背景

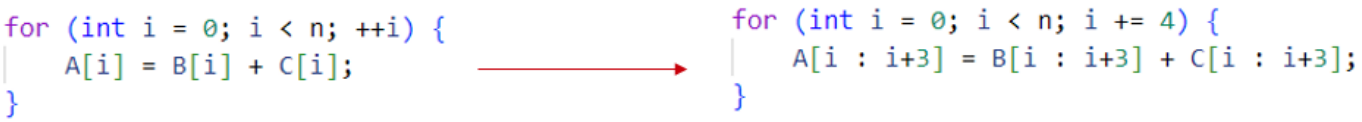
向量化是一种将程序中标量代码转换为向量代码的优化手段。当前很多芯片架构都拥有向量计算单元，架构指令本身支持单指令多数据(SIMD)的并行计算，一条指令同时计算多个数据。使用向量化优化后，可以实现一个cycle计算多个标量数据，从而带来巨大的性能提升。

在LLVM框架中有两个自动向量化pass：循环向量化(Loop Vectorizer)和 SLP向量化(Superword-level Parallelism)。其中循环向量化关注循环迭代间的向量化机会，会使循环的迭代次数减少，单次迭代的计算数据量增大。而SLP向量化关注迭代内的向量化机会，会将单次迭代中的相似标量计算/访存等操作合并为一条向量指令，也就是减少单次迭代内的指令生成数量。

这两个向量化pass都在LLVM中端opt部分执行，其中SLP向量化位置更靠后一些，在循环向量化完成且控制流图简化(CFG Simplify)后执行。

循环向量化

循环向量化扩展循环中的指令，转换的逻辑如下图所示。这个优化在LLVM中是默认开启状态，如果想要关闭可以使用 -fno-vectorize 选项。




```
#pragma clang loop vectorize_width(2)

for (…) {

}
```

1. 优化诊断

实际用户代码中，会有很多循环因为各种原因(比如：过于复杂的控制流、数据类型不支持等)无法完成向量化。LLVM提供了打印诊断信息的能力方便开发者调试，这些信息会提示一个循环向量化是否成功以及失败原因，不过失败原因可能不会非常细节。

以下方循环为例，这个循环使用 #pragma 指定要做向量化，但是循环中有个向量化不支持的switch语句，所以会向量化失败。

```
#pragma clang loop vectorize(enable)

for (int i = 0; i < 128; i++) {
```

```
        switch (A[i])

}
```

LLVM提供的的优化诊断有三种：

(1)-Rpass=loop-vectorize：指示成功被向量化的循环，对这个循环不会打印。

(2)-Rpass-missed=loop-vectorize：指示向量化失败的循环，以及循环有没有被指定向量化。对示例循环的打印信息如下图，其中 Force=true 表示这个循环被指定了向量化，而提示信息表达的是位于test.cpp文件的第10行的for循环向量化失败。

```
test.cpp:10:5: remark: loop not vectorized (Force=true) [-Rpass-missed=loop-vectorize]
    for (int i = 0; i < 128; i++) {
    ^
```

(3)-Rpass-analysis=loop-vectorize：指示导致向量化失败的语句及原因。对于示例循环的打印信息如下图。提示信息表达的是位于test.cpp第11行的 switch 语句不支持导致这个语句所在的循环向量化失败。

```
test.cpp:11:9: remark: loop not vectorized: loop contains a switch statement [-Rpass-analysis]
        switch (A[i]) {
        ^
```

2. 支持场景

循环向量化支持对多种不同的循环场景进行向量化，较为常见的为以下这些。

(1)Unknown trip count

循环的迭代次数为编译期未知变量，执行次数不可知。如下例中，循环的迭代次数n是函数的参数，在编译期是变量。这种情况下循环向量化会将循环切分为非尾块部分(整除向量宽度)和尾块部分(余数)，通过 if 分支来控制运行时的执行流程。

```
int func(int n)

{

    ...

    for (int i = 0; i < n; ++i) {

     A[i] = B[i] + C[i];

    }
```

```
    ...
}
```

(2)Reductions

归约操作，如累加操作等，将一段数据计算为一个标量数据。如下例中，对一段内存A中的所有数据进行求和运算。变形后会先将 sum 扩展为向量并循环执行向量加法 sums[4] = A[0:3] + A[4:7] + … + A[n – 3 : n]，再使用LLVM的 reduction intrinsic 对 sums[4] 进行累加。

```
unsinged sum = 0;

for (int i = 0; i < n; ++i) {

    sum += A[i];

}
```

(3)If-conversions

循环包含较为简单的if判断语句形成控制流。

```
unsigned sum = 0;

for (int i = 0; i < n; ++i) {

    if (A[i] > B[i])

        sum += A[i];

}
```

(4)Reverse Iterators

循环反向遍历，迭代顺序从后向前。

```
for (int i = n; i > 0; --i) {

```

```
    A[i] += 1;
}
```

(5)Inductions

循环的归纳变量(i)参与运算。

```
for (int i = 0; i < n; ++i) {
    A[i] += i;
}
```

(6)Scatter/Gather

非连续的分散/聚集内存参与计算。

```
for (int i = 0; i < n; ++i) {
    A[i] += B[i* 2];
}
```

(7)Function calls

循环中包含函数调用计算。

```
for (int i = 0; i < n; ++i) {
    A[i] = sqrt(B[i]);
}
```

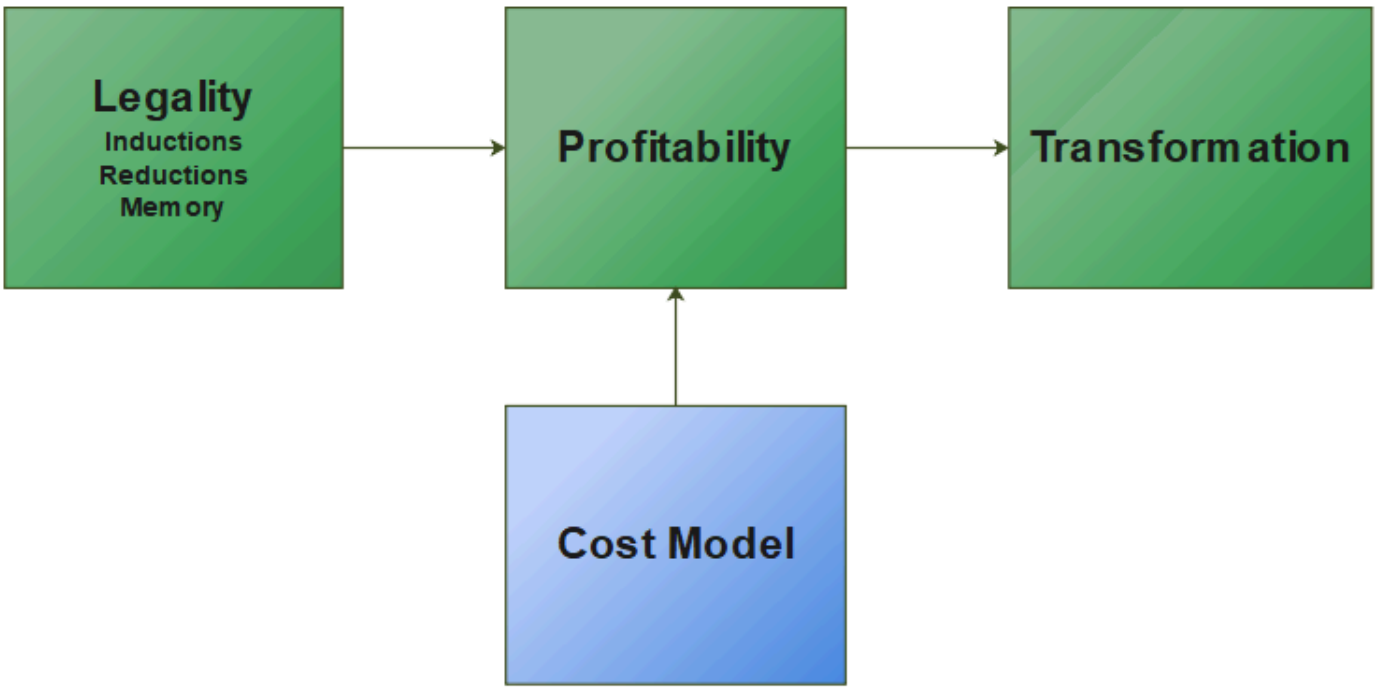
(8)Mixed Types

循环中参与计算的元素的类型和类型宽度可以不一致。

```
int func(int n, int *A, char *B)

{

    ...

    for (int i = 0; i < n; ++i) {

     A[i] = B[i] * 2;

    }

    ...

}
```

3. 循环向量化总体流程

循环向量化的流程总体分为三步：合法性分析 -> 收益分析 -> 中间表示转换。



首先是对循环做合法性检查，检查内容很多，包括指令合法和内存合法等。指令合法检查这个计算操作本身是否被向量化支持，比如在Induction和Reduction循环中，只有部分计算操作可以支持；而在函数调用计算中，只有LLVM内置函数且有向量版本的函数调用可以支持。内存合法主要是检查访存模式是

否安全无依赖，比如 A[i] = A[i - 1] + 2 这种计算，后一个迭代运算必须要等前一个迭代运算完成才能，这样就无法向量化。

接下来是使用代价模型来计算向量化收益，代价越低就代表收益越高。这一步会计算标量和所有硬件允许的vector width的代价并作出比较，决定循环是否应该做向量化优化以及计算向量宽度。

最后一步是以第二步计算出的向量宽度为切分目标，生成非尾块和尾块部分，并对最内层计算做出扩展。

```
for (int i = 0; i < n; ++i) {

    A[i] = B[i] + C[i];

}
```

切分后变为：

```
for (int i = 0; i < n; i += 4) {

    for (int j = i; j < i + 4; ++j) {

        A[j] = B[j] + C[j];

    }

}


for (int k = (n / 4) * 4; k < n; ++k) {

    A[k] = B[k] + C[k];

}
```

扩展后变为：

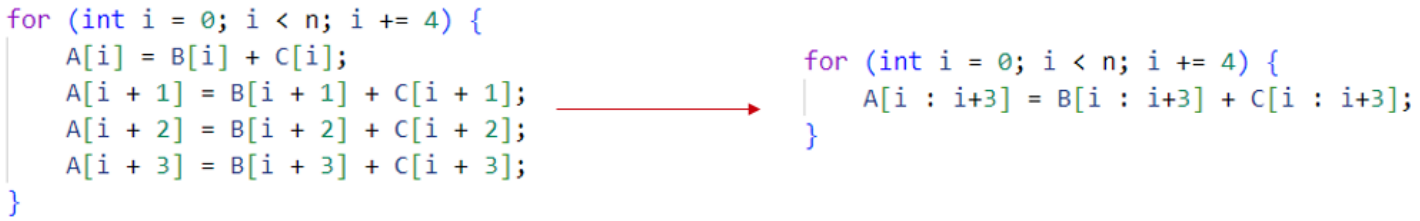
```
for (int i = 0; i < n; i += 4) {

    A [i : i+3] = B [i : i+3] + C [i : i+3];
```

```
    }


for (int k = (n / 4) * 4; k < n; ++k) {

    A[k] = B[k] + C[k];

}
```

其中第一个循环内的计算会被替换为硬件的向量计算指令。

SLP向量化

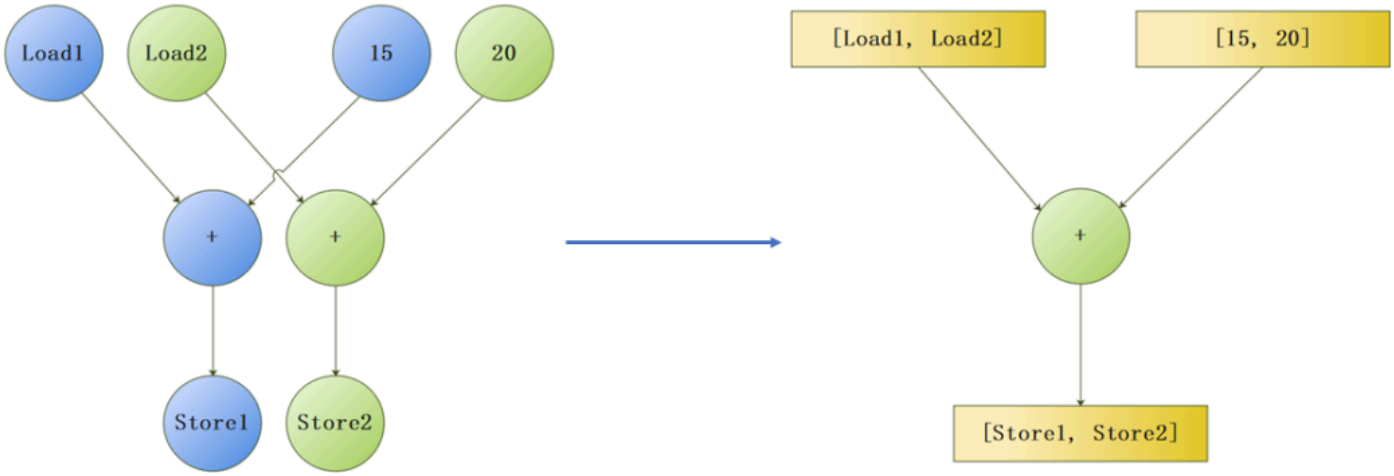
SLP向量化的目的是结合同一循环迭代内的多个计算操作，将多条标量计算合并为一条向量计算，在减少计算指令执行次数的同时，减少Code Size和寄存器压力。这个优化在LLVM中是默认开启状态，如果想要关闭可以使用 -fno-slp-vectorize 选项。SLP向量化的转换逻辑如下图，我们可以看到循环的迭代次数并没有发生变化，但是单次迭代内的四个标量运算合并成了一个向量运算。



SLP向量化的潜在目标计算需要满足两点：无数据依赖和相同操作。多条计算之间的操作数和结果不能有依赖关系，才能保证可以使用向量化并行执行。这些计算的运算符(如：加/减/乘/除)也需要相同。

SLP向量化会以目标运算的store指令为起点搜索，将运算符和操作数构建出树形结构，随后使用代价模型计算树的profit并决定合并哪些计算，最后将树形结构合并，形成向量计算。以下方计算为例，流程大致如下图所示。

```
A[i] = B[i] + 15;

A[i + 1] = B[i + 1] + 20;
```



总结

本文介绍了一种编译器中常用的优化技术——自动向量化。自动向量化技术可以在中间代码层面，针对硬件的特性或用户的指示，对代码中的循环根据特定长度进行切分并将循环中的原标量计算操作替换为硬件支持的向量计算操作，发挥硬件的向量计算单元算力。熟悉自动向量化优化及诊断可以为提升目标代码性能提供非常大的帮助。

参考文献

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

2. https://llvm.org/devmtg/2014-02/slides/golin-AutoVectorizationLLVM.pdf