AutoOMP: Automatic OpenMP parallelization of code using LLVM

McCall Saltzman
School of Science
Rensselaer Polytechnic Institute
110 8th St, Troy NY 12180

Adam Freeman
School of Science
Rensselaer Polytechnic Institute
110 8th St, Troy NY 12180

***Abstract*—AutoOMP attempts to leverage the flexibility and power of the LLVM compiler infrastructure to accelerate cases of serial code using OpenMP. First, LLVM is used to lower code to an intermediate language. Then, using the LLVM code analysis framework, AutoOMP detects which loops can be safely parallelized and which can not. AutoOMP breaks out the parallelizable loops into their own functions, creating OpenMP loops out of them.**

I. Introduction

Writing parallel code has always been a more complicated task than writing serial code. With parallel code, the programmer needs to consider the impact of thread dependencies, shared memory and variables, and many other factors which can cause deadlock or incorrect code. However with modern parallel architectures it becomes increasingly important that programs and software be able to take advantage of the speedup from the modern parallel architectures.

There have been many attempts to ease the burden of parallelization on the programmer. New languages have been developed with parallelization in mind. Frameworks have made it easier than ever to integrate parallel code into a serial project or construct an entire code base out of parallel code. However none of these attempts have taken over the programming world, due to a lack of programmer familiarity with the languages and frameworks. Clearly, there is a need for a parallelization mechanism which does not place any burden on the programmer at all, allowing for seamless and entirely hassle free parallelization. AutoOMP attempts to fill this niche by adding parallelization at compile time using LLVM, and integrating code with the OpenMP parallelization framework.

A. The LLVM Compiler Infrastructure

The LLVM compiler infrastructure is a "collection of modular and reusable compiler and toolchain technologies." [1] It has gained immense popularity in recent years due to its wide range of supported languages and architectures, and quick additions of new language standards such as c++17 in the C++ frontend to LLVM, clang. While the clang frontend for C and C++ is one of the most popular languages used with LLVM, the compiler supports many other languages through the use of different front ends. Other popular languages include Common Lisp, Fortran, Haskell, Java (bytecode), Python, and Swift. Any of the languages that LLVM supports can be compiled to any of the architectures that LLVM supports, which include every major architecture and some unconventional ones. For example, a program written in C++ can be compiled to x86 machine code. Haskell can be compiled to Javascript using LLVM.

The LLVM compiler infrastructure achieves this flexibility through the use of an intermediate language. A program compiled using LLVM is first read by one of the many front ends to LLVM. The job of an LLVM front end is to parse and convert human readable code into the LLVM Intermediate Language, henceforth simply called IR. While the initial source code varies widely between language and paradigm, the IR produced from the front end will always have the same structure and can be operated on independently of the source language.

Due to the uniformity of the structure, LLVM runs several optimization and analysis passes once the code has been converted to IR, again regardless of the source language. Some of these passes are purely for analysis, to collect information to later be used in optimization passes. These analysis passes can collect such information as the Dominator front for a function, or collecting information about the loops in the program. These analysis passes then provide information about the code to optimization passes, which can more easily make educated decisions about how to optimize the code and how to ensure the safety and correctness of the code. The optimization passes can state to the pass manager which other passes they depend on so that the required information is present when that pass is run. This also defines the order in which passes are run. This can also control the order of optimization passes, so that a pass that must be run last or before other passes can specify when it will run.

Once all the optimization passes run, LLVM uses a backend to lower the IR to machine code. Many backends exist for LLVM covering all major architectures and many esoteric ones as well. This is the stage at which architecture specific optimizations take place.

B. OpenMP

OpenMP, short for Open Multi-Processing, is an API for locally running many threads to achieve parallelization. OpenMP uses the fork-join model, meaning that there is a master thread that is spawning slave threads to aid in the computation. OpenMP aims to make the job of parallelization extremely easy for the programmer, and to do so uses preprocessor macros, such as `#pragma omp parallel` to mark certain sections of code as parallelizable. The compiler will then see the sections of code marked as such, and transparently break them out into a separate function, replacing the original section of code with a function call to openMP libraries. The called function will then fork numerous times depending on the environment and call the broken out section of code.

II. AutoOMP Architecture

AutoOMP is implemented as an LLVM optimization pass. As every part of LLVM is, AutoOMP is written using C++. More specifically, AutoOMP is implemented as a Module pass, which means that the pass gets run once on each module. This differs from, say, a function pass which gets run once and separately on each function. AutoOMP must be run as a module pass due to the forced process of breaking out loops into separate functions. A function pass does not have the ability to touch functions outside its own scope, so a higher scope is needed in order to create new functions. Our module pass specifies that it must be run after two other passes - the `LoopInfoWrapperPass` and the `DominatorTreeWrapperPass`. These passes are wrappers for analysis passes that gather information on all the loops in the module, and information on the Dominator tree for every function. One notes that the dominator tree pass is a function pass, so access to the analysis from this pass must also specify a specific function to retrieve analysis from.

When our AutoOMP pass is run, each function in the module is looped though. We then retrieve the list of loops from the previously run `LoopInfoWrapperPass`, and filter it based on which functions are in the loop. This filtering and separate retrieval of the loops is necessary because of the next step in our loop selection process, which is to query the dominator tree for each loop. As the dominator tree pass is a function pass and not a module pass, a query to this pass requires that you specify a specific function to obtain information about. Since we are filtering the set of all loops in the module by one particular function, we can use this function to query the previously run dominator tree analysis pass.

We then iterate over every loop to test if it can be separated out and parallelized. This is basically a further filter to eliminate all loops that don't fulfill a certain set of properties that we define. For example, the loop must be in simplified form, a form specified by the llvm IR as having a loop variable counting up from zero. We also check by examining its starting and terminating instructions, to verify that they are well formed and branch to the loop header and exit. We also rule out any loops which contain a function call, as this can easily lead to complicated dependencies and incorrectness in parallelization.

Once a loop is selected for parallelization, we first extract the loop into a separate function. After a few sanity checks like ensuring that the loop actually has instructions before and after its execution, we use a LLVM CodeExtractor to extract the entire code block containing of the loop into a separate function.

After a loop becomes its own function, we need to create the intermediate OpenMP function that will call our newly separated loop, as well as construct some scaffolding into the newly extracted function. This includes such things as adding parsing for arguments and establishing a new scope for this function. These changes are also necessary for being called by openMP, as openMP uses them to establish a thread local context. Once we are sure that the extracted function will have all the pieces that openMP may require, we create a function call to the openMP fork function with our recently extracted function as an argument. Also passed to this function are variables that are required by the scope of the extracted function but not necessarily changing during the lifetime of the loop. The dominator tree is used indirectly in this process. We then insert the newly created call to fork in openMP into the original code where the loop used to be, hopefully creating a seamless call to openMP and seamlessly adding parallelization.

III. Conclusion

Unfortunately, in the time available we were unable to fully implement runnable automatic open mp conversion. The documentation available for the OpenMP implementation is limited and as such we were unable to find the appropriate calling conventions for both `kpmc_fork_call`, and `kpmc_for_static_init`. However, the project was to some degree a success. The AutoOMP pass correctly identifies a subset of parallelizable loops, and can separate them from the main function into a callable region. Additionally, the loop variables, and memory accesses can be identified, analyzed for dependencies, and properly added to the header region as seen in OpenMP code compiled with Clang. This shows that if the calling conventions for the missing functions can be figured out, the pass will be able to produce runnable open mp code.

In summary, LLVM is a promising framework for adding parallelism to existing codebases. Once completely working, the AutoOMP pass will enable programmers to search for speedups in their code without any more overhead than adding a compiler option. While the approach taken by AutoOMP may not have been the most performant, enabling programmers to test speeding up their code may also allow them to see where they need to optimize for better performance. While LLVM currently does support OpenMP, the documentation in that area is distinctly lacking, leading to the issues AutoOMP is experiencing with malformed call instructions. If resolved, basic parallelism will be available to all programs that can target LLVM IR.

A. Future Work

Moving forward, the AutoOMP project will be continued. We aim to find the proper way to build these missing call instructions, and furthermore, to narrow down the range of loops we can parallelize. Additionally, we intend to use more of the code analysis tools available within LLVM to find special case loops and parallelize them differently. Once the code is runnable, we intend to compare performance and accuracy of several popular benchmark suites in a few different languages, both to verify that the loops we accelerate are safe to do so, and that the speedup gained from this is both useful and substantial.

REFERENCES

[1] LLVM Overview, The LLVM Compiler Infrastructure Project, [Online]. Available: https://llvm.org/. [Accessed: 03-May-2017].