编译器优化那些事儿（1）：SLP矢量化介绍

Introduction

Superword Level Parallelism (SLP)矢量化是llvm auto-vectorization中的一种，另一种是loop vectorizer，详见于Auto-Vectorization in LLVM[1]。它在2000年由Larsen 和 Amarasinghe首次作为basic block矢量化提出。SLP矢量化的目标是将相似的独立指令组合成向量指令，内存访问、算术运算、比较运算、PHI节点都可以使用这种技术进行矢量化。它和循环矢量化最大的差异在于，循环矢量化关注迭代间的矢量化机会，而SLP更关注于迭代内basic block中的矢量化的机会。

一个简单的小例子 case.cpp[1]：

```
1. void foo(float a1, float a2, float b1, float b2, float *A) {
2.   A[0] = a1*(a1 + b1);
3.   A[1] = a2*(a2 + b2);
4.   A[2] = a1*(a1 + b1);
5.   A[3] = a2*(a2 + b2);
6. }
```

命令：clang++ case.cpp -O3 -S ； SLP在clang中是默认使能的，可以看到汇编中已出现使用矢量寄存器的fadd和fmul。

```
mov     v0.s[1], v1.s[0]
mov     v2.s[1], v3.s[0]
fadd    v1.2s, v0.2s, v2.2s
fmul    v0.2s, v1.2s, v0.2s
mov     v0.d[1], v0.d[0]
str     q0, [x0]
ret
```

如果编译命令中加上选项-fno-slp-vectorize 或者 -mllvm -vectorize-slp=false 关闭该优化，则只能得到标量的版本。

```
fadd    s2, s0, s2
fadd    s3, s1, s3
fmul    s0, s2, s0
fmul    s1, s3, s1
stp     s0, s1, [x0]
stp     s0, s1, [x0, #8]
ret
```

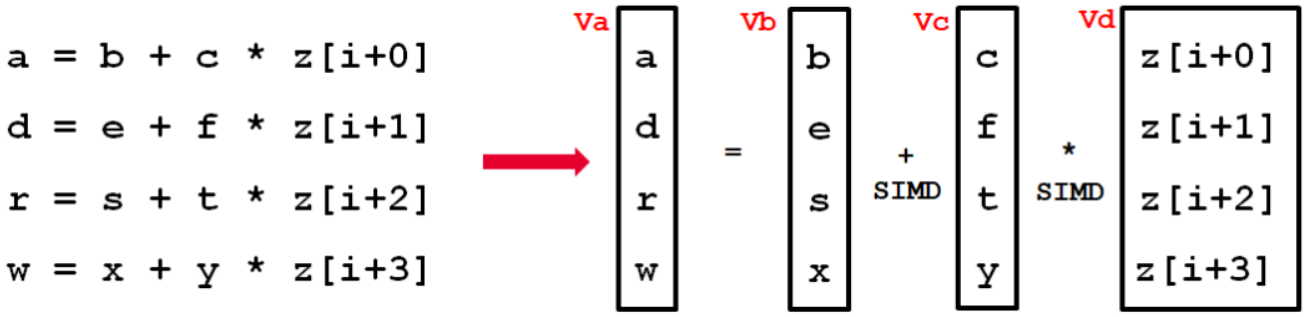
让我们来跟随《Exploiting Superword Level Parallelism with Multimedia Instruction Sets》[2]这篇经典论文来探究一下SLP矢量化的奥秘。

原始SLP算法介绍

1. 概述

论文中用一张图来解释了SLP要做的事情：

原始SLP例子[2]



这四条语句中的位置相对应的操作数，比如(b,e,s,x)可以pack到一个向量寄存器 Vb 中，同样的，(c,f,t,y)可以pack到 Vc，(z[i+0]~z[i+3])可以到 Vd。然后可以利用simd指令进行相应的矢量化计算。最后根据 Va 中 (a,d,r,w) 的被使用方式，可能还需要将他们从向量寄存器中load出来，称为unpack。

所以，如果pack操作数的开销 + 矢量化执行的开销 + unpack操作数的开销小于原本执行的开销，那就证明SLP矢量化具有性能收益[3]。

2. 优化场景

为了进一步说明SLP和循环矢量化在优化场景上的差异，论文[2]中给了两个例子(可以通过 https://godbolt.org/z/EWr4zTc3P直接查看汇编情况)。

(1)对于原始循环 a，既可以通过 scalar expansion (a method of converting scalar data to match the dimensions of vector or matrix data.) 和 loop fission (the opposite of loop fusion: a loop is split into two or more loops.) 后被转换为可以进行循环向量化的形式 b，一个induction和一个reduction；也可以经过unroll和rename之后变为 d 这样的形式，做SLP。但其实由于论文比较老了，目前llvm编译器对于a这样形式的循环可以直接做矢量化。

(a) Original loop.

```
1. for (i=0; i<16; i++) {
2.   localdiff = ref[i] - curr[i];
3.   diff += abs(localdiff);
4. }
```

(b) After scalar expansion and loop fission.

```
1. for (i=0; i<16; i++) {
2.   T[i] = ref[i] - curr[i];
3. }
4.
5. for (i=0; i<16; i++) {
6.   diff += abs(T[i]);
7. }
```

(c) Superword level parallelism exposed after unrolling.

```
 1. for (i=0; i<16; i+=4) {
 2.   localdiff = ref[i+0] - curr[i+0];
 3.   diff += abs(localdiff);
 4.
 5.   localdiff = ref[i+1] - curr[i+1];
 6.   diff += abs(localdiff);
 7.
 8.   localdiff = ref[i+2] - curr[i+2];
 9.   diff += abs(localdiff);
10.
11.   localdiff = ref[i+3] - curr[i+3];
12.   diff += abs(localdiff);
13. }
```

(d) Packable statements grouped together after renaming.

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff += abs(localdiff0);
  diff += abs(localdiff1);
  diff += abs(localdiff2);
  diff += abs(localdiff3);
}
```

(2)但是对于如下例子，循环向量化需要将do while循环转换为for循环，恢复归纳变量，将展开后的循环恢复为未展开的形式(loop rerolling)。而SLP只需要将计算 dst[{0, 1, 2, 3}] 的这四条语句组合成一条 使用向量化指令的语句即可。

```
do {
  dst[0] = (src1[0] + src2[0]) >> 1;
  dst[1] = (src1[1] + src2[1]) >> 1;
  dst[2] = (src1[2] + src2[2]) >> 1;
  dst[3] = (src1[3] + src2[3]) >> 1;

  dst += 4;
  src1 += 4;
  src2 += 4;
}
while (dst != end);
```

看到这里，可以了解到哪些是SLP的优化机会。论文中提出了一种简单的算法来实现，简而言之是通过寻找independent(无数据依赖)、isomorphic(相同操作)的指令组合成一条向量化指令。

那么如何找呢?

3. 算法描述

作者注意到如果被 pack 的指令的操作数引用的是相邻的内存，那么特别适合 SLP 执行。所以核心算法就是从识别 adjacent memory references 开始的。

当然寻找这样的相邻内存引用前也需要做一些准备工作，主要是三部分：(1) Loop Unrolling；(2) Alignment analysis；(3) Pre-Optimization(主要是一些死代码和冗余代码消除)。具体不展开讲。

接下来我们来看看核心算法，主要分为以下4步：

(1)Identifying Adjacent Memory References

(2)Extending the PackSet

(3)Combination

(4)Scheduling

伪代码[4]是：

$$
\begin{aligned}
&SLP\_extract : BasicBlockB \rightarrow BasicBlock \\
&\quad PackSetP \leftarrow \emptyset \\
&\quad P \leftarrow find\_adj\_refs(B, P) \\
&\quad P \leftarrow extend\_packlist(B, P) \\
&\quad P \leftarrow combine\_packs(P) \\
&\quad return \quad schedule(B, [], P)
\end{aligned}
$$

(1)第一步 find_adj_refs

先来看第一步：Identifying Adjacent Memory References

$$
\begin{aligned}
&find\_adj\_refs : BasicBlockB \times PackSetP \rightarrow PackSet \\
&\quad foreach\ Stmt\ s \in B\ do \\
&\quad\quad foreach\ Stmt\ s' \in B\ where\ s \neq s'\ do \\
&\quad\quad\quad if\ has\_mem\_ref(s) \wedge has\_mem\_ref(s')\ then \\
&\quad\quad\quad\quad if\ adjacent\ (s, s')\ then \\
&\quad\quad\quad\quad\quad Int\ align\ \leftarrow get\_alignment(s) \\
&\quad\quad\quad\quad\quad\quad if\ stmts\_can\_pack(B, P, s, s', align)\ then \\
&\quad\quad\quad\quad\quad\quad\quad P \leftarrow P \cup \{\langle s, s' \rangle\} \\
&\quad return\ P
\end{aligned}
$$

函数 find_adj_refs 的输入是 BasicBlock，输出为集合 PackSet。

遍历BasicBlock里面的任意语句对<s, s'>，如果他们访问了相邻的内存(比如s访问了arr[1]，s'访问了arr[2]),并且他俩能够pack到一起(即stmts_can_pack() 返回true)，那么将语句对<s, s'>加入集合PacketSet。

这里用到了一个辅助函数stmts_can_pack，伪代码如下：

$$
\begin{aligned}
&stmts_can_pack : BasicBlockB \times PackSetP \times \\
&\qquad\qquad Stmts \times Stmts' \times Intalign \rightarrow Boolean \\
&\quad if\ isomorphic(s, s')\ then \\
&\quad\quad if\ independent(s, s')\ then \\
&\quad\quad\quad if\ \forall \langle t, t' \rangle \in P.t\ \neq s\ then \\
&\quad\quad\quad\quad if\ \forall \langle t, t' \rangle \in P.t' \neq s'\ then \\
&\quad\quad\quad\quad\quad Int\ align_s \leftarrow get\_alignment(s) \\
&\quad\quad\quad\quad\quad Int\ align_{s'} \leftarrow get\_alignment(s') \\
&\quad\quad\quad\quad\quad if\ align_s \equiv \top \vee align_s \equiv align\ then \\
&\quad\quad\quad\quad\quad\quad if\ align_{s'} \equiv \top \vee aligns_{s'} \equiv align + data_size(s')\ then \\
&\quad\quad\quad\quad\quad\quad\quad return\ true \\
&\quad return\ false
\end{aligned}
$$

声明了可以pack到一起的条件：

a. s 和 s’是相同操作 (isomorphic)

b. s 和 s’无数据依赖 (independent)

c. s 之前没有作为左操作数出现在 PackSet 中，s’之前没有作为右操作数出现在 PackSet 中

d. s 和 s’满足对齐要求 (consistent)，即要求新加入的语句对的数据类型也是可以和已存在的语句对在内存上是对齐的

(2)第二步：Extending the PackSet

从第一步我们可以获得PacketSet，第二步沿着其中包含的语句的defs 和 uses 来扩充PacketSet。所以这一步的输入是PacketSet，输出是扩充后的PacketSet。

伪代码如下：

$$
\begin{aligned}
&extend\_packlist : BasicBlock\ B \times PackSet\ P \rightarrow PackSet \\
&\quad repeat \\
&\quad\quad PackSet\ P\_prev \leftarrow P \\
&\quad\quad foreach\ Pack\ p \in P\ do \\
&\quad\quad\quad P \leftarrow follow\_use\_defs(B, P, p) \\
&\quad\quad\quad P \leftarrow follow\_def\_uses(B, P, p) \\
&\quad until P \equiv P_{\text{prev}} \\
&\quad return P
\end{aligned}
$$

对于PacketSet中的每一个元素pack， 即语句对<s, s'>，不断执行follow_use_defs 和 follow_def_uses函数来分别在同一个BasicBlock中寻找s和s'的源操作数和目标操作数相关的语句，判断两个条件，一个是stmts_can_pack是否可以pack，另一个是根据cost model判断是否有收益，从而扩充PacketSet，直至其不能加入更多的Pack。

(3)第三步：Combination

这一步的输入为已经尽可能多的<s,s'>语句对组成的PacketSet，输出则为尽可能可以合并语句对之后的PacketSet。

那么怎么合并呢？伪代码如下：

$$
\begin{aligned}
&combine\_packs : PackSet\ P \rightarrow PackSet \\
&\quad repeat \\
&\quad PackSet\ P\_\text{prev} \leftarrow P \\
&\quad\quad foreach\ Pack\ p = \langle s_1, \ldots, s_n \rangle \in P\ do \\
&\quad\quad foreach\ Pack\ p' = \langle s'_1, \ldots, s'_m \rangle \in P\ do \\
&\quad\quad\quad if\ s\_n \equiv s\_1'\ then \\
&\quad\quad\quad\quad P \leftarrow P - \{p, p'\} \cup \{\langle s_1, \ldots, s_n, s'_2, \ldots, s'_m \rangle\} \\
&\quad until\ P \equiv P\_prev \\
&\quad return\ P
\end{aligned}
$$

对于两个Pack，p = <s1,..,sn>和 p' = <s1',..,sm'>，如果sn与s1‘相同，那么恭喜，p和 p’ 可以合并成新的 p'' = <s1,..,sn,s2',..,sm'>。

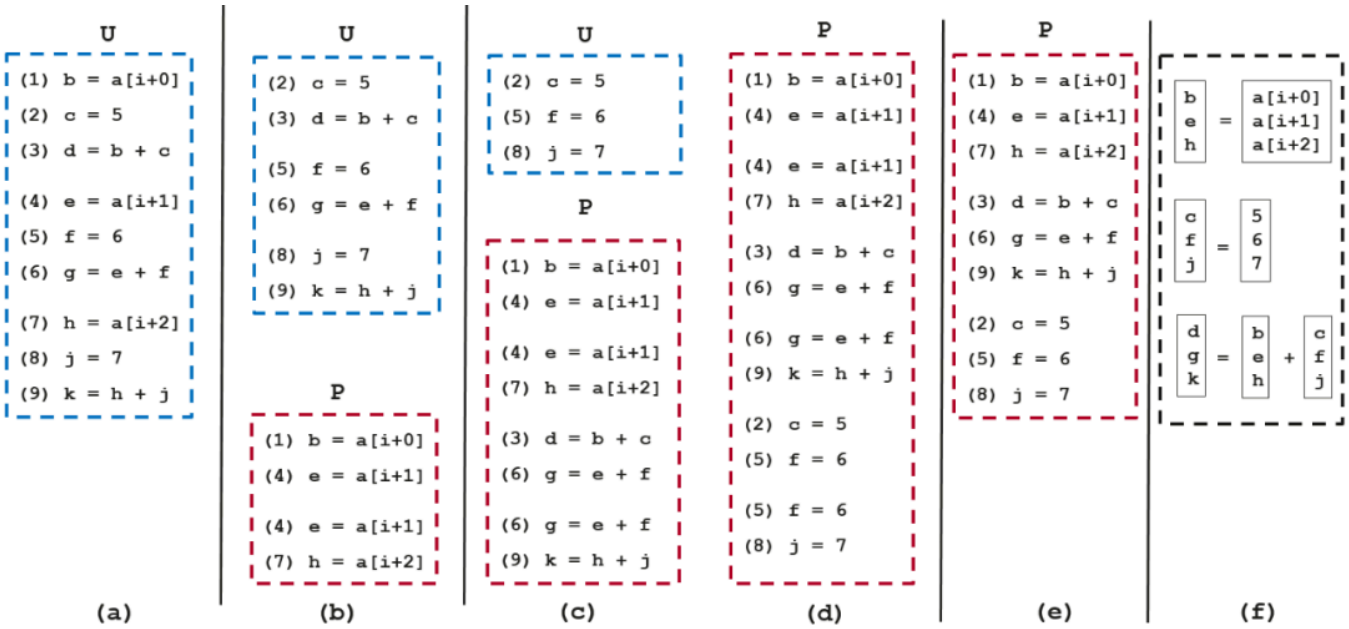
(4)最后一步：Scheduling

将PackSet中的语句对根据数据依赖关系整理成simd指令，如果是有循环依赖的pack，那么revert掉，不再对这pack里的指令矢量化。

最后输出的是包含SIMD指令的BasicBlock。

4. 一个例子

为了更好理解，论文中也给出了一个例子，我们简单过一下：



(1)初始状态，BasicBlock中指令，如(a)。

(2)执行find_adj_refs, 将<1, 4> 和 <4, 7> 加入PackSet， 如(b)。

(3)执行extern_packlist：

a. 函数follow_use_defs 去寻找对a[i+0], a[i+1], a[i+2]进行def的语句，无语句对加入 P

b. 函数follow_def_uses 去寻找对b, e, h 使用的语句，将<3, 6> 和 <6, 9> 加入P ，如(c)

c. 函数follow_use_defs 去寻找 c, f, j 进行def的语句，将<2, 5> 和 <5, 8> 加入P ，如(d)

d. 再执行一次follow_def_uses，发现没有新的语句对可以加入P了，停止。

(4)执行combine_packs：

a. <(1), (4)> 和 <(4), (7)> 合并为 <(1), (4), (7)>

b.<(3), (6)> 和 <(6), (9)> 合并为 <(3), (6), (9)>

c. <(2), (5)> 和 <(5), (8)> 合并为 <(2), (5), (8)>

合并后状态，如(e)。

(5) 执行 scheduling：注意依赖关系，比如 3 依赖于1, 2，最终状态如(f)。

Loop-Aware SLP 算法介绍

LLVM 中的 SLP vectorization ， 是受Loop-Aware SLP in GCC(by Ira Rosen, Dorit Nuzman,Ayal Zaks)[4]这篇论文启发来实现的。

```
//
// This pass implements the Bottom Up SLP vectorizer. It detects consecutive
// stores that can be put together into vector-stores. Next, it attempts to
// construct vectorizable tree using the use-def chains. If a profitable tree
// was found, the SLP vectorizer performs vectorization on the tree.
//
// The pass is inspired by the work described in the paper:
//  "Loop-Aware SLP in GCC" by Ira Rosen, Dorit Nuzman, Ayal Zaks.
```

1. 简介

Loop-Aware 方法是对基础 SLP 方法的改进，更加重视对跟Loop相关的向量化机会挖掘，其思想是 :

首先，通过循环展开将迭代间并行转换为迭代内并行，使循环体内的同构语句条数足够多；

再利用 SLP 方法进行向量发掘。当循环展开次数为 1 时，Loop-Aware 方法相当于 SLP 方法，当循环展开次数为向量化因子 (vector factor，简称 VF) 时，将同一条语句展开后的多条语句打包成向量。然而，当循环展开不合法或者并行度低于向量化因子时，Loop-Aware 方法无法简单实施。

换言之，Loop-Aware 向量化方法的实质就是当迭代内并行度较低时，通过循环展开将迭代间并行转换为迭代内并行，其要求循环的迭代间并行度较高。

一个典型例子，它可以使能以下因同构语句条数不够多而原始SLP无法矢量化的场景：

```
1. for (i=0; i<N; i++)
2. {
3.  a[2*i] = b[2*i] + x0;
4.  a[2*i+1] = b[2*i+1] + x1;
5. }
```

需要借助loop unroll，最终矢量化为以下形式：

```
1. for (i=0; i<N/2; i++)
2. {
3.  va[4*i:4*i+3] = vb[4*i:4*i+3] + {x0,x1,x0,x1};
4. }
```

2. 具体差异

与原始SLP方法的差别，论文作者在其提交给GCC的PATCH中有说明[5]，主要有以下三条：

(1)Loop-Aware SLP 着眼于Loop相关的bb块，而不是程序中的任意bb块。这么做的原因有两个，一是可以复用已有的循环矢量化的框架，二是大多数有价值的优化机会都在循环中。

(2)原始SLP算法起始于相邻内存的load或stroe，称之为seed，根据def-use扩展，并合并成Vectorize Size(VS)大小的组。Loop-Aware SLP的seed来自于interleaving analysis之后预先确定的一组相邻store，所以不需要原始算法中的合并这一步骤。具体来说就是，Loop-Aware借助loop-unroll使得在寻找seed时就能天然地找到能够刚好合并到一个向量寄存器中的指令，而原始SLP需要在合并阶段做排布。

(3)Loop-Aware SLP结合了SLP-based和Loop-based矢量化，所以对于以下循环：

```
1. for (i=0; i<N; i++)
2. {
3.  a[4*i] = b[4*i] + x0;
4.  a[4*i+1] = b[4*i+1] + x1;
5.  a[4*i+2] = b[4*i+2] + x2;
6.  a[4*i+3] = b[4*i+3] + x3;
7.
```

```
 8.  c[i] = 0;
 9. }
```

可以优化成以下形式：

```
 1. for (i=0; i<N/4; i++)
 2. {
 3.  //SLP矢量化部分
 4.  va[16*i:16*i+3] = vb[16*i:16*i+3] + {x0,x1,x2,x3};
 5.  va[16*i+4:16*i+7] = vb[16*i+4:16*i+7] + {x0,x1,x2,x3};
 6.  va[16*i+8:16*i+11] = vb[16*i+8:16*i+11] + {x0,x1,x2,x3};
 7.  va[16*i+12:16*i+15] = vb[16*i+12:16*i+15] + {x0,x1,x2,x3};
 8.  //Loop矢量化部分
 9.  vc[4*i:4*i+3] = {0,0,0,0};
10. }
```

源码阅读

SLP 是一个 transform pass，在 LLVM 14 中该pass 的实现代码位于llvm/lib/Transforms/Vectorize/SLPVectorizer.cpp 和 llvm/Transforms/Vectorize/SLPVectorizer.h中。

1. 提供的选项

(1)开源选项

选项名称	默认值	描述
vectorize-slp	TRUE	Run the SLP vectorization passes
slp-threshold	0	Only vectorize if you gain more than this number
slp-vectorize-hor	TRUE	Attempt to vectorize horizontal reductions
slp-vectorize-hor-store	FALSE	Attempt to vectorize horizontal reductions feeding into a store
slp-max-reg-size	128	Attempt to vectorize for this register size in bits
slp-max-vf	0	Maximum SLP vectorization factor (0=unlimited)
slp-max-store-lookup	32	Maximum depth of the lookup for consecutive stores.
slp-schedule-budget	100000	Limit the size of the SLP scheduling region per block
slp-min-reg-size	128	Attempt to vectorize for this register size in bits
slp-recursion-max-depth	12	Limit the recursion depth when building a vectorizable tree

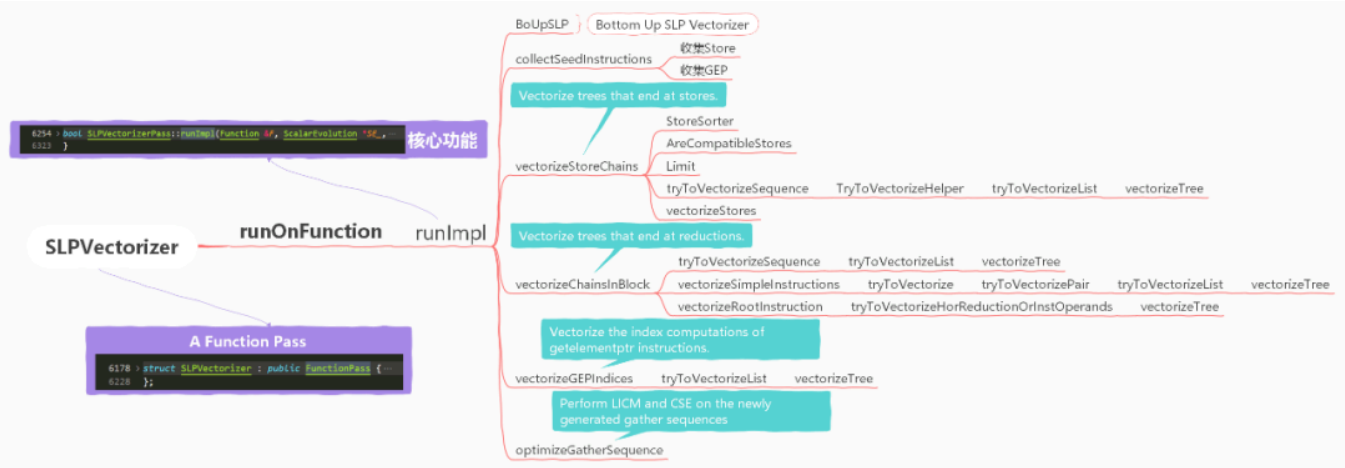
slp-min-tree-size	3	Only vectorize small trees if they are fully vectorizable
slp-max-look-ahead-depth	2	The maximum look-ahead depth for operand reordering scores
view-slp-tree		Display the SLP trees with Graphviz

(2)毕昇编译器额外提供选项

项名称	默认值	描述
slp-vectorize-load-aggressive	TRUE	Enable load vectorization aggressively (default = on)
slp-look-ahead-users-budget	2	The maximum number of users to visit while visiting the predecessors. This prevents compilation time increase.

2. 实现

该 pass 的实现较为复杂，源码有10k+，粗略结构如下：



(1)SLPVectorizer

代码行数：6178 ~ 6228

该Pass是个function pass，以function为单位进行优化，意味着用的资源也是function级别的。addRequired指的是该PASS中用到的分析结果，addPreserved指的是该pass执行后相应的analysis pass的分析结果仍然有效。

(2)runImpl()

代码行数：6254~ 6323

该Pass的核心功能在此函数中管理，用到了两个容器 Stores和GEPs，定义在头文件：

```
1. using StoreList = SmallVector<StoreInst *, 8>;
2. using StoreListMap = MapVector<Value *, StoreList>;
3. using GEPList = SmallVector<GetElementPtrInst *, 8>;
4. using GEPListMap = MapVector<Value *, GEPList>;
5. /// The store instructions in a basic block organized by base pointer.
6.   StoreListMap Stores;
7.
8. /// The getelementptr instructions in a basic block organized by base pointer.
9.   GEPListMap GEPs;
```

可以理解成两个map，以base pointer为key，instructions为 value。

开始优化前，先做两个无法SLP的判断：a. 判断架构是否有矢量化寄存器；b. 判断function attribute是否包含NoImplicitFloat，如果包含则不做。

然后先使用 bottom-up SLP 类从store开始构建从store开始的的指令链。

之后调用DT->updateDFSNumbers(); 来排序(/// updateDFSNumbers - Assign In and Out numbers to the nodes while walking dominator tree in dfs order.)

接着使用post order(后序)遍历当前function中所有BB块，在遍历中尝试去矢量化，三个场景，a. Vectorize trees that end at stores. b. Vectorize trees that end at reductions. c. vectorize the index computations of getelementptr instructions.

如果矢量化成功了，那么做收尾的调整。

```
1. /// Perform LICM and CSE on the newly generated gather sequences.
2. void optimizeGatherSequence();
```

(3)BoUpSLP

代码行数：550 ~ 2448

声明成员函数和结构类型，具体可以参考https://llvm.org/doxygen/classllvm_1_1slpvectorizer_1_1BoUpSLP.html。

(4)collectSeedInstructions

代码行数：6468 ~ 6501

遍历BB块，寻找两样东西，符合条件的store和GEP。Stores和GEPs是两个map，访问同一个基地址的操作放进同一个key的value中。

store 条件1：
```
bool isSimple() const { return !isAtomic() && !isVolatile(); }
```

store 条件2：
```
isValidElementType(SI->getValueOperand()->getType())
```

GEP条件1：
```
!(GEP->getNumIndices() > 1 || isa<Constant>(Idx))
```

GEP条件2：
```
isValidElementType(Idx->getType())
```

GEP条件3：
```
!(GEP->getType()->isVectorTy())
```

符合以上条件的store或GEP可以做为seed。

(5)vectorizeStoreChains
```
// Vectorize trees that end at stores.
```

代码行数：10423 ~ 10511

(这部分和llvm-12差异较大，引入了一个函数模板tryToVectorizeSequence)

遍历Stores，如果一个base pointer相关的指令不少于两条，就尝试矢量化，调用函数 vectorizeStores

代码行数：8442 ~ 8573

定义了两个比较器StoreSorter 和 AreCompatibleStores， 对Stores中的store进行排序(///Sort by type, base pointers and values operand)。以及limit，获取最小的VF。

以上三个辅助函数给函数 tryToVectorizeSequence 用。

(6)vectorizeChainsInBlock

```
1. // Vectorize trees that end at reductions.
2. // Ran into an instruction without users, like terminator, or function call  with ignored return value, store
```

代码行数：10089 ~ 10330

对PHI节点下手，将PHI节点作为key。

(7)vectorizeGEPIndices

```
1. // Vectorize the index computations of getelementptr instructions. This
2. // is primarily intended to catch gather-like idioms ending at
3. // non-consecutive loads.
```

代码行数：10331 ~ 10422

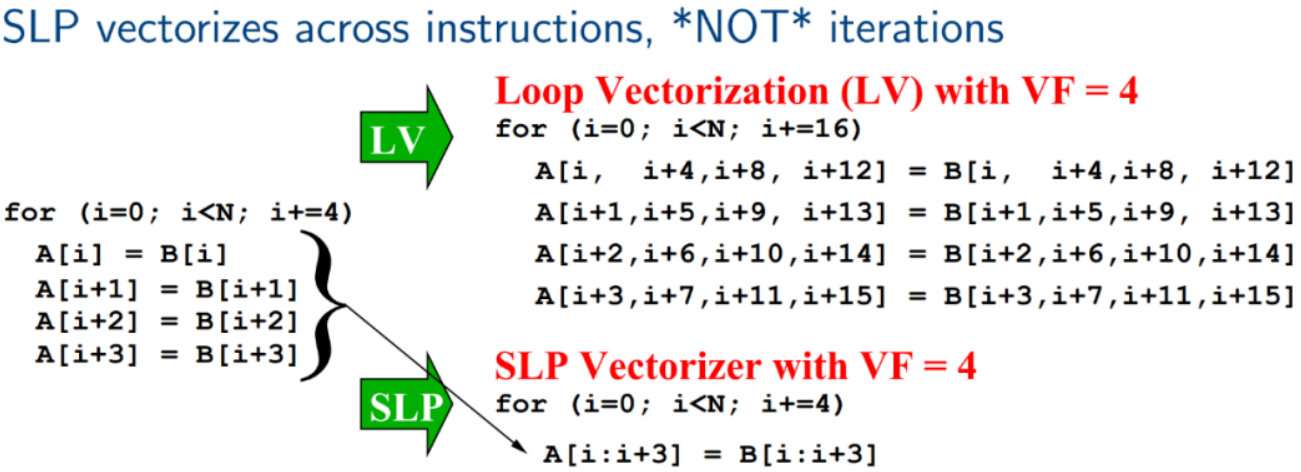
(8)vectorizeTree

以上(5)，(6)，(7)三大类矢量化场景，最终都要用到vectorizeTree函数。

总结

最后以一个例子来总结，SLP和循环矢量化的差异[6]：

SLP与LV差异[6]



本文主要带大家了解了传统SLP矢量化优化的基本思想，以及Loop-Aware SLP的使用场景，并且大致了解了llvm中SLP pass 的源码架构，对于具体实现向量化代码的构造函数以及cost model机制需要各位对SLP感兴趣的读者深入学习，同时llvm作为一个优秀的现代C++项目，其中的数据结构，编程技巧都能启发大家，受益颇多。

另外，SLP本身作为llvm中自动矢量化中的一部分，可以弥补一部分循环矢量化无法覆盖到的优化场景。社区中对于SLP的讨论也比较火热，感兴趣的读者也可以到llvm社区参与讨论https://llvm.org/。以下列举了一些近年来关于SLP的研究论文：

1. PostSLP: Cross-Region Vectorization of Fully or Partially Vectorized Code, LCPC,2019

2. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse, CGO,2019

3. goSLP: globally optimized superword level parallelism framework, SPLASH, 2018

4. Look-Ahead SLP: Auto-vectorization in the presence of commutative operations, CGO, 2018

5. VW-SLP: Auto-vectorization with adaptive vector width, PACT, 2018

6. SuperGraph-SLP Auto-Vectorization,PACT,2017

7. PSLP: padded SLP automatic vectorization, PACT, 2015

8. Throttling Automatic Vectorization: When Less is More, CGO, 2015

参考资料

[1].https://llvm.org/docs/Vectorizers.html

[2].https://groups.csail.mit.edu/cag/slp/SLP-PLDI-2000.pdf

[3].https://llvm-clang-study-notes.readthedocs.io/_/downloads/en/latest/pdf/

[4].https://gcc.gnu.org/wiki/HomePage?action=AttachFile&do=get&target=GCC2007-Proceedings.pdf

[5].https://gcc.gnu.org/legacy-ml/gcc-patches/2007-08/msg00854.html

[6].http://vporpo.me/papers/postslp_lcpc2019_slides.pdf