against OpenACC programs compiled by industrial compilers, and against CUDA programs compiled with Nvidia's compiler.

Unfortunately they don't detail exactly how the OpenACC directive calls are handled, although we can assume that, like OpenUH, OpenACC runtime library calls are created in the intermediate representation.

IV. Conclusions

After researching how other compilers handle OpenMP directives, we realize there are several options for supporting OpenMP for Quack.

Our first option would be to change our intermediate representation. It would be possible to replace LLVM with either WHIRL or HIR, which both already support OpenMP and have an associated OpenMP runtime.

This would require replacing the scanning and parsing stages of OpenARC or OpenUH with our quack scanner and parser. Because OpenARC input is strictly C while OpenUH can support C++, WHIRL would be a better choice of IR, as it already contains infrastructure needed to handle Quack classes.

Alternatively, instead of adapting the IR's used in the other compilers, we could modify our IR to mimic the behavior of WHIRL and HIR. From our experiments with the Clang compiler, we know LLVM supports an OpenMP runtime, detailed in [10]. We can push the OpenMP directives into our IR, which is currently a tree-based structure written in C++. We can then write an additional pass or traversal that replaces the OpenMP constructs with multi-threaded code that calls the runtime library, similar to the *lower_mp* routine of WHIRL. This would require a significant amount of work, as translating some OpenMP constructs require restructuring the AST. With these first two options, we would be handling the OpenMP constructs inside the AST.

As a third option, we could extend the LLVM C++ API to support OpenMP constructs. This would be the most difficult option, but also possibly the most useful. For example, our extensions would include calls such as:

```
  Builder.CreateOpenMPConstruct()
```

These extensions would insert the appropriate OpenMP Runtime Library calls into the API's Module class, and restructure the code as necessary. With this option, we would be handling the OpenMP constructs in the code generation phase.

In conclusion, we motivated support for OpenMP in the Quack programming language. We explained initial attempts at supporting OpenMP, and surveyed the steps other research compilers have taken to support OpenMP. Finally, using the knowledge gained from research surveys and our initial attempts, we have proposed several possible avenues for supporting OpenMP in the Quack Programming Language.

References

[1] M. Young, "Quack: A language for beginning compiler writers reference grammar," 2017.

[2] V. Paxson, W. Estes, and J. Millaway, "Flex: the fast lexical analyzer," *U RL http://flex. sourceforge. net*, 2012.

[3] C. Donnelly and R. Stallman, "Bison. the yacc-compatible parser generator," 2004.

[4] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization.* IEEE Computer Society, 2004, p. 75.

[5] "Llvm c++ api," http://llvm.org/docs/doxygen/html/index.html, accessed: 2017-03-25.

[6] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[7] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[8] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: An optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[9] S. Lee and J. S. Vetter, "Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing.* ACM, 2014, pp. 115–120.

[10] "Llvm openmp runtime library interface," http://openmp.llvm.org/Reference.pdf, accessed: 2017-03-25.