Loop Fusion, Loop Distribution and Their Place in the Loop Optimization Pipeline

April 9, 2019

Kit Barton, IBM Canada
Johannes Doerfert, Argonne National Labs
Hal Finkel, Argonne National Labs
Michael Kruse, Argonne National Labs
Agenda

- Loop Fusion Current Status
- Loop Distribution
- Data Dependence Graph
- Loop Optimization Pipeline
- Next Steps
Loop Fusion

Combine two (or more) loops into a single loop

```c
for (int i=0; i < N; ++i) {
    A[i] = i;
}
for (int j=0; j < N; ++j) {
    B[j] = j;
}
```

Motivation
- Data reuse, parallelism, minimizing bandwidth, ...
- Increase scope for loop optimizations

Our Goals
1. Way to learn how to implement a loop optimization in LLVM
2. Starting point for establishing a loop optimization pipeline in LLVM

Requirements
In order for two loops, $L_j$ and $L_k$ to be fused, they must satisfy the following conditions:

1. $L_j$ and $L_k$ must be adjacent
2. $L_j$ and $L_k$ must iterate the same number of times
3. $L_j$ and $L_k$ must be control flow equivalent
4. There cannot be any negative distance dependencies between $L_j$ and $L_k$
Loop Fusion – Current Status

Initial patch for *Basic Loop Fusion*: [https://reviews.llvm.org/D55851](https://reviews.llvm.org/D55851)

Approved, but waiting confirmation of one remaining review comments

Improvements to *basic loop fusion* currently under development

1. Require rotated loops
2. Handling of guarded loops
3. Merging latch blocks during fusion
Loop Rotation

Convert a loop into a do/while style loop
Canonicalize loop latch to have a single successor

Motivation
– Canonicalize loop latch to have a single successor
– Makes analysis for loop fusion easier because loop structure is canonical
– Makes mechanics of fusing loops easier because the latch and exiting blocks are the same

https://reviews.llvm.org/D22630
Loop Rotation – Guarded Loops

When compiler cannot prove loop body will execute at least once, it inserts a guard.
It also inserts a loop epilogue, that will be executed once after the body is finished executing.

Example:
```c
int A[100];
void example(long N) {
    for (int i = 0; i < N; ++i)
        A[i] = i;
}
```
Loop guards cause loop preheaders to be no longer control flow equivalent
Loop epilogue cause loops to no longer be adjacent
Loop rotate summary

We want to focus on fusing rotated loops only
  Work on a canonical form of loops makes the implementation for fusion simpler

In some cases, loop rotate is creating a guard block
  Necessary because it is creating a do loop
  If it cannot prove the loop should execute at least once iteration, it needs a guard at the beginning
  Guard block will make loop preheaders not control flow equivalent, meaning they cannot be fused

When a guard block is created, a loop epilogue is also created
  Provides a location to sink statements that only need to be run once, after the loop body finishes
  Epilogue block makes loops not adjacent (temporary limitation of fusion)

Running SimplifyCFG after loop rotate will cleanup (empty) epilogue block, but not (valid) guard
Possible solutions for loop fusion

Make guard block and epilogue block part of the canonical structure of loops

In cases where no guard is necessary, add a “trivial” guard block (i.e., if (1) { } )

Modify LoopSimplify to always ensure a guard block is present

Add similar interfaces for other components to allow getLoopGuard, etc.

Modify Loop Fusion to use guard block for control flow equivalence checks and adjacency check

Modify LoopFusion to handle cases for guarded loops and non-guarded loops separately

If a guard is present:
1. Use the guard block for the control flow equivalence checks
2. Use guard block successors for adjacency check (if successor of a loop guard is another loop guard, loops are adjacent)
3. Check if guard blocks have same branch and can be merged

If no guard is present:
1. Use preheader for control flow equivalence checks (done today)
2. Use exit block to check for adjacency (if exit block of loop is another loop preheader/guard, loops are adjacent)
Loop Fusion – merging latches

Current implementation of fusion simply changes edges in CFG when fusing:

• This prevents fusion in nested loops because inner loops are not adjacent after fusing outer loops

• This can be improved by actually merging the blocks from the two loops

• Allows for subsequent fusion of nested loops because inner loops are now control flow equivalent

• Only implemented for rotated loops!
## Number of Loops Fused

### SPEC 2017 Basic Fusion

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Candidates for Fusion</th>
<th>Loops Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench_r</td>
<td>487</td>
<td>0</td>
</tr>
<tr>
<td>gcc_r</td>
<td>2019</td>
<td>2</td>
</tr>
<tr>
<td>namd_r</td>
<td>1393</td>
<td>0</td>
</tr>
<tr>
<td>parest_r</td>
<td>9385</td>
<td>1</td>
</tr>
<tr>
<td>povray_r</td>
<td>294</td>
<td>0</td>
</tr>
<tr>
<td>lbm_r</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>omnetpp_r</td>
<td>198</td>
<td>0</td>
</tr>
<tr>
<td>x264_r</td>
<td>797</td>
<td>0</td>
</tr>
<tr>
<td>blender_r</td>
<td>4190</td>
<td>6</td>
</tr>
<tr>
<td>deepsjeng_r</td>
<td>99</td>
<td>13</td>
</tr>
<tr>
<td>imagick_r</td>
<td>2710</td>
<td>2</td>
</tr>
<tr>
<td>nab_r</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>xz_r</td>
<td>70</td>
<td>0</td>
</tr>
</tbody>
</table>

### SPEC 2017 Basic Fusion + Merge Latches

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Candidates for Fusion</th>
<th>Loops Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench_r</td>
<td>479</td>
<td>0</td>
</tr>
<tr>
<td>gcc_r</td>
<td>1930</td>
<td>1</td>
</tr>
<tr>
<td>namd_r</td>
<td>1393</td>
<td>0</td>
</tr>
<tr>
<td>parest_r</td>
<td>9349</td>
<td>1</td>
</tr>
<tr>
<td>povray_r</td>
<td>287</td>
<td>0</td>
</tr>
<tr>
<td>lbm_r</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>omnetpp_r</td>
<td>195</td>
<td>0</td>
</tr>
<tr>
<td>x264_r</td>
<td>765</td>
<td>0</td>
</tr>
<tr>
<td>blender_r</td>
<td>4079</td>
<td>6</td>
</tr>
<tr>
<td>deepsjeng_r</td>
<td>99</td>
<td>13</td>
</tr>
<tr>
<td>imagick_r</td>
<td>2700</td>
<td>2</td>
</tr>
<tr>
<td>nab_r</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>xz_r</td>
<td>69</td>
<td>0</td>
</tr>
</tbody>
</table>

### SPEC 2017 Basic Fusion + Merge Latches + Guard Handling

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Candidates for Fusion</th>
<th>Loops Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench_r</td>
<td>480</td>
<td>0</td>
</tr>
<tr>
<td>gcc_r</td>
<td>1937</td>
<td>4</td>
</tr>
<tr>
<td>namd_r</td>
<td>1393</td>
<td>0</td>
</tr>
<tr>
<td>parest_r</td>
<td>9349</td>
<td>1</td>
</tr>
<tr>
<td>povray_r</td>
<td>286</td>
<td>0</td>
</tr>
<tr>
<td>lbm_r</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>omnetpp_r</td>
<td>195</td>
<td>0</td>
</tr>
<tr>
<td>x264_r</td>
<td>765</td>
<td>1</td>
</tr>
<tr>
<td>blender_r</td>
<td>4078</td>
<td>5</td>
</tr>
<tr>
<td>deepsjeng_r</td>
<td>99</td>
<td>14</td>
</tr>
<tr>
<td>imagick_r</td>
<td>2700</td>
<td>3</td>
</tr>
<tr>
<td>nab_r</td>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>xz_r</td>
<td>69</td>
<td>1</td>
</tr>
</tbody>
</table>
Loop Distribution

Separate a loop nest into two (or more) loop nests

```c
for (int i=0; i < N; ++i) {
    A[i] = B[i] * r;
    for (int j=1; j < N; ++j) {
        C[i][j] = D[i][j-1] / A[i];
        E[i][j] = E[i][j] * A[i];
    }
}
```

```c
for (int i=0; i < N; ++i) {
    A[i] = B[i] * r;
    for (int j=1; j < N; ++j) {
        C[i][j] = D[i][j-1] / A[i];
        E[i][j] = E[i][j] * A[i];
    }
}
```

Motivation

– Data reuse, parallelism, minimizing bandwidth, ...
– Improve effectiveness of subsequent loop optimizations
  • Creating perfect nests, removing loop carried dependencies, etc

Our Goals

1. Continue to improve and strengthen loop optimizations in LLVM
2. Create common infrastructure and utilities that can be used for many loop optimizations
Current implementation of loop distribution in LLVM

Provides mechanics of distributing *inner* loops

Focuses entirely on distributing loops for vectorization
  Uses the LoopAccessInfo classes, which were developed for loop vectorization
  Only tries to distribute inner loops

Added in the pass pipeline but not enabled by default
Heuristic to determine how to distribute loops

We are creating heuristics that can be used to determine how to distribute a loop nest.

Heuristics are based on the heuristics used in IBM’s XL Compilers.

Use two key data structures not currently available in LLVM:
- Data Dependence Graph
- Affinity Graph
Data Dependence Graph

Directed multigraph that represents data dependencies between statements
   Nodes correspond to either a single statement or a group of statements
   Edges represent data dependencies between nodes

A directed edge from node $N_i$ to $N_j$ represents a data dependency from $N_i$ to $N_j$

Edges can have attributes that represent the type of data dependence and a distance/direction vector
```
int A[100];
int B[100];
void example() {
    for (int i = 0; i < 100; ++i)
}
```
Affinity Graph

Undirected weighted graph
   Nodes correspond to strongly connected components in the DDG
   Edges represent *affinity* between the nodes

Nodes in the graph correspond to strongly connected components in the DDG
   Nodes also have characteristics that are relevant to loop distribution including:
      – Platform specific metrics such as register requirements, functional unit requirements and prefetchable data streams
      – Also indicates self dependence for parallelization and vectorization

Edges represent affinity between the nodes
   Currently only measure of affinity is data reuse
How to distribute loops

Use greedy algorithm to distribute nodes in the affinity graph

Nodes are gathered in increasing order of desirability

Decision about grouping nodes is based on:
1. affinity graph
2. data dependencies
3. desirability based on node attributes
   If grouping nodes together exceeds platform-specific threshold or adds data dependencies then nodes should not be grouped together
Placing loop fusion and distribution in the loop opt pipeline

Fuse early, primarily to create opportunities for other loop optimizations
  Use loop rotate to create (guarded) do loops
  Rely on Loop Simplify to put loops in canonical form

Run loop distribution later, after fusion
  Do we want to rely on distribution “undoing” decisions made by fusion?

However, since both fusion and distribution can be run for multiple criteria, maybe want to run them multiple times
  1. Run early as an optimization-enable pass
  2. Run late to make platform-specific optimization decisions (with target-specific overrides)
Next Steps

Loop Fusion
- Converge on direction for loop rotation and guarded loops
- Move intervening code from between loops to make them adjacent
- Improving dependence analysis
- Placing loop fusion in the pipeline and enable by default

Loop Distribution
- Post patches for DDG
- Post patches for initial loop distribution
- Discuss interface for affinity graph