Pattern matching for idiom search in LLVM IR

Rajnish Aggarwal, Peter Oostema
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

1 Abstract

Compiler analysis generally falls into one of two categories: it could be general analysis that applies to all code (constant propagation, common sub-expression elimination etc), or it could consist of a set of pattern matching rules that transform certain idioms into better implementations (peephole optimizations, strength reduction etc.). In this work, we identify algorithmic patterns given an intermediate representation of system level languages C/C++, Java in LLVM. Extract features to lift this representation and transform it into a domain specific language (DSL) which implements high performance kernels for efficient utilization of hardware resources. The ultimate goal of this pattern matching is to encapsulate as much of the program as possible in higher level descriptions to enable powerful optimizations [1].

2 Introduction

The task of pattern recognition for idiom search as well as general purpose loop optimizations has been well studied. In [7], Mendis et al. lift convolution kernels in the stripped x86 binaries of Adobe Photoshop. The analysis comprises of run-time profiling, including memory profiling and expression tree resolution to formulate the actual pattern in the code. Similarly, [4] Kamil et al. lifts stencil kernels from Fortran code by generating a high level summary of the code that performs this operation by a technique termed as *verified lifting*. They take as input a block of potentially optimized code and infer a summary expressed in a high-level predicate language that is provably equivalent to the semantics of the original program. After summarizing these implementations, they can be lifted to any high performance kernel or a standard DSL implementation.

A great amount of research has also been done in the domain of polyhedral loop optimizations. The LLVM Polly Compiler [2] implements these optimizations which: identify static control parts (SCoP) [5] that are single entry, single exit basic blocks. All of the loop bounds and conditionals of a SCoP are affine functions and affine inequalities, respectively of the surrounding loop iterators and global parameters. These global parameters are invariant during the execution of a SCoP, but their values

are unknown at compiler time. Listing (1) is an example of SCoP that contains the statement S. The polyhedral model is a mathematical framework for loop nest optimizations, which focuses on modeling and optimizing the memory access behavior of a program and abstracts from individual computational operations.

```
    for (i = 0; i < M; ++i)
        for (j = 0; j < N; ++j)
            for (p = 0; p < K; ++p)
S:              C[i][j] += A[i][p] * B[p][j]
```

Listing 1: Example of matrix-matrix multiplication

The high performance community also becomes an integral part of this development because after recognizing a pattern one needs to have the ability to lift it to a suitable implementation. Hence, Henry et al. [3] show how one can organize data for efficient SIMD computations. Low et al. [6] show that a mathematical model can be designed for stencil computations and is enough for high performance implementation of stencil kernels. Once formalized, the choice of the implementation that suits the needs of the computation becomes imperative as it has already been generalized.

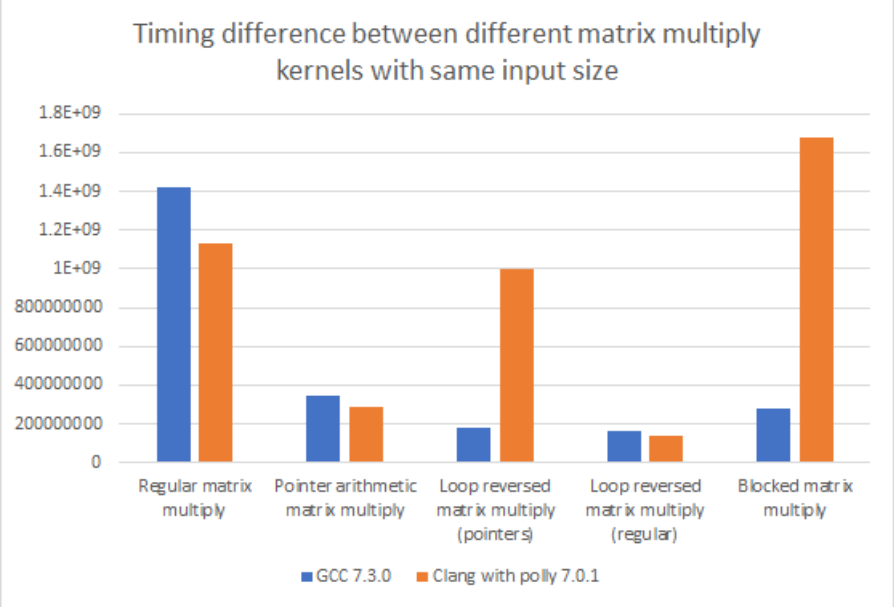

Figure 1: Performance difference in LLVM and GCC with the same matrix multiply kernel written in different ways (y-axis is in number of cycles taken for the kernel)

```
for (i = 0; i < M; ++i)
    for (p = 0; p < k; ++p)
        for (j = 0; j < N; ++j)
            C[i][j] += A[i][p] * B[p][j]
```

Listing 2: Example of matrix-matrix multiplication with loop reversal

```
for (i = 0; i < M; ++i)
    for (j = 0; j < N; ++j)
        for (p = 0; p < K; ++p)
            *(C + i * N + j) += *(A + i * N + p) *
                                *(B + p * K + j)
```

Listing 3: Example of matrix-matrix multiplication with pointer arithmetic

```
for (i = 0; i < M; ++i)
    for (p = 0; p < k; ++p)
        for (j = 0; j < N; ++j)
            *(C + i * N + j) += *(A + i * N + p) *
                                *(B + p * K + j)
```

Listing 4: Example of matrix-matrix multiplication with loop reversal

```
for (int i = 0; i < M; i += 6)
    for (int j = 0; j < N; j += 16)
        for (int p = 0; p < K; p += 32)
            for (int ii = i; ii < i + 6; ++ii)
                for (int jj = j; jj < j + 16; ++jj)
                    for (int pp = p; pp < p + 32; ++pp)
                        *(C + ii * N + jj) +=
                            *(A + ii * N + pp) *
                            *(B + pp * K + jj)
```

Listing 5: Example of blocked matrix multiply

2.1 The problem

The method of pattern recognition as described in [7] [4] requires run-time profiling information, linear equation solvers and users guided pragmas for detecting potential code blocks for optimizations. Moreover the techniques are designed to lift application specific kernels which cannot be generalized to different forms of numeric computations the code might have.

State of the art compilers like GCC and LLVM Polly implement all the methods of polyhedral loop optimizations and stencil detection as described above. However they face the halting problem [8] in general which inhibits many possible areas of program optimizations. Listing 1, 2, 3, 4 and 5 describe various implementations of matrix multiply. As we can see, there is a stark difference in performance although the operation being performed and the results are exactly the same (1).

Through our approach (Section 3) of recognizing patterns within the IR, we are able resolve all these implementations to a common template and hence can attain the same performance for all the listings which also surpasses the best performance of the modern compilers.

2.2 Our Approach

We present a generalized approach for pattern recognition from a given intermediate representation of LLVM IR. We can then lift this pattern into a high performance implementation by instrumenting a function call to a high performance kernel or an equivalent DSL representation. Through this approach we decouple the two problems of recognition and it's subsequent high performance implementation thus being able to achieve the same performance for all implementations of a kernel.

We rely on the user to provide the pattern of computation e.g.[+, *] that needs to be recognized, based on which we can detect the apt DSL/HPC function call and also filter the SCoP's of interest. Through the algorithm described in Section 3, we detect the overall tensor shape by working on the variables used to index into each of them.

3 Algorithm Details and Design

3.1 Detection

The first step in replacing a tensor operation idiom is to detect potential blocks that may contain them. First a single-entry single-exit block needs to be found, so that replacing the block will not change the control flow behavior of the program. Next since we are looking for tensor operations a store operation to an array in memory needs to be found, in other words a store to a point in memory with some variable offset. From the store instruction we search to find what is being stored, values from other arrays, and what operations are performed on these. If these steps match with the idiom being searched for we move on to verification.

3.2 Verification

Verification matches the detected code to the definition of the tensor operation being searched for. The first steps for verification are finding the indexing expression trees for the arrays and the ranges of these index variables. Currently our program is limited to arrays indexed with variables that are simply incremented to larger values and does not handle recursive structures. Because of this limitation we can flatten the expression tree to long string of sums of products. The products are index variables multiplied with a constant. Tensor operations are typically defined by what gets accumulated in a unique element of the resulting tensor. We find this by eliminating the index variables found in the resulting matrix and finding what in the tensor being operated on gets referenced for this single element. The ranges of the remaining index variables gives what elements of the input go into the output tensor. The ranges of the index variables on the output give the size of the tensor, which is necessary for creating a DSL expression for the idiom. The results found here are then checked against the definditon of the idiom being searched for.
This example of matrix multiplication (Listing 5) is blocked in every dimension, so each gets two variables iterating inside and outside blocks for each dimension. The indexing expressions are shown in Figure (2). After the common subexpression elimination matrices A, and B are left with their "k"

variables. If it it is a matrix multiplication the range of these will equal the size of their common dimension. If neither of these have been transposed the range of these will equal the multiplication factor on the other. Each element in a row of A is multiplied by an element in a column of B stored separated by the last element by the range of k. If one of the matrices is transposed the range needs to simply match between the two. Lastly the ranges of the eliminated variables need to match. The variables from A need to match one dimension of C, and B the other dimension.

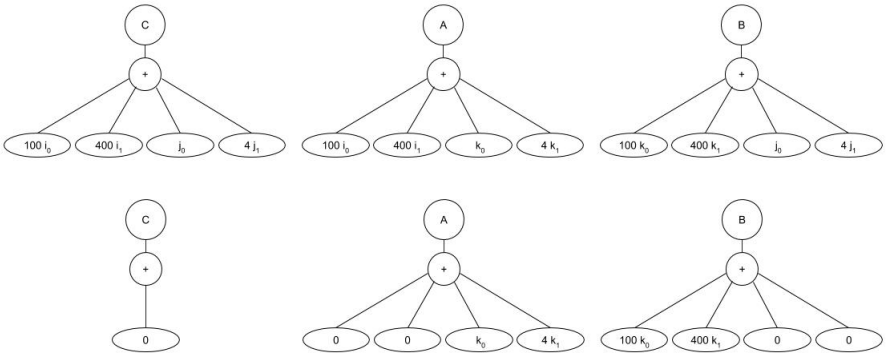

Figure 2: Index expressions for blocked C[i][j] = A[i][k] * B[k][j]. The top image gives the expression for all indexing, the bottom gives the indexes that store to a single element in C.

Another example tensor operation is a 2D convolution, such as a reduction. The indexing expressions are shown in Figure (3). For convolution the remaining index variables in the input tensor after common sub expression elimination will describe the filter. Their ranges will give the dimensions of the filter in both dimensions. The eliminated variables will have to match in the input and output tensors with similar ranges (plus or minus the filter size). The convolution will be lifted from finding the tensor and filter sizes from the ranges of the iterators.

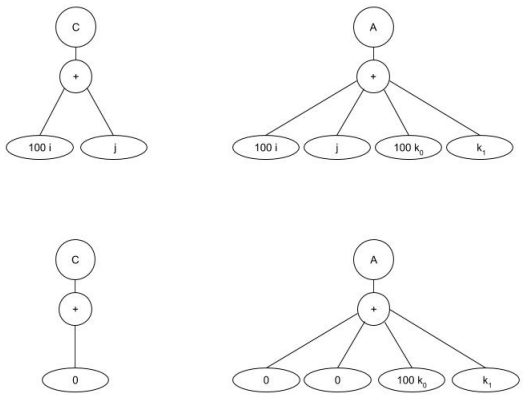

Figure 3: Index expressions 2D convolution C = conv(A). The top image gives the expression for all indexing, the bottom gives the indexes that store to a single element in C.

4 Experimental Setup

We demonstrate the pattern recognition of Listing 1 and 3 through our experiments. Following are the steps involved:

- Detect a store instruction of the type [+, *]. Verify that the LHS (Matrix C) is a matrix and similarly the RHS are either matrices or vectors
- Find all the variables used to index into each of these tensors and find their ranges (for the more general cases we need to resolve the expression tree). An assumption here is that only the sizes of the tensors are variables and the blocking parameters are statically defined
- All indexing done with pointer arithmetic as shown in Listing 3, 4 and 5 is of the type [<outermostIdx> * all inner sizes + ... + <innermostIdx> * 1], using this information we can figure out the order in which the variables index into the tensor
- Next, verify this template using the algorithm listed in Section 3. Also check that the set of loops corresponds to a SCoP i.e. the region of interest is a single entry single exit basic block
- Remove the set of loops which represent this SCoP and instrument the new function call

4.1 Assumptions and things not part of the code

In our implementation we assume that the high performance function is known, however when there exists a set of functions, we can validate the output for correctness and timing through each to figure out the correct implementation. This incurs a one time cost of trying out different kernels.

Also, for the implementation, we have just changed the pre-headers successor to the loop exit thereby ignoring the entire loop. Removing all the uses and taking care of the PHI nodes became painful at some point.

Since our high performance kernel uses Intel AVX intrinsincs, we have not instrumented the function call. We evaluate performance by running a stand alone kernel which takes in as inputs only those parameters that are evaluated by the pattern recognition algorithm.

5 Experimental Evaluation

The pattern detected for Listing 1 and 3 is shown in listing 6 and 7. Our LLVM pass can be invoked by **-inst-finder** flag.

```
================================================
Matrix C
================================================
Name:   %i26.0 = phi i32 [ 0, %for.end25 ], [ %inc52, %for.inc51 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 40000
```

```
Name:   %j30.0 = phi i32 [ 0, %for.body29 ], [ %inc49, %for.inc48 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 400
================================================
Matrix A
================================================
Name:   %i26.0 = phi i32 [ 0, %for.end25 ], [ %inc52, %for.inc51 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 4000
Name:   %k.0 = phi i32 [ 0, %for.body33 ], [ %inc46, %for.inc45 ]
Start: i32 0
End: i32 10
Stride: i32 1
Size: 40
================================================
Matrix B
================================================
Name:   %k.0 = phi i32 [ 0, %for.body33 ], [ %inc46, %for.inc45 ]
Start: i32 0
End: i32 10
Stride: i32 1
Size: 4000
Name:   %j30.0 = phi i32 [ 0, %for.body29 ], [ %inc49, %for.inc48 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 400
```

Listing 6: Pattern detected for Listing 1 (size of index is in bytes)

```
================================================
Matrix C
================================================
Name:   %i30.0 = phi i32 [ 0, %for.end29 ], [ %inc59, %for.inc58 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 400
Name:   %j34.0 = phi i32 [ 0, %for.body33 ], [ %inc56, %for.inc55 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 4
================================================
Matrix A
================================================
Name:   %i30.0 = phi i32 [ 0, %for.end29 ], [ %inc59, %for.inc58 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 40
Name:   %k.0 = phi i32 [ 0, %for.body37 ], [ %inc53, %for.inc52 ]
Start: i32 0
End: i32 10
Stride: i32 1
Size: 4
================================================
```

```
Matrix B
================================================
Name:   %k.0 = phi i32 [ 0, %for.body37 ], [ %inc53, %for.inc52 ]
Start: i32 0
End: i32 10
Stride: i32 1
Size: 400
Name:   %j34.0 = phi i32 [ 0, %for.body33 ], [ %inc56, %for.inc55 ]
Start: i32 0
End: i32 100
Stride: i32 1
Size: 4
================================================
```

Listing 7: Pattern detected for Listing 3 (size of index is in bytes)

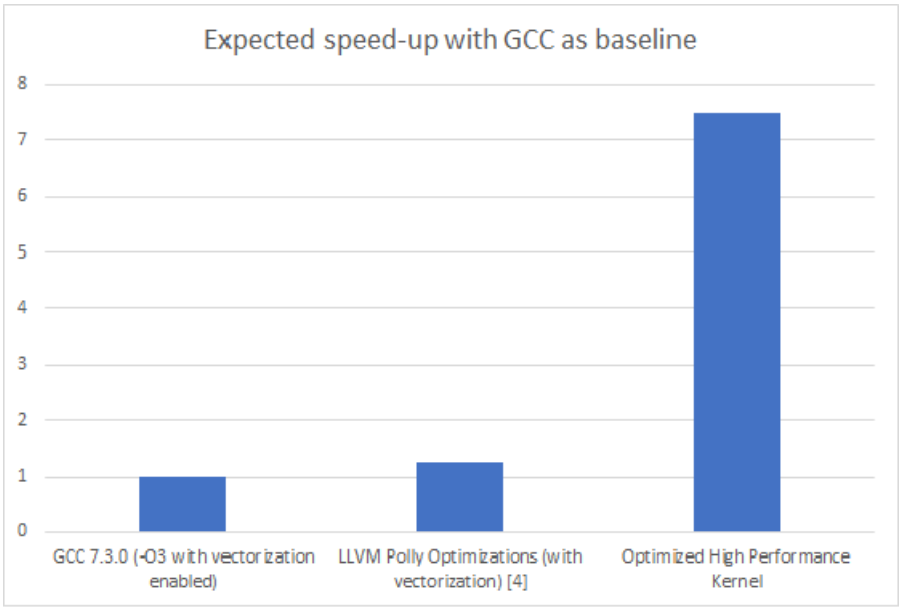

Figure 4: Expected speed-up of high performance kernel with respect to Polly and GCC

The expected speed-up with respect to GCC and LLVM Polly is shown in (4). Also, since our implementation can detect matrix multiplication of all the listings 1-5 presented above, we can achieve a performance of nearly 80x over both Polly and GCC (with Listing 1's implementation) as the timing for the high performance kernel will remain the same.

6 Surprises and Lessons Learned

Since we need a sorted order of the indices to be passed to the kernel, we need to do run-time evaluation for their actual sizes, sort them for ordering and then perform the instrumentation. This is an implementation bottleneck and does not break the analysis framework.

The first odd surprise was the difficulty of extracting information from special LLVM instructions that describe indexing using "[]". They have no simple protocol for extracting dimension information

and require a sort of work around to find the dimensions of the matrix. Arrays referenced with pointer arithmetic where surprisingly easier to deal with as they include the multiply instruction with the index variable.

The difficulty between detecting candidates for pattern matching and actually verifying them was surprising. As seen in our approach finding a tensor operation is relatively easy while checking that is meets the definition of the operation is much more work. Also our best effort for verification algorithm doesn't work on matrix multiply algorithms that mess with the definition like, the method of four Russians or Strassen's algorithm. Methods that do not make it apparent that the dot product of a row in A and a column in B goes to a single element in C, are difficult to verify.

Although this would be taken care of in future work it should be surprisingly easy to generalize our approach to detecting other idioms. Tensor operations are mainly a definition of what part of the input have an operator applied and where they are stored in the output. So tracking this information can find the pattern in the code. The difficulty will be setting the analysis to find a wider variety of idioms as each has a different set of parameters being searched for.

7 Conclusions and Future Work

We created a system for identifying and replacing tensor operations in LLVM with high level DSL descriptions, which are again replaced by high performance code. Our LLVM pass works for many different implementations of matrix multiply and generalizing to more operations would be the aim of future work. What it can verify is limited by the ranges that can be analyzed, and so we do not verify code with recursive tensor operations. Our work outputs code that runs 7x faster for matrix multiply than similar optimizing compilers, such as GCC, and Polly.

In the future this work could be generalized to detect and verify additional common idioms. The framework should easily generalize to handpicked tensor operations. Those that fit well into the idea of finding what input gets transformed and goes where into the output. Convolutions and stencil operations would be simple to implement by simply checking for a different format of input going to a single element in the output. Generalizing further would open up an extremely large set of programming idioms by finding traversals or other recursive structures. These again would likely be relatively easy to find specific patterns and much harder for the framework to give the a DSL expression for a large set of patterns.

8 Contribution

Rajnish Aggarwal: 50%

Peter Oostema: 50%

9 Acknowledgement

We would like to thank Mr. Chris Fallin for his guidance. This work has been derived from his PhD dissertation. We would also like to thank Professor Phil Gibbons and Abhilasha Jain for their time and guidance.

References

[1] Christopher Ian Fallin. *Finding and Exploiting Parallelism with Data-Structure-Aware Static and Dynamic Analysis*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2019.

[2] Roman Gareev, Tobias Grosser, and Michael Kruse. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):34, 2018.

[3] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[4] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *ACM SIGPLAN Notices*, volume 51, pages 711–726. ACM, 2016.

[5] Aditya Kumar and Sebastian Pop. Scop detection: a fast algorithm for industrial compilers. In *IMPACT 2016 Sixth International Workshop on Polyhedral Compilation Techniques*, 2016.

[6] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):12, 2016.

[7] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. *Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code*, volume 50. ACM, 2015.

[8] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.