Loop Diagonalization

Vedant Kumar
vsk@berkeley.edu

August 2, 2014

1 Abstract

The eigenvalue equation $Mv = \lambda v$ is a powerful statement about matrix and scalar multiplications. In the context of compiler optimization, we can use it to transform linearizable loops which run in $O(n)$ steps into matrix operations which run in $O(\log n)$ steps. This paper defines loop diagonalization (the process of rewriting linearizable loops as efficient eigendecompositions), discusses the implementation of this optimization with LLVM, and discusses the advantages and limitations of this method.

2 Overview

Roughly speaking, a linearizable loop may equally well be represented as a matrix M. Let the loop's state variables reside in a vector v. Each iteration of the loop effects the update $v' = Mv$.

It follows inductively that if one iteration of a linearizable loop gives Mv, n iterations of the loop can be simulated by computing $M^n v$. Performing this matrix exponentiation naively requires up to $O(nm^3)$ scalar multiplictaions.

Linear algebra comes to the rescue. Recall the equation $M = PDP^{-1}$. The column vectors of P are the eigenvectors of M, and the diagonal matrix D contains the corresponding eigenvalues. We can find this eigendecomposition for any square M so long as P is invertible.

It is known that $M^n = PD^nP^{-1}$, and that D^n can be computed in $O(m \log n)$ steps using the repeated squaring algorithm. With these facts we can replace linearizable loops with fast matrix operations.

3 Diagonalization

The following definitions are useful:

Def. Linearizable Loop: A tuple (L, M, v) where L is a list of basic blocks with a preheader and one backedge, M is a m by m square matrix with m linearly independent eigenvectors, and v is a vector of state variables. Let $v' = Mv$: the *only* permissible instructions in L are the arithmetic operations which contribute to the goal of updating v to v' with no other side effects, excepting the branch and jump instructions required to construct a loop.

Def. Loop Diagonalization: Given a program containing a linearizable loop (L, M, v), replace L with a single basic block that computes $M^n v$ using $PD^nP^{-1}v$.

Loop diagonalization transforms functions in $O(nm^3)$ into two matrix multiplications and a diagonal matrix exponentiation, which is in $O(m^3 + m \log n)$. This can result in an appreciable increase in program efficiency, as will be shown.

4 The Fibonacci Example

Consider the iterative procedure for computing the n-th Fibonacci number:

```
function FIB(n)
    a ← 1
    b ← 1
    for i ∈ [2...n] do
        tmp ← a
        a ← b
        b ← tmp + b
    end for
    return b
end function
```

The state vector is $v = (a, b)^T$. Initially, $v_0 = (1, 1)^T$. After each iteration of the loop, $a' = b$ and $b' = a + b$. These linear combinations are encoded by:

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Let $\phi = \frac{1+\sqrt{5}}{2}$. The eigenvalues of M are ϕ and $1 - \phi$. The corresponding eigenvectors are $(1, \phi)^T$ and $(\phi, -1)^T$. M is diagonalizable because its eigenvalues are distinct and its eigenvectors are linearly independent:

$$M = \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix} \begin{bmatrix} \phi & 0 \\ 0 & 1-\phi \end{bmatrix} \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix}^{-1}$$

Given $M = PDP^{-1}$, $M^n = PD^nP^{-1}$:

$$M^n = \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix} \begin{bmatrix} \phi & 0 \\ 0 & 1-\phi \end{bmatrix}^n \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix}^{-1}$$
$$= \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix} \begin{bmatrix} \phi^n & 0 \\ 0 & (1-\phi)^n \end{bmatrix} \begin{bmatrix} 1 & \phi \\ \phi & -1 \end{bmatrix}^{-1}$$

Now the Fibonacci loop has been diagonalized:

function FIB(n)
    $(a, b)^T \leftarrow PD^{n-1}P^{-1}(1, 1)^T$
    return b
end function

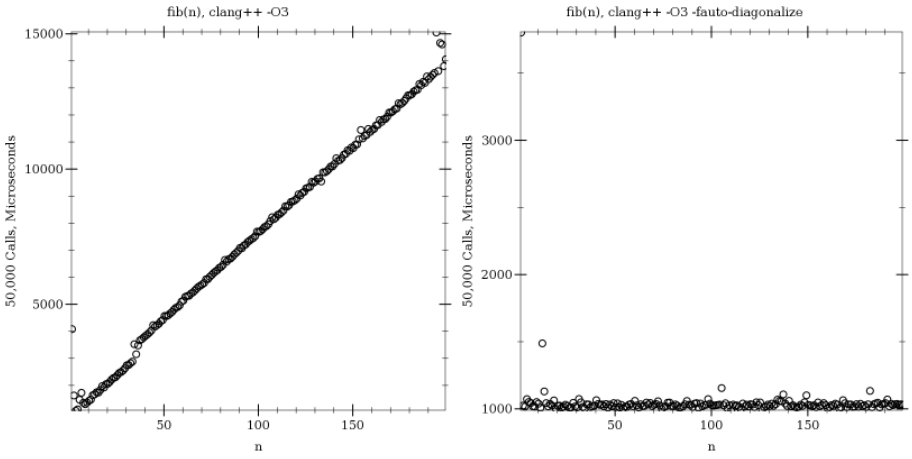
5 Implementation

I implemented automatic loop diagonalization using the LLVM compiler infrastructure. This involved creating an instance of `llvm::LoopPass` (`ADPass`) which can be loaded from a dynamic library. `ADPass` determines if a loop is linearizable by filtering out unsupported instructions. It then builds the loop matrix M by performing a depth-first search on the exit phi-nodes in the loop. This search allows the pass to determine the coefficients of every linear combination of v in the loop, which are exactly the entries of M. Next, the pass uses the `Eigen` library to find the eigendecomposition of M. Finally the pass deletes the original loop, inserts newly generated bitcode corresponding to the decomposition into the program, and rewires the values flowing into the exit phi-nodes. The source code is available under a non-restrictive free software license here.

6 Testing

To test the loop diagonalization algorithm, I created a test suite of iterative processes written in C++ and compiled them at the highest optimization levels available in LLVM 3.2 and clang++ on Linux x86-64. I then created alternate versions of the binaries which were post-processed with `ADPass`. `ADPass` was able to diagonalize basic loops but depended upon pre-processing by the compiler to eliminate stack-spills and canonicalize loops. It produced good results for all tested programs. I did not attempt to analyze loops which I knew the pass cannot optimize.

Consider the iterative Fibonacci procedure discussed in a previous section. I measured the time it took to call $fib(0)$, ..., $fib(200)$ 50,000 times each. I first tried this with the normal binary and then compared its performance to the auto-diagonalized binary. As expected, the normal program (left) scales linearly with the size of the input, whereas the diagonalized program (right) exhibits $O(\log n)$ performance.



7 Discussion

Matrix decompositions have applications in many fields, but their use in compiler loop optimizations appears to be novel [1]. It isn't clear if this optimization pays for its implementation complexity.

Leaving the issue of practicality aside, there are important technical limitations to `ADPass`. The most severe restriction is that function calls, branches, and other instructions with unpredictable side effects cannot occur within a linearizable loop. In general it is not possible to lift this draconian restriction without solving undecidable problems such as 'when exactly is this branch taken?' and 'does this function call return?'. Another problem is that there is inherent numerical instability in IEEE 754 floating point numbers, which means that the effects of auto-diagonalization are not always completely transparent.

With that said, we're left with a somewhat academic optimization that produces some interesting results. It allows linearizable $O(nm^3)$ processes to run in $O(m^3 + m \log n)$, using the simple magic trick $Mv = \lambda v$.

8 References

1. LLVM, `http://llvm.org/`.

2. Eigen, v3. Gaël Guennebaud and Benoît Jacob and others. `http://eigen.tuxfamily.org`.

[1] This might have been true when it was written in 2012, but papers on abstract acceleration of general linear loops have been published since then.