中端优化

前置分析

别名分析

内存依赖分析

基指针分析

具体优化

Mem2Reg

RemovePhi

ConstIdx2Value

将常下标的数组访问内联成值。

LocalArrayLift

将局部数组提升到全局，减少栈分配空间以及访存次数

SoraPass

将聚合的数组拆分成多个标量寄存器，减少访存次数

DeadArgEliminate

消除函数中“无用”的参数

DeadCodeEliminate

消除对于结果不产生影响的分支跳转

DeadCodeEliminate

消除对于结果不产生影响的死代码

DeadLoopEliminate

消除对于结果不产生影响的循环

DeadRetEliminate

消除函数中无用的返回值

RemoveBlocks

消除无用的基本块与不可达的基本块，简化控制流图

SimplifyCFG

简化控制流图：

- 合并基本块
- 消除只有跳转语句的基本块
- 修改跳转目标，节约跳转语句的次数

冗余存储的消除

消除冗余的存储，减少访存次数，具体包括：

- 编译期间能判断无法被使用的store

冗余加载的消除

消除冗余的加载，减少访存次数，具体包括：

- 编译期间能确定的Load的值
- 已经存在标量寄存器中Load的值

无用指令的消除

消除无用的指令，减少指令数目

FuncIncline

函数内联，将函数内联到调用处，减少函数调用过程的开销

TailCall2Loop

尾递归优化，将尾递归优化为循环，减少函数调用的开销

分支预测

对于循环中的分支，根据循环的特性进行分支预测，为后端基本块的排布服务

LoopSimplifyForm

将建立好的循环规约为规范形式，具体包括：

- 唯一的preheader
- 唯一的latch
- 所有的exit被header支配

将循环规约后方便其他循环优化变换

LCSSA

在循环的出口块插入phi指令，将循环内部变量的外部使用限定为LCSSA，方便循环优化变换

LoopUnswitching

循环分支提升，尽量将循环中的分支提取出来，为其他优化创造新的机会

LoopUnroll

分为ConstLoopUnroll和LoopUnroll

- ConstLoopUnroll：对于编译期可确定次数循环，我们首先预估展开后大小，若不超过阈值，采用全展开策略
- LoopUnroll：对于循环次数不固定的循环，目前采用4路展开策略

LVN

局部值编号。按照支配树进行替换，不需要多跑GCM保证正确

GVN

全局值编号，用于消除全局冗余计算，该算法较为经典不再赘述；进行GVN pass后需要再进行GCM pass以保证正确

GCM

全局代码移动。将代码在不破坏支配顺序的前提下移动到循环深度浅、支配深度深的地方

SCEV

识别循环中的归纳变量

运算强度削减

重结合

约束削减

值域分析与值域折叠

可以服务后端的乘除优化，因为如果能默认被除数是正数的话直接用简单的位运算就能解决。

LoopUnSwitching

StillLoopMove

不变子循环外提。类似一个大循环里套着一个小的子循环，这个小子循环

Sum2Mul

求和变成乘法，比较局限的一个优化

后端优化

带权寄存器分配

带cost的图着色寄存器分配策略，通过中端传来的循环信息在后端建立循环结构，并且根据该信息，以循环的深度和执行次数综合为寄存器进行赋值，可以保证在不得不溢出的情况下优先溢出循环深度较浅、执行次数较少的寄存器，尽可能地减少溢出带来的损失。

循环不变量外提

虽然中端实现了循环不变量外提，但是考虑到后端与中端的差异性，该优化在后端依旧大有可为。

具体来讲，后端会从中端提取所有循环块的信息，并获取他的所有header、entering和exiting，然后遍历所有循环内部的块，将所有类似于
li、lla等数值不会有任何变化的指令外提到循环外，以削减每次循环的强度。

块排序

将跳转概率比较大的两个块放在挨着的地方

块内联

将直接j跳转的块内联，可以减少j指令的开销。

控制流图简化

在中端和后端的前期，为了保证指令格式的简洁，约定每一个块中br指令一定在块的最末尾。但是在真正执行的时候会造成很多负面影响。因此需要对cfg进行一定程度的简化。

重定向

对于所有仅有一个j指令的块，将指向它的块的指令所指向的块都换成该j指令指向的块

j省略

经过块重排，保证跳转概率较大的几个块大概率会依次排布，如果j的目标块就是下一个紧邻的块，就将j指令省略。

访存优化

考虑到计算的局部性，可能在一个块内的集中部分会出现多次访存同一个地址的行为。在后端建立一个访存表，跟踪记录所有内存的加载情况和寄存器关联。如果某个寄存器已经存放了某一处内存的值，那么对这个内存的取就可以转化为寄存器间的移动，从而减少访存。

计算优化

在后端需要尽可能多的方式对计算进行优化，减少寄存器以减少寄存器分配的压力。

其中的优化策略包括：

- 立即数加载复用(如果有较短区间内两次加载一样的数值，那么就将两条指令所用的寄存器合并)
- 地址加载复用(如果有较短区间内两次加载一样的地址，那么就将两条指令所用的寄存器合并)
- srai+slli等位运算可以用更短的方式(如andi)优化。

乘除常数优化

乘除法指令的代价非常巨大，因此需要考虑用一系列加法和移位指令来替代常数作为参数的乘除指令

乘法优化

将乘法规约为x*n，其中n为常数的格式。针对该n选择不同的指令组合。类似dp表的格式，在程序初始化的时候计算在乘法代价内(let mul_cost = 3)条指令可能产生的组合，例如3可以视为两次位移求和。最终可以根据其n生成一个乘法方案(mul_plan)，供codegen阶段翻译生成

除法/取余优化

可以参考论文Division by Invariant Integers using Multiplication

注意在正数和被除数是2的幂次的情况下有奇效。

而且取余如果是正数并且被除数是2的幂次的情况有特殊优化空间(andi)可以自己试试。