Extending Polygeist to Generate OpenMP SIMD and GPU MLIR Code

Arun Thangamani, Vincent Loechner, Stéphane Genaud

Extending Polygeist to Generate OpenMP SIMD and GPU MLIR Code

Arun Thangamani*, Vincent Loechner(✉), and Stéphane Genaud

University of Strasbourg and Inria, France.
{thangamani,loecher,genaud}@unistra.fr

Abstract. The state-of-the-art source-to-source polyhedral schedulers annotate loops that can be vectorized with directives, which are merely recommendations to the compiler. However, standard compiler auto-vectorizers may fail to vectorize them because of the complexity of the loops structure or nested statements in the restructured code. The Polygeist compilation framework can generate polyhedral optimized (tiling and parallel loops) MLIR code, but it neither annotates the loops with vectorization directives nor auto-generates the vectorized code.
In this paper we describe a proposal to extend Polygeist to generate OpenMP SIMD MLIR code for vector loops. We also want to further extend the code generation process to support GPU MLIR code thereby targeting accelerated architectures.

Keywords: loop optimization · MLIR · compilers · polyhedral techniques · heterogeneous architectures

1 Introduction and Motivation

Loop optimizations are a main topic in compilers as the computation of many applications mostly revolves around complex or nested loop structures. Generating an efficient code for these loop structures results in a significant reduction in execution time by reaping the benefits of parallelism, data locality, and many others. The polyhedral optimization techniques are efficient in optimizing nested loop structures. They (i) extract the polyhedral structure, (ii) find an optimizing schedule for the polyhedral representation, and (iii) finally convert the optimized polyhedra into code. There are many open-source polyhedral schedulers available like Pluto [1], PoCC [8], Polly [2], and PPCG [12]. These schedulers do three loop transformations with the polyhedral information: (i) loop tiling, (ii) parallelization, and (iii) vectorization. Tiling is achieved by re-structuring the loop nests, parallel loops are expressed by inserting `parallel for` OpenMP directives, and vectorization opportunities are marked with vectorizaion directives by some compilers. This latter directive is merely a recommendation for the back-end compilers to vectorize the upcoming loop structure but we observed that in few cases standard compilers do not actually vectorize it. We carried

* Arun Thangamani is a 3^{rd} year PhD student at University of Strasbourg, France.

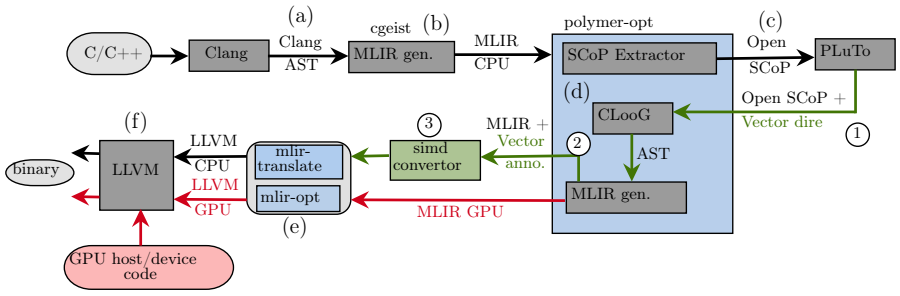

Fig. 1: Overview of Polygeist compilation-flow being modified by our proposed technique. Green arrows show our extended/modified version of CPU code generation. Red arrows show our newly proposed GPU code generation

out a detailed study on polyhedral schedulers [10] and found that the main reasons reported by compilers to not vectorize the marked code were, for example: (i) could not determine the number of loop iterations, (ii) multiple nested loops, (iii) complex loop statements (like function calls, irregular control flow, and pointer/array deference), (iv) control flow inside loops, and (v) inner-loop count not invariant. Further, in a few cases the vectorization is not applied because the vector cost-model of the back-end compiler estimates that it is not profitable. Thus, we are not able to fully reap the benefits of polyhedral techniques.

1.1 Polygeist

MLIR [4] is a part of the LLVM [3] framework and it offers a means to express code operations and types through an extensible set of common *intermediate representations* (IR), called dialects, each dedicated to a specific concern at different levels of abstractions. *Lowering* a dialect in MLIR refers to the process of transforming operations and constructs from a higher-level dialect into a lower-level representation that is closer to the target execution environment or hardware platform. Polygeist [5] is a compilation tool that lowers `C/C++` to Polyhedral MLIR code. Figure 1 shows the Polygeist compilation process (in gray, excluding the simd converter and GPU code generator). It takes an input source program, and (a) extracts the high level-information using `clang` and generates a Clang AST, (b) with a code generator named *cgeist*, Polygeist generates MLIR equivalent code from the Clang AST, (c) with a built-in tool named *polymer-opt*, the polyhedral representation is extracted (via *SCoP Extractor*), and using Pluto (or another external polyhedral scheduler) the extracted polyhedral representation is optimized for loop tiling and parallelization, (d) from the optimized `OpenSCoP` file, the CLooG code generator emits an AST which serves as an input for *cgeist* to generate MLIR code. *Cgeist* retains the information of the parallel loop identified by Pluto by marking it with the `scop.parallel` loop attribute, (e) an MLIR pass lowers the `scop.parallel` attribute loop to an OpenMP parallel loop, (f) finally the lowered MLIR code is translated into LLVM IR, that is passed to LLVM to generate an object file.

```
1 #pragma scop
2 for (i = 0; i < n; i++) {
3  for (j = 0; j <= i; j++)
4   C[i][j] *= beta;  // S0
5  for (k = 0; k < m; k++)
6   for (j = 0; j <= i; j++)
7    C[i][j] += A[j][k]*alpha*B[i][k] + B[j][k]*alpha*A[i][k]; // S1
8 }
9 #pragma endscop
```

Listing 1: Compute intensive loops of the syr2k kernel.

```
1  // ... code skipped for space
2  affine.for %arg7 = 0 to #map()[%0] {scop.parallel}
3   affine.for %arg8 = 0 to #map1(%arg7)
4    affine.for %arg9 = #map2(%arg7) to min #map3(%arg7)[%0]
5     affine.for %arg10 = #map2(%arg8) to min #map4(%arg9, %arg8)
6      S0
7
8  affine.for %arg7 = 0 to #map()[%0] {scop.parallel}
9   affine.for %arg8 = 0 to #map1(%arg7)
10   affine.for %arg9 = 0 to #map()[%1]
11    affine.for %arg10 = #map2(%arg7) to min #map3(%arg7)[%0]
12     affine.for %arg11 = #map2(%arg8) to min #map4(%arg10, %arg8)
13      affine.for %arg12 = #map2(%arg9) to min #map3(%arg9)[%1]
14       S1
```

Listing 2: Polyhedral optimized MLIR code generated by Polygeist for the loops shown in Listing 1.

Listing 2 shows the polyhedral MLIR optimized code generated by Polygeist for the loops of the syr2k kernel (from PolyBench/C [7]) shown in Listing 1. The `scop.parallel` annotated `affine.for` loop at line 2 and 8 will be converted to MLIR OpenMP parallel loop during lowering. Lines 3-5 and 9-13 are tiled loops executing statements S0 and S1, respectively. It is to be noticed that Pluto identifies the `affine.for` loop in line 5 of Listing 2 as vectorizable using vector directives, but Polygeist does not use this information as it can not lower the loop statements to an MLIR vector code.

Considering this context, we propose to extend the MLIR code generation process of Polygeist to convert the loops which are vector annotated by the polyhedral compiler to an MLIR OpenMP SIMD loop, thereby emitting SIMD instructions for loop statements with the help of OpenMP.

2 Related Work

Polly [2] is the first LLVM project that targets polyhedral optimization on the LLVM IR. However at the LLVM IR level, it is difficult to recover the structure of the code (affine loops, bounds, tests, array references, etc.), and it requires post-processing of the transformed code to integrate it back to the IR. The support of many vector options and GPU code generation in Polly was discontinued in 2023. Polygeist [5] is a step to ease this process and to work at the higher level of abstraction of MLIR.

In our previous work [9] [11], we introduced an optimized and heterogeneous code generation process for cardiac electrophysiology simulations with the help

```
1  affine.for %arg7 = 0 to #map()[%0]
2   affine.for %arg8 = 0 to #map1(%arg7)
3    affine.for %arg9 = #map2(%arg7) to min #map3(%arg7)[%0]
4     %c1 = arith.constant 1 : index
5     %2 = arith.muli %arg8, %c32 : index
6     %3 = arith.addi %arg9, %c1 : index
7     %4 = arith.muli %arg8, %c32 : index
8     %5 = arith.addi %4, %c32 : index
9     %6 = arith.cmpi slt, %3, %5 : index
10    %7 = arith.select %6, %3, %5 : index
11    omp.simdloop simdlen(4) for (%arg10):index = (%2) to (%7) step (%c1){
12     S0
13     omp.yield
14    }
15 // ... code of S1 skipped for space
```

Listing 3: OpenMP SIMD loop generation by our proposed technique for vector annotated loop(s) from Listing 2.

of MLIR. Our main focus was to optimize a compute intensive *parallel for* loop in the openCARP cardiac electrophysiology simulator [6]. The traditional compilers could not efficiently vectorize the loop as it includes complex statements (like function calls, array access, and pointer de-references), irregular control flow and others. We traverse the AST of the *parallel for* loops and with the help of MLIR Python bindings our code generator emits equivalent MLIR vector instructions using the `vector` types for CPU or MLIR code with the standard types (e.g., `f64`) for GPU. Then, the emitted MLIR is lowered with respect to the target architecture (x86, `CUDA` for Nvidia, or `ROCm` for AMD) and linked to the rest of the code with the help of the LLVM compiler infrastructure.

3 OpenMP SIMD Code Generation

This section presents three major changes in CPU code generation to be able to generate SIMD code, as follows:

① Identify which loop(s) can be vectorized from the Pluto output,
② Annotate those `affine.loop`'s with the attribute `scop.vector`,
③ A new optimization pass that converts those marked loops to `omp.simdloop`. The pass should compute the arguments of the simd constructs: (i) size of the vector (`simdlen`), (ii) the step value between the loop iterations, and (iii) the lower/upper bound loop variables.

The green compilation flow in Figure 1 shows the modified/extended approach of our MLIR CPU code generation technique. We rely on the `affine` and `omp` dialects for simd loop creation. Listing 3 line 11 shows the `affine.for` loop in line 5 of Listing 2 is converted to `omp.simdloop` by our modified approach. Firstly, Pluto scheduler identifies the loops that could be vectorized and informs Polygeist. We modified *cgeist* so that it annotates the vectorizable loop(s) with the `scop.vector` attribute. Our simd converter pass then converts those annotated loops to `omp.simd` loop. Using an environmental variable the pass allows the user to choose the `simdlen` (set to either 2, 4, 8, or 16 targeting `SSE`, `AVX2`,

or `AVX512` vector architectures). In line 11 of Listing 3, the simd length is set to 4 targeting `AVX2` architecture with using `double` types. The `step` value is set to the default value of one. Arguments `%2` and `%7` in line 11 are the lower and upper bound values of the simd loop, respectively. In Listing 2, the lower and upper bounds are expressed as affine maps (#map2 and #map4), such expression is not defined in the openMP dialect. Therefore, we have to substitute the map definition with arithmetic operations as shown in lines 4-10 of listing 3. Finally, the Polygeist existing compilation flow will lower the simd loop to LLVM IR.

4 Experimental Results

We implemented the proposed CPU compilation flow on top of the Polygeist source from the git repository. We used a 2x20-cores CascadeLake Intel Xeon Gold 5218R @2.1GHz CPU and `AVX2` vector architecture set for our evaluation. We choose six benchmarks (*2mm*, *syr2k*, *gramschmidt*, *correlation*, *nussinov*, and *heat-3D*), one from each category of PolyBench/C [7] with `EXTRALARGE` data-set for evaluating our implementation for CPU code generation. Unfortunately, the OpenMP SIMD optimized MLIR code took the same execution time as the code without our optimization and does not improve the use of the vector instructions. Indeed, the `omp.simd` loop was not vectorized. Since this functionality is newly added to MLIR, it has limited support. The MLIR framework is undergoing very active development phases to support various compiler optimizations. So, in a very near future MLIR could provide complete optimization support for OpenMP SIMD loops and we hope that our proposed technique improves vectorization.

Another work-around would be to emit the vector instructions directly with the help of MLIR's `vector` dialect instead of relying on SIMD loops, but this would duplicate the work that `omp.simd` should do.

5 Future Work

Using the multiple code generation capabilities of MLIR, we will further extend Polygeist (red compilation flow in Figure 1) to implement the polyhedral GPU MLIR code generation for nested loop structures. We propose to generate MLIR GPU code for the loops enclosed within `scop` and let the rest of the code be in `C/C++`. The generated MLIR GPU code will be enclosed within two nested `scf.for` loops which are later lowered into an outer and an inner loop iterating over the GPU `blocks` and `threads`, respectively. The MLIR GPU code is first lowered to its GPU specific low-level dialect. This can be for example either `nvvm` (the Nvidia CUDA IR) or `rocdl` (the AMD ROCm IR) with respect to the target GPU architecture. Next is the MLIR translation pass that converts the low level MLIR code (`llvm` dialect) into an LLVM IR representation. The last step is the linking phase, where the `C/C++` emitted LLVM IR host code with the help of `clang` and the MLIR translated optimized LLVM IR are linked together into an object file using the LLVM compiler framework.

One challenge we may face is the function or library calls inside the nested loops. An equivalent GPU code is required for those function calls to keep the computations within the GPU rather than making frequent and costly context switches between host and device. Another challenge is the memory management on GPU. We have to choose between the unified memory model or a manual allocation and data transfers between the host and device for efficient management.

6 Conclusion

We propose a compilation flow in which Polygeist is extended to generate OpenMP SIMD (implemented) and GPU (to implement) MLIR code for nested loop structures. Although, the results are not satisfying we hope that our optimizations are good target for loops that have SIMD nature but cannot be auto-vectorized.

References

1. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI '08. p. 101–113. https://doi.org/10.1145/1375581.1375595
2. Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. **22**(4) (2012), https://doi.org/10.1142/S0129626412500107
3. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: CGO. pp. 75–88. San Jose, CA, USA (Mar 2004)
4. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: CGO '21. pp. 2–14
5. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: Raising c to polyhedral mlir. In: PACT '21. https://doi.org/10.1109/PACT52795.2021.00011
6. Plank, G., Loewe, A., Neic, A., Augustin, C., Huang, Y.L., Gsell, M.A., Karabelas, E., Nothstein, M., Prassl, A.J., Sánchez, J., Seemann, G., Vigmond, E.J.: The openCARP simulation environment for cardiac electrophysiology. Computer Methods and Programs in Biomedicine **208**, 106223 (2021). https://doi.org/10.1016/j.cmpb.2021.106223
7. Pouchet, L.N., Yuki, T.: PolyBench/C version 4.2.1-beta (2022), http://polybench.sf.net
8. Pouchet, L.N.: PoCC - The Polyhedral Compiler Collection. http://web.cs.ucla.edu/~pouchet/software/pocc/ (2012)
9. Thangamani, A., Jost, T.T., Loechner, V., Genaud, S., Bramas, B.: Lifting code generation of cardiac physiology simulation to novel compiler technology. p. 68–80. CGO 2023. https://doi.org/10.1145/3579990.3580008
10. Thangamani, A., Loechner, V., Genaud, S.: A survey of general-purpose polyhedral compilers. ACM Transactions on Architecture and Code Optimization (2024)
11. Trevisan Jost, T., Thangamani, A., Colin, R., Loechner, V., Genaud, S., Bramas, B.: Gpu code generation of cardiac electrophysiology simulation with mlir. In: Euro-Par '23. pp. 549–563
12. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for cuda. In: ACM TACO. vol. 9 (jan 2013). https://doi.org/10.1145/2400682.2400713