编译论坛

程序编写优化 I

嘉宾：王梦园

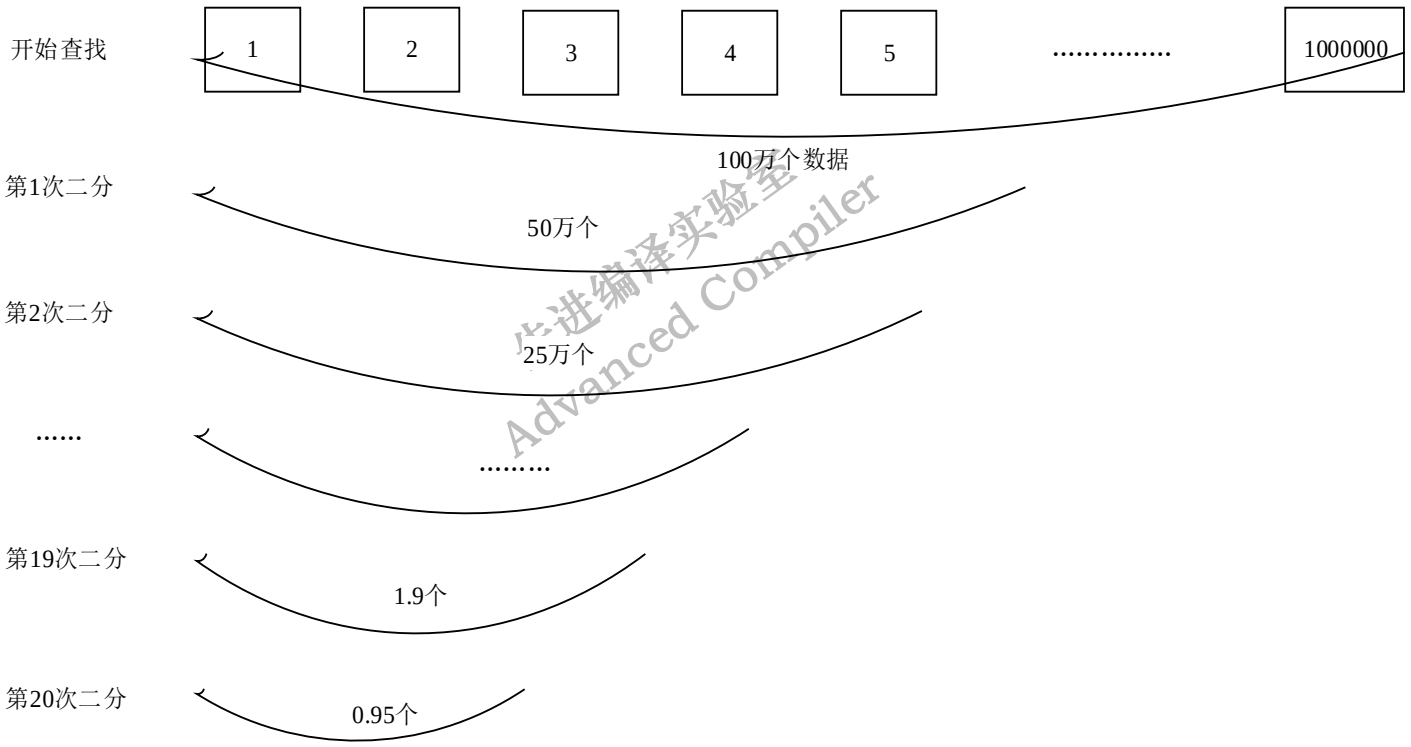
目录

6.1 算法优化

- 算法简介
- 选择适合算法
- 改进算法策略

算法简介

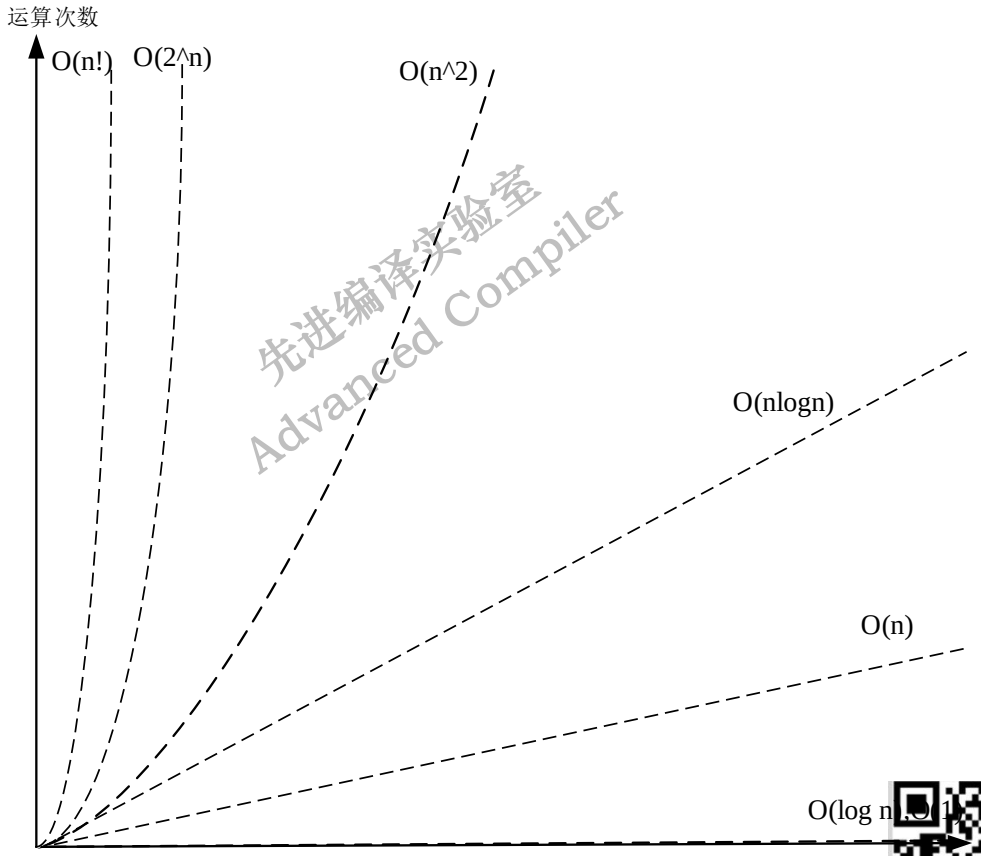
任何问题的解决都有一定的方法和步骤，算法就是计算机解决问题过程的描述。从程序设计的角度看，算法由一系列求解问题的指令构成，能根据规范的输入，在有限的时间内获得有效的输出结果，代表了用系统的方法来描述解决问题的一种策略机制。



算法简介

算法复杂度这一指标用于考量算法需要的时间和空间。算法的时间复杂度，即程序执行时间增长的变化趋势。空间复杂度是指一个程序在运行过程中临时占用存储空间大小的一个量度。

一般来说，常见的时间复杂度量级有常数阶、对数阶、线性阶、线性对数阶、平方阶、指数阶、阶乘阶等，依次按照顺序时间复杂度越来越大，执行效率也越来越低。



选择适合算法

算法优化是指通过对算法进行更好的设计以提升程序的性能，程序中的算法优化可以从选择合适的算法以及算法自身的优化这两方面进行考虑，本节结合常用的排序算法进行说明。

常用的排序算法有十余种，分为基于比较的插入、选择、交换这三类，如冒泡排序、快速排序等。下面选择三类排序算法中的几个典型算法及优化思路进行介绍，分析每种算法的复杂度及其性能的差异，阐述不同应用场景下选择合适算法对程序性能的影响：

选择适合算法

冒泡排序。冒泡排序是一种简单的交换排序算法。通过元素的两两比较，判断是否符合排序要求，如果不符合就交换位置来达到排序的目的。对于n个需要排序的数据来说，其算法复杂度为$O(n^2)$。其每次排序之后仍会继续进行下一轮的比较，无效计算较多，因此只适用于数据量较小的场景。


```
void bubble_sort(int a[], int n)
{
    int i, j, temp;
    for ( j = 0; j < n - 1; j++){//总共需要冒泡多少次
        for ( i = 0; i < n - 1 - j; i++){
            if (a[i] > a[i + 1]){
                temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;//交换操作
            }
        }
    }
}
```

选择适合算法

直接插入排序。插入排序是通过把序列中的值插入一个有序序列对应的位置上，直到该序列结束。插入排序的赋值操作次数是比较操作次数减去(n-1)次，平均来说插入排序算法复杂度为$O(n^2)$，但在部分数据有序的情况下其效率比冒泡排序更快。

```
void insertSort(int *arr, size_t size){
    assert(arr);
        for (int idx = 1; idx <= size - 1; idx++){//idx表示插入次数，共进行n-1次插入
            int end = idx;
            int temp = arr[end];
            while (end > 0 && temp < arr[end - 1]){//将比当前元素大的元素都往后移动一个位置
                arr[end] = arr[end - 1];
                end--;
            }
                arr[end] = temp; //元素后移后要插入的位置就空出了，找到该位置插入
    }
```

选择适合算法

简单选择排序。这种排序方式每次从待处理数据中选出最小的，放在已经排好序的序列末尾，直至所有排序结束。假设有n个待排序的数据，则比较的总次数为n*(n-1)/2，时间复杂度为$O(n^2)$。可以看出选择排序的对比次数较多但数据移动次数较少，在数据量大的情况下其效率明显优于冒泡排序，适用于大多数排序场景。

```
void selectSort(int a[], int len){
    int i, j, temp;
    int minIndex = 0;
        for (i = 0; i < len - 1; i++){
            minIndex = i;
                for (j = i + 1; j < len; j++){
                    if (a[j] < a[minIndex]){
                        minIndex = j;//
                    }
                }
                if (minIndex != i) {
                    temp = a[i];
                    a[i] = a[minIndex];
                    a[minIndex] = temp;
                }
    }
}
```

改进算法策略

通过前文的描述已经说明了算法的性能表现会对程序性能产生较大影响，所以在保证正确性的前提下通过改进算法策略提升程序性能是十分有必要的。改进算法策略的方法归结起来可以分为以下两点：

①从算法过程出发，尽量减少算法复杂度，提高其运行的效率。

②从算法编码出发，运用一些技巧优化算法中的编码方式，从编码角度提升算法性能。

改进算法策略

假设输入向量包含下面7个元素，分别为(8，-33，16，9，-12，45，67)，则该向量的连续子向量的最大和为x[3-7]的总和125，此算法实现及改进思路如下：

枚举算法实现。此向量中所有子向量数都是正数时，显然子向量最大和就是整个输入向量的和。当输入向量中含有负数时，遍历所有数组下标范围[i,j]的子向量求和，通过比较计算出最大和。

```
for (int i = 0; i < n; i++){
    for (int j = 0; j <= i; j++){
        sum = 0;
        for (int k = j; k <= i; k++){
            sum += x[k];
        }
        ans = max(ans, sum);
    }
}
```

改进算法策略

枚举算法优化。上述枚举算法示例中存在部分重复运算，此时可以利用一个变量保存之前的运算结果避免部分冗余计算。经过改进后算法的复杂度为$O(n^2)$，改进后部分代码如下。


```
for (int i = 0; i < n; i++){
    sum = 0;
        for (int j = i; j < n; j++){
            sum += x[j];
            maxsum = max(maxsum, sum);
        }
}
```

改进算法策略

分治算法。可以采用分治技术将问题分解成2个与原问题形式相同的子问题分别递归求解。把向量序列从中间分为左右A和B两部分，把A作为新的输入序列求出左半部分的最大子向量和A1，同理求出右半部分的最大子向量和B1，将跨越左右部分的最大向量和称为C1。最后取这三个子向量和中的最大值即可。该算法的时间复杂度为$O(nlogn)$，整体性能高于前两种算法。

```
for (int i = mid; i >= 0; i--){
    t += x[i];
    left = max1(left, t);
}
int right = x[mid + 1];
t = 0;
for (int i = mid + 1; i <= r; i++){
    t += x[i];
    right = max1(right, t);
}
maxsum = max1(maxsum, left + right);
return maxsum;
```

改进算法策略

线性算法。此问题还可以从头到尾扫描数组的思路求解。假设往一个长度为i的向量后面插入第i+1个数，此时只需要从包含第i+1个数和不包含第i+1个数的序列中选出最大的子向量和，而后者是已知的。因此x[i]作为末尾元素时能找到的最大子向量和和要么是x[i]本身，要么是x[i-1]作为末尾元素时能找到的最大子向量和再拼接上x[i]。该代码时间复杂度为$O(n)$。

```
for (int i = 1; i < 8; i++) {
    f[i] = max(f[i - 1] + x[i], x[i]);
    maxsofar = max(f[i], maxsofar);
}
    return  maxsofar;
```

参考资料

AI框架发展白皮书（2022年）

https://syncedreview.com/2020/12/14/a-brief-history-of-deep-learning-frameworks/

先进编译实验室
王磊
2022年09月

AdvancedCompiler

Tel: 13839830713

编译论坛

程序编写优化 II

嘉宾：王梦园

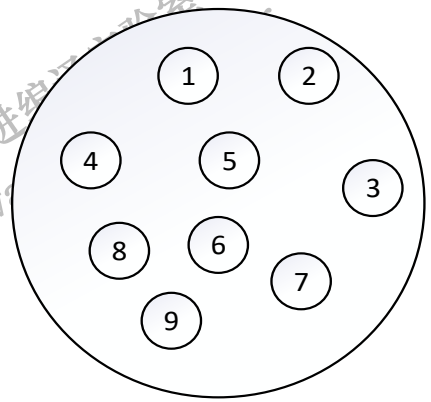
目录

6.2 数据结构优化

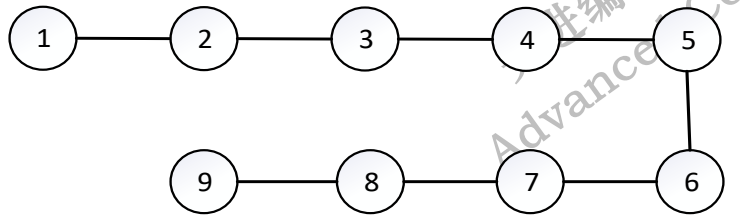
- 典型数据结构的性能分析
- 选择适合的数据类型
- 选择适合的数据结构

典型数据结构的性能分析

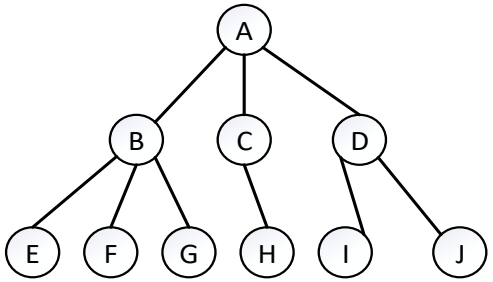
按照分类标准的不同，数据结构可以分为逻辑结构和存储结构，数据的逻辑结构是指数据对象中的数据元素之间的相互关系，与数据的存储尚未关联，是从具体问题抽象出来的数学模型，主要分为集合结构、线性结构、树形结构、图形结构等。



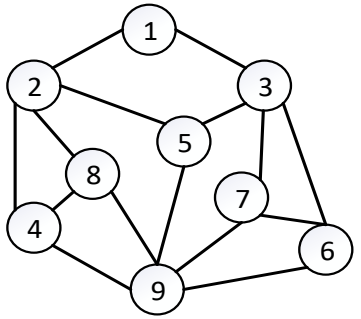
(a)集合结构



(b)线性结构



(c)树形结构



(d)图形结构

典型数据结构的性能分析

常用的数据存储结构有数组、栈、队列、链表、树、哈希表、堆、图等，这些数据结构有各自的优缺点，适用的场景也各不相同，这些数据结构的优缺点如下表。

数据结构	优点	缺点
数组	插入快	查找慢、删除慢、大小固定
有序数组	查找快	插入慢、删除慢、大小固定
栈	后进先出	存取其它项慢
队列	先进先出	存取其它项慢
链表	插入、删除快	查找慢
二叉树	查找、插入、删除快	算法复杂
哈希表	存取快、插入快、删除快	无关键字存取慢、存储空间使用率低
堆	插入快、删除快、对大数据项存储快	对其它数据项存取慢
图	依据现实世界建模	算法复杂

典型数据结构的性能分析

执行效率较快的数据结构复杂程度一般较高，但并不是使用最快的结构就是最好的方案，仍需要根据实际情况进行考虑。

数据结构	查找	插入	删除	遍历
数组	$O(\mathrm{N})$	$O(1)$	$O(\mathrm{N})$	----
有序数组	$O(nlogn)$	$O(\mathrm{N})$	$O(\mathrm{N})$	$O(\mathrm{N})$
链表	$O(\mathrm{N})$	$O(1)$	$O(\mathrm{N})$	----
有序链表	$O(\mathrm{N})$	$O(\mathrm{N})$	$O(\mathrm{N})$	$O(\mathrm{N})$
二叉树（一般情况）	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(\mathrm{N})$
二叉树（最坏情况）	$O(\mathrm{N})$	$O(\mathrm{N})$	$O(\mathrm{N})$	$O(\mathrm{N})$
平衡树	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(\mathrm{N})$
哈希表	$O(1)$	$O(1)$	$O(1)$	----

典型数据结构的性能分析

为进一步说明选择不同数据结构对于完成相同的操作会导致程序性能的差异，选用数组、链表结构以查找数据为例进行测试。

使用有序数组数据结构编写查找程序，程序编写完成后，利用gettimeofday这个计时函数对有序数组查找耗时进行测试，数组个数为10^7，结果显示其查找完成花费了0.048ms的时间。

```
int Bin_Search(int* num, int cnt, int target){
int first = 0, last = cnt - 1, mid;
int counter = 0;
   while (first <= last) {
      counter++;
      mid = (first + last) / 2;//确定中间元素
      if (num[mid] > target) {
         last = mid - 1;//mid已经交换过了，last往前移一位
      }
      else if (num[mid] < target) {
         first = mid + 1;//mid已经交换过了，first往后移一位
      }
      else { //判断是否相等
         return 1;
      }
   }
   return 0;
}
```

典型数据结构的性能分析

选取有序链表作为数据存储结构编写程序进行目标数据的查找。程序运行后利用计时函数测试耗时情况如下，完成查找10^7个数据所花费时间为19ms，程序性能不如使用有序数数组数据结构。

```
int list_search(struct Node* list,int value) {
    struct Node* p;
        for (p = list->next; p; p = p->next) {
            if (p->value == value) {
                return 1;
            }
        }
    return 0;
}
void list_visit(struct Node* list) {
    for (struct Node* p = list->next; p; p = p->next) {
        printf("%d ", p->value);
    }
}
```

典型数据结构的性能分析

在满足精度要求的情况下，不同的数据类型也会对程序的性能有影响，具体包括两个方面，一是选择存储空间更小的数据类型，二是选择更适合硬件结构的数据类型。

通常小尺寸类型数据的访问速度比大尺寸类型数据快，且小尺寸的数据可以在缓存中放更多的数据。为验证上述结论，使用SSE指令对两种数据类型进行加法运算，并测试运行时间进行对比。

```
for(int i=0;i<N;i+=4){
    x = _mm_loadu_si128((__m128i*)&op3[i]);
    y = _mm_loadu_si128((__m128i*)&op4[i]);
    z = _mm_add_epi32(x, y);
    _mm_store_si128((__m128i*)&result2[i], z);
}
```

程序测试结果显示在处理单个数据时，SSE指令处理短整型数据的速度比处理整型数据快2倍左右。

典型数据结构的性能分析

不同的数据类型的程序在相同架构上的性能也有所不同。此时在满足正确性以及计算精度的要求下，可以对数据类型进行转换，帮助发挥硬件平台特性，提升程序的运算性能。

当使用256*256大小的矩阵规模进行测试时，单精度矩阵乘测试结果为0.036s，双精度为0.068s，加速比为1.88倍。

```
for (int k = n - 4; k >= 0; k -= 4) {
    t1 = _mm_loadu_ps(a[i] + k);
    t2 = _mm_loadu_ps(b[j] + k);
    t1 = _mm_mul_ps(t1, t2);
   sum = _mm_add_ps(sum, t1);
}
   sum = _mm_hadd_ps(sum, sum);
   sum = _mm_hadd_ps(sum, sum);
    _mm_store_ss(c[i] + j, sum);
     for (int k = (n % 4) - 1; k >= 0; --k){
        c[i][j] += a[i][k] * b[j][k];
     }
```

选择适合的数据结构

在解决具体问题时选择合适的数据结构是非常重要的，以稀疏矩阵向量乘法为例进一步说明数据结构选择的重要性。稀疏矩阵向量乘是科学工程计算的核心算法，为了提升稀疏矩阵向量乘法的性能，将采用坐标存储、行压缩、对角存储以及埃尔帕克存储四种稀疏矩阵存储格式实现稀疏矩阵向量乘法并进行测试对比。

选择适合的数据结构

坐标存储。坐标存储格式也称为三元组存储格式，其分别存储每个非零元素的行索引row、列索引col以及数值data。这种存储方式的主要优点是灵活、简单、易于按行和按列访问稀疏矩阵。

$$A=\begin{bmatrix}1 & 5 & 0 & 0\\ 0 & 2 & 6 & 0\\ 8 & 0 & 3 & 7\\ 0 & 9 & 0 & 4\end{bmatrix}$$

row=[0,0,1,1,2,2,2,3,3]

col=[0,1,1,2,0,2,3,1,3]

data=[1,5,2,6,8,3,7,9,4]

选择适合的数据结构

当稀疏矩阵存储格式为坐标存储格式时，向量乘法部分代码实现如下。

```
void spmv(int row[NNZ], int col[NNZ],DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE]) {//坐标存储的矩阵向量乘
        int i;
        for (i = 0; i < SIZE; i++) {
                y[i] = 0;
        }
        for ( i = 0; i < NNZ; i++)
                y[row[i]] += values[i] * x[col[i]];
}
```

选择适合的数据结构

行压缩存储。主要思想是逐行对稀疏矩阵进行压缩，并且记录每行首个非零元素的位置信息。行压缩存储格式由三个数组构成，数值data按行存放非零元素，列号col记录对应于data中每个非零元素位于的列数，指针ptr记录每行第一个非零元素在data中的存储位置。

$$A=\begin{bmatrix}1 & 5 & 0 & 0\\ 0 & 2 & 6 & 0\\ 8 & 0 & 3 & 7\\ 0 & 9 & 0 & 4\end{bmatrix}$$

ptr=[0,2,4,7]

col=[0,1,1,2,0,2,3,1,3]

data=[1,5,2,6,8,3,7,9,4]

选择适合的数据结构

当稀疏矩阵存储格式为行压缩存储时，向量乘法实现部分代码如下。

```
void spmv(int ptr[NUM_ROWS + 1], int col[NNZ], DTYPE data[NNZ], DTYPE y[SIZE], DTYPE x[SIZE]) {//行压缩存储的矩阵向量乘
    for (int i = 0; i < NUM_ROWS; i++) {
      DTYPE y0 = 0;
      for (int k = ptr[i]; k < ptr[i + 1]; k++) {
          y0 += data[k] * x[col[k]];
      }
      y[i] = y0;
    }
}
```

选择适合的数据结构

对角存储。对角存储格式专为由多条非零对角线组成的稀疏矩阵设计，对角存储格式采用一个二维矩阵和一个向量来存储原矩阵，其中二维矩阵中的每一列用来存储原矩阵中的一条对角线，而向量则用来存储各列对应原矩阵中主对角线的偏移量。

$$A=\begin{bmatrix}1 & 5 & 0 & 0\\ 0 & 2 & 6 & 0\\ 8 & 0 & 3 & 7\\ 0 & 9 & 0 & 4\end{bmatrix}$$

offsets=[-2,0,1]

$$data=\begin{bmatrix}* & 1 & 5\\ * & 2 & 6\\ 8 & 3 & 7\\ 9 & 4 & *\end{bmatrix}$$

选择适合的数据结构

当稀疏矩阵存储格式为对角存储时，向量乘法实现部分代码如下。

```
void spmv(DTYPE data[12], int offsets[SIZE - 1], DTYPE y[SIZE], DTYPE x[SIZE]) {//对角存储的矩阵向量乘
        int i, j, k, N;
        int Istart, Jstart, stride = 4;
        for (i = 0; i < SIZE - 1; i++) {
                k = offsets[i];
                Istart = max(0, -k);
                Jstart = max(0, k);
                N = min(SIZE - Istart, SIZE - Jstart);
                for (j = 0; j < N; j++) {
                        if (data[Istart + i * stride + j] != X)////其中X对应源数组的*表示不存在该数
                        y[Istart + j] += data[Istart + i * stride + j] * x[Jstart + j];}
        }
}
```

选择适合的数据结构

ELLPACK存储。对于一个m*n的矩阵，每行最多有k个非零值元素，ELLPACK格式将非零值存储于一个m*k的稠密矩阵Data中。相应的列指针被存储在指数矩阵 Indices中，然后用0或者其它的哨兵值来填补空缺。与对角存储格式一样，indices和data矩阵都是按列顺序来存储的，当稀疏矩阵存储格式为ELLPACK时，向量乘法实现如下。

$$A=\begin{bmatrix}1 & 5 & 0 & 0\\ 0 & 2 & 6 & 0\\ 8 & 0 & 3 & 7\\ 0 & 9 & 0 & 4\end{bmatrix}$$

$$\text{indices}=\begin{bmatrix}0 & 1 & *\\ 1 & 2 & *\\ 0 & 2 & 3\\ 1 & 3 & *\end{bmatrix}$$

$$\text{data}=\begin{bmatrix}1 & 5 & *\\ 2 & 6 & *\\ 8 & 3 & 7\\ 9 & 4 & *\end{bmatrix}$$

选择适合的数据结构

当稀疏矩阵存储格式为ELLPACK时，向量乘法实现如下。

```
void spmv(DTYPE data[12], int indices[12], DTYPE y[SIZE], DTYPE x[SIZE]) {//ELLPACK存储的矩阵向量乘
        int n, i, k, N;
        int max_ncols = SIZE - 1, num_rows = SIZE;
        for (n = 0; n < max_ncols; n++)
                for (i = 0; i < num_rows; i++)
                        if (data[n * num_rows + i] != X)
                                y[i] += data[n * num_rows + i] * x[indices[n * num_rows + i]];
}
```

选择适合的数据结构

1. 当采用对角存储格式时，其性能往往好于其它格式。缺点是采用对角格式，非零元素的坐标需要通过计算才能得到，因此通常在稀疏矩阵向量乘法计算中不会采用对角格式。

2. 只有非零元所占比例大于某一阈值时，使用ELLPACK存储格式会取得更好的性能。

3. 对角存储和ELLPACK这两种格式都需要对矩阵进行补零操作，对稀疏矩阵向量乘程序性能有一定影响，而坐标存储和行压缩存储格式则不存在这个问题，它们只存储矩阵中的非零元素，不会引入不必要的开销。

4. 每种稀疏矩阵存储结构的不同将导致在进行优化时必须选择不同的方法，因此需针对每一种方法选择不同的优化策略，以获得最优的性能。

参考资料

AI框架发展白皮书（2022年）

https://syncedreview.com/2020/12/14/a-brief-history-of-deep-learning-frameworks/

先进编译实验室
王磊
2022年09月

AdvancedCompiler

Tel: 13839830713

编译论坛

程序编写优化 III

嘉宾：王梦园

目录

6.3 过程级优化

- 别名消除
- 常数传播
- 传参优化
- 内联替换
- 过程克隆
- 全局变量优化

别名消除

C语言中为了方便编码为变量定义了别名，但在同一程序中两个以上的指针引用相同的存储位置时，将存在指针别名的问题。

```
void add(int* a, int* b) {
    int C = 5;
    for (int i = 0; i < N; i++)
        a[i] = b[i - 1] + C;
}
int main() {
    int a[N], b[N];
    int i;
    for (i = 0; i < N; i++) {
        a[i] = i;
        b[i] = i + 1;
    }
    add(a, b);
    printf("%d\n", a[1]);
}
```

```
void add(int* restrict a, int* restrict b) {//编译器可以做优化
    int C = 5;
    for (int i = 0; i < N; i++)
        a[i] = b[i - 1] + C;
}
int main() {
    int a[N], b[N];
    int i;
    for (i = 0; i < N; i++) {
        a[i] = i;
        b[i] = i + 1;
    }
    add(a, b);
    printf("%d\n", a[1]);
}
```

常数传播

常数传播是指替代表示式中已知常数的过程，一般在编译前期进行。实际程序中可能存在复杂的控制流，编译器把所有情况的常数替换都识别出来并对程序实施正确的常数替换优化是较为困难的，因此建议优化人员尽量手动进行常数传播优化。

```
int main() {
    int a = 16;
    int i;
    const int n = 256;
    int x[n];
    for (i = 0; i < n; i++) {
        x[i] = a / 4 + i;//优化前
    }
    return 0;
}
```

```
#include <stdio.h>
int main() {
    int a = 16;
    int i;
    const int n = 256;
    int x[n];
    for (i = 0; i < n; i++) {
        x[i] = 4 + i;//优化后
    }
    return 0;
}
```

内联替换

为了节省函数调用的时空开销，可以采用内联替换的思路优化程序，具体优化思路为函数在被调用处复制函数代码副本，并通过代码膨胀将被调函数体副本直接在调用处进行内联替换，同时被调过程内的形参也将被替换为主调过程内的实参。

```
void func1(int* x, int k) {
    x[k] = x[k] + k;
}
int main() {
    int i;
    const int n = 256;
    int a[n];
    for (i = 0; i < n; i++)
        a[i] = i;
    for (i = 0; i < n; i++)
        func1(&a[0], i);
    printf("%d", a[5]);
}
```

```
int main() {
    int i;
    const int n = 256;
    int a[n];
    for (i = 0; i < n; i++)
        a[i] = a[i] + i;//func1内联替换
}
```

过程克隆

过程克隆是指当一个过程在不同的调用环境下表现出不同的特性时根据需要生成该过程的多个实现，便于后续针对每个实现进行不同的优化处理，程序中的调用点会根据其上下文的属性信息来选择调用过程实现的某个版本。

```
void func(int *A,int j){
    int k = 1;
    for (int i = 0; i < N-j; i++) {
        A[i + j] = A[i] + k;
    }
}
int main() {
    int A[N] = {0}, i, j;
    j = rand() % 10;
    func(A,j);
}
```

```
void func1(int *A,int j){
    int k = 1;
    for (int i = 0; i < N-j; i++) {
        A[i + j] = A[i] + k;
    }
}
void func2(int *A,int j){
    int k = 1;
    for (int i = 0; i < N; i++) {
        A[i+j] = A[i] + k;//后续可以进行向量化优化
    }
}
int main() {
    int A[N] = {0}, i, j;
    j = rand() % 10;
    if(j=0||j>4)
        func2(A,j);
    else
        func1(A,j);
}
```

全局变量优化

全局变量尤其是多个文件共享的全局数据结构会阻碍编译器的优化。 并且其使得程序员不便追踪其变化，难以进行手工优化。对于并行程序来说，全局变量除非在迫不得已的情况下才建议使用，就算要使用全局变量，也尽量通过参数传递的方式。

```
#include <stdio.h>
int a = 1;
void func() {
        int c = 14;
        a = a + c;
}
int main() {
        func();
        printf("%d", a);
}
```

```
#include <stdio.h>
int a = 1;
void func(int *a) {
        int c = 14;
        *a = *a + c;
}
int main() {
        func(&a);
        printf("%d", a);
}
```

参考资料

AI框架发展白皮书（2022年）

https://syncedreview.com/2020/12/14/a-brief-history-of-deep-learning-frameworks/

AdvancedCompiler

Tel: 13839830713

编译论坛

程序编写优化 IV

嘉宾：王梦园

目录

6.4 循环级优化

循环不变量外提

循环不变量是指在循环迭代空间内值不发生变化的变量。由于循环不变量的值在循环的迭代空间内不发生变化，因此可将其外提到循环外仅计算一次，避免其在循环体内重复计算。示例如下，经过循环不变量外提后，上述循环的计算强度得到了削弱，提高了代码的性能。


```
for (int i = 1; i < N; i++)
    for (int j = 1; j < M; j++)
        U[i] = U[i] + W[i]*W[i]* D[j] / (dt * dt);
```

```
T1 = 1 / (dt * dt);
  for (int i = 1; i < N; i++) {
     T2 = W[i]*W[i];
     for (int j = 1; j < M; j++)
        U[i] = U[i] + T2 * D[j] * T1;
  }
```

循环展开

循环展开是一种常用的提高程序性能方法，它通过将循环体内的代码复制多次的操作，进而减少循环分支指令执行的次数，增大处理器指令调度的空间，获得更多的指令级并行。


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

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j += 4) {
        A[i][j] = A[i][j] + B[i][j] * C[i][j];
        A[i][j + 1] = A[i][j + 1] + B[i][j + 1] * C[i][j + 1];
        A[i][j + 2] = A[i][j + 2] + B[i][j + 2] * C[i][j + 2];
        A[i][j + 3] = A[i][j + 3] + B[i][j + 3] * C[i][j + 3];
    }
```

循环合并

循环合并是指将具有相同迭代空间的两个循环合成一个循环的过程，其属于语句层次的循环变换。但并不是所有循环都可以进行合并，循环合并需要满足合法性要求，有些情况下循环合并会导致结果错误。

for (i = 1; i < N; i++) *A[i] = B[i] + C;//S1 语句* *for (i = 1; i < N; i++)* *D[i] = A[i + 1] + E;//S2 语句*	*for (i = 1; i < N; i++) {* *A[i] = B[i] + C;//S1 语句* *D[i] = A[i + 1] + E;//S2 语句* *}*

循环合并

满足合法性要求的循环合并可以减小循环的迭代开销以及并行化的启动和通信开销，还可能增强寄存器的重用。

```
for (i = 0; i < N; i++)
    x[i] = a[i] + b[i];
for (i = 0; i < N; i++)
    y[i] = a[i] - b[i];
```

```
for (i = 0; i < N; i++) {
    x[i] = a[i] + b[i];
    y[i] = a[i] - b[i];
```

循环分段

循环分段可将单层循环变换为多层嵌套循环，循环分段的段长可根据需要选取。如果原循环是可并行化的循环，则分段后依然可以实施并行化变换。通常采用循环分段技术实现外层的并行化以及内层的向量化，以达到利用系统多层次并行资源的目的。

```
for (i = 0; i < N; i++)
    A[i] = B[i] + C[i];
```

```
int K = 32;
  for (i = 0; i < N; i += K) {
     for (j = i; j < i + K -1; j += 4) {
        ymm0 = _mm_load_ps(B + j);
        ymm1 = _mm_load_ps(C + j);
        ymm2 = _mm_add_ps(ymm0, ymm1);
        _mm_storeu_ps(A + j, ymm2);
     }
  }
```

循环交换

循环交换是一个重排序变换，在程序的向量化和并行化识别以及增强数据局部性方面都起着重要的作用。

```
for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
        for (i = 0; i < N; i++)
            A[i][j] = A[i][j] + B[i][k] * C[k][j];
```

```
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
            A[i][j] = A[i][j] + B[i][k] * C[k][j];
```

循环交换在某些情况下也能提高寄存器的重用能力。为了提高寄存器重用能力的循环交换的目的在于把携带依赖的循环放在最内层的位置，使可以被重用的值保留在寄存器中。

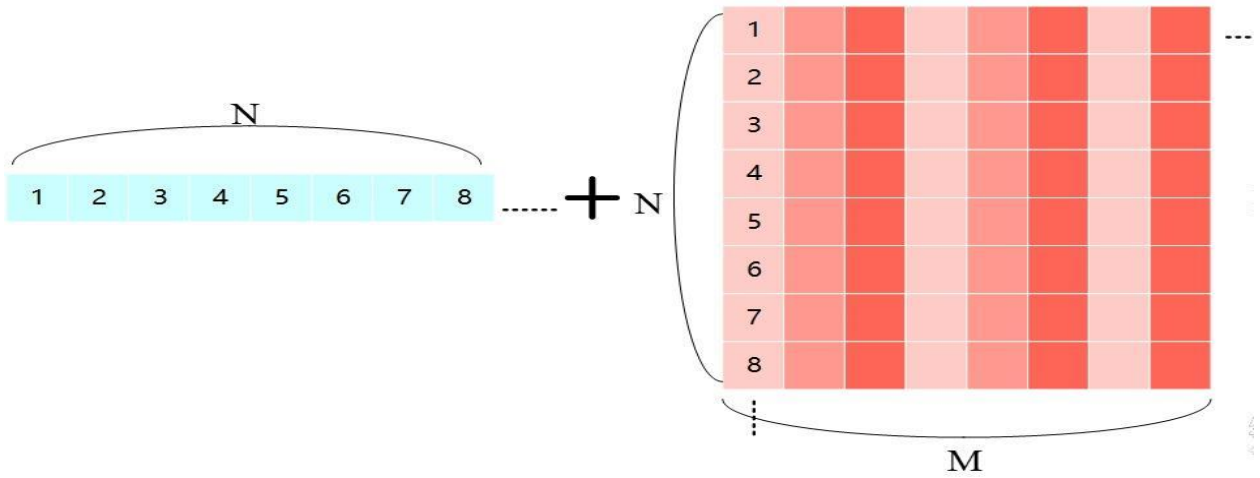

```
for (i = 1; j < M; j++)
    for (j = 1; i < N; i++)
        A[i][j] = A[i - 1][j];
```

```
for (j = 1; j < N; j++)
    for (i = 1; i < M; i++)
        A[i][j] = A[i - 1][j];
```

循环分块

循环分块是指通过增加循环嵌套的维度来提升数据局部性的循环变换技术，是对多重循环的迭代空间进行重新划分的过程，循环分块前后要保证迭代空间相同。循环分块是循环交换和循环分段的结合。可以提高程序的局部性，增加数据重用来提升程序的性能。

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



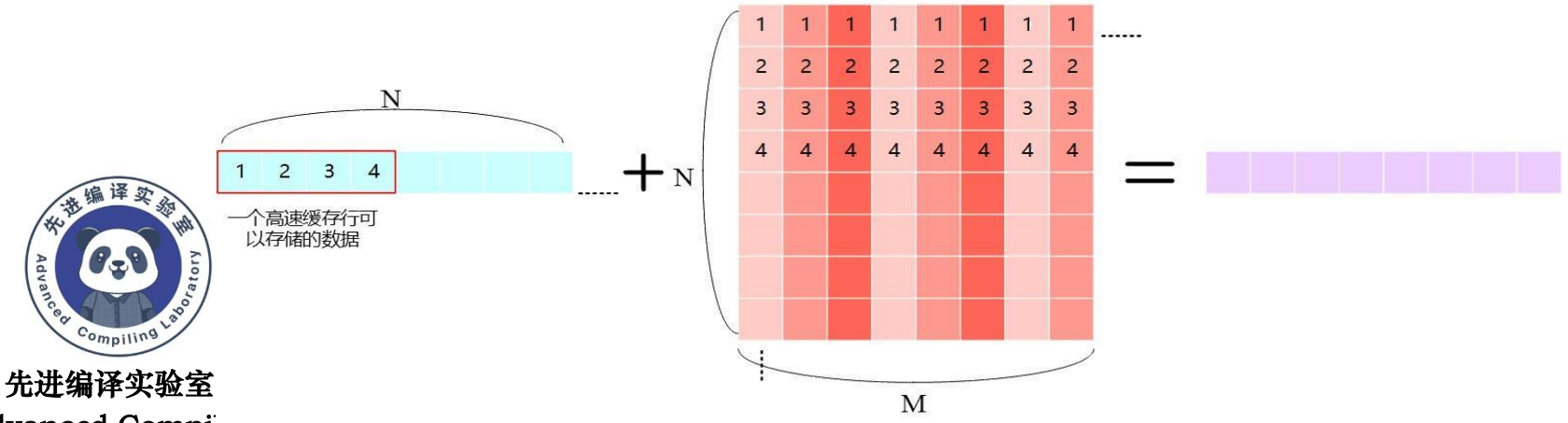
循环分块

C语言访存数据是行优先的原则，设置S长度与硬件平台高速缓存行相匹配，将控制这些分段上进行迭代的循环移动最外层的位置，这样单次迭代数据量得到了大幅度的减少，减少了从主存取数的时间，进而提升了性能。

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

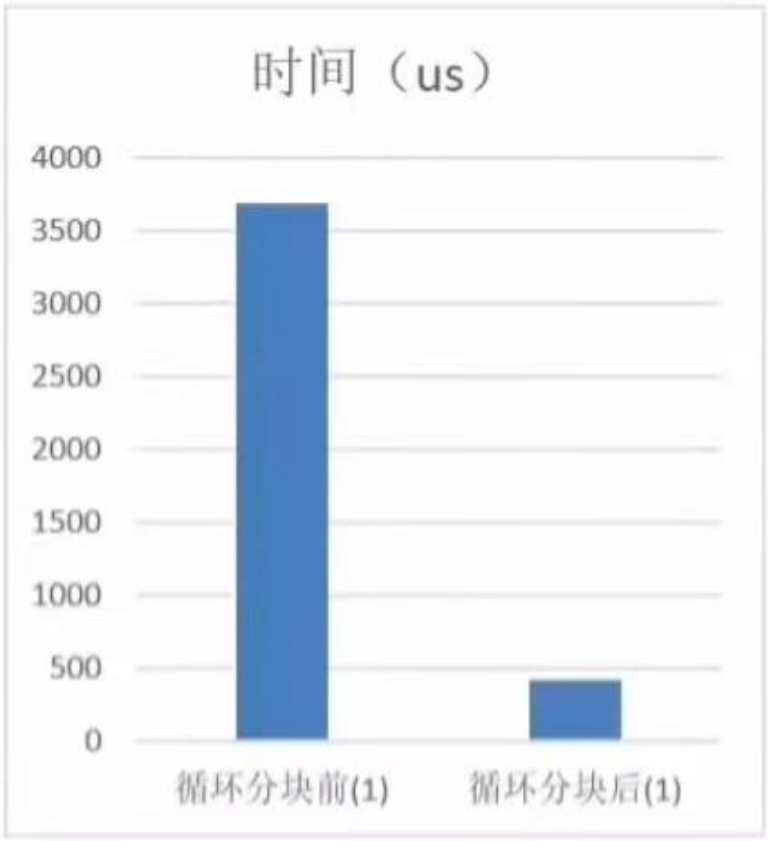
循环分块

```
S = 4;
for (i = 0; i < N; i+=S){
  for (j = 0; j < M; j++){
    for (I = i;I < MIN(i+S,N);I++){
      B[I] = B[I] + A[I][j];
    }
  }
}
```



循环分块

对循环分块前后优化效果进行测试，编译器版本为llvm-13，示例优化前执行时间为3600us，优化后执行时间为397us，可以看出有将近十倍的加速效果。



循环分布

循环分布将一个循环分解为多个循环，每个循环都有与原循环相同的迭代空间，但只包含原循环的语句子集。通过循环分布可以a减少指令缓存的压力，还能增加寄存器的重用，常用于分解出可向量化或可并行化的循环，进而将可向量化部分的代码转为向量执行。

```
for (i = 1; i < N; i++) {
    A[i + 1] = A[i] + C;//S1语句
    B[i] = B[i] + D;//S2语句
}
```

```
for (i = 1; i < N; i++)
    A[i + 1] = A[i] + C;
    ymm0 = _mm_set_ps(D, D, D, D);
    for (i = 0; i < N / 4; i++) {
        ymm1 = _mm_load_ps(B + 4 * i);
        ymm2 = _mm_add_ps(ymm0, ymm1);
        _mm_storeu_ps(B + 4 * i, ymm2);
    }
```

循环分裂

循环分裂是对循环的迭代次数进行拆分，将循环的迭代次数拆成两段或者多段，但是拆分后的循环不存在主体循环之说，也就是拆分成迭代次数都比较多的两个或者多个循环。

for (i = 1; i < N; i++) * Vec[i] = Vec[i] + Vec[M];*	*for (i = 1; i < M; i++)* * Vec[i] = Vec[i] + Vec[M];* *for (i = M + 1; i < N; i++)* * Vec[i] = Vec[i] + Vec[M];*

参考资料

AI框架发展白皮书（2022年）

https://syncedreview.com/2020/12/14/a-brief-history-of-deep-learning-frameworks/

AdvancedCompiler

Tel: 13839830713

编译论坛

程序编写优化 V

嘉宾：王梦园

目录

6.5 语句级优化

- 删除冗余语句
- 代数变换
- 去除相关性
- 公共子表达式优化
- 分支语句优化

删除冗余语句

在开发和修改程序时可能遗留有死代码，死代码是指程序在一个完整的执行过程该段代码并没有得到任何的运行，也可能是一些声明了但没有用到的变量，此时可以将其删除，避免程序在运行中进行不相关的运算行为，减少运行的时间。


```
int a=1,b=2;
int c,d;
  if(b>0)
     c=a+b;
  else
     c=a-b;
d=c+1;
d=a+2;//语句S
```

```
int a=1,b=2;
int c,d;
 d=a+2;//语句S
```

代数变换

程序员在编写程序时可能忽略了代数表达式也可以进一步优化，达到简化计算缩短运行时间的目的。此代码中的计算语句可以进行简化，原计算语句中含有乘法、加法和除法三种运算，而简化后仅剩乘法运算。

int a = 2, b = 3; *a = (a + a) + (6 * a) / 2;* *b = (b + b) + (6 * b) / 2;* *printf("a=%d,b=%d", a, b);*	*int a = 2, b = 3;* *a = 5 * a;* *b = 5 * b;* *printf("a=%d,b=%d", a, b);*

去除相关性-标量扩展

语句中依赖关系的存在非常不利于进行语序调整、向量化等优化方法的开展，且由于编译器优化的局限性，需要优化人员在编写程序时尽量消除依赖关系。

标量扩展是将循环中的标量引用用编译器生成的临时数组引用替换，可以有效地消除一些由内存单元的重用而导致的依赖。

for (int i = 1; i < N; i++) { * T = A[i];* * A[i] = B[i];* * B[i] = T;* *}*	*for (int i = 0; i < N; i++) {* * T[i] = A[i];* * A[i] = B[i];* * B[i] = T[i];* *}*

去除相关性-标量重命名

此循环中语句S1到S4之间存在真依赖，S1到S3之间存在输出依赖，这些语句间的依赖都是由于标量T引起的，此时可以通过引入两个不同的变量代替现有的变量T来消除依赖。

T = 2; //S1语句 *y = T + T; //S2语句* *T = a - b; //S3语句* *z = T * T; //S4语句*	*T = 2; //S1语句* *y = T + T; //S2语句* *T1 = a - b; //S3语句* *z = T1 * T1; //S4语句*

去除相关性-数组重命名

数组的存储单元有时被重用会导致不必要的反依赖和输出依赖，此时可以使用数组重命名的方法来消除。

```
for (int i = 1; i < N; i++) {
  A[i] = A[i - 1] + X; //S1语句
  Y[i] = A[i] + Z;  //S2语句
  A[i] = B[i] + C;  //S3语句
}
```

```
for (int i = 1; i < N; i++) {
  A1[i] = A[i - 1] + X; //S1语句
  Y[i] = A1[i] + Z;  //S2语句
  A[i] = B[i] + C;   //S3语句
}
```

数组重命名需要增加和数组大小成比例的额外内存空间，因此数组重命名的安全性和有利性都比标量重命名复杂，这种代价可能会严重影响到程序的性能，因此在实施数组重命名时应当更加谨慎。

公共子表达式优化

当程序中表达式含有两个或者更多的相同子表达式，仅需要计算一次子表达式的值即可

```
int a = 1, b = 5;//改进前需要计算三次a+b的值
if ((a + b) > 3 && (a + b) < 10) {
a = a + b;
}
```

```
int a = 1, b = 5;
int tmp = a + b;//改进后只需计算一次a+b的值
if (tmp > 3 && tmp < 10) {
a = tmp;
}
```

分支语句优化-合并判断条件

当程序中的分支判断条件是复杂表达式，优化人员可以将其进行优化。

int a1 = 1, a2 = 2, a3 = 3; *int a = 4, b = 5;* *//改进前* *if ((a1 != 0) && (a2 != 0) && (a3 != 0)){* *a = a + b;* *}*	*//改进后* *int temp = (a1 && a2 && a3);* *printf("%d\n", temp);* *if (temp != 0) {//简化分支判断条件，提高流水线性能* *a = a + b;* *}*

分支语句优化-生成选择指令

一些平台支持选择指令，选择指令是一个三目运算指令，在某些情况下可以将分支指令用选择指令进行替换，达到提升效率的目的。

int x; *int a = 4, b = 5;* *//改进前* *if (a > 0)* *x = a;* *else* *x = b;*	*int x;* *int a = 4, b = 5;* *x = (a > 0 ? a : b); //改进后--将分支判断移除，生成一条选择指令*

分支语句优化-运用条件编译

由于宏条件在编译时就已经确定，编译器可直接忽略不成立的分支，所以条件编译是在编译时判断。而普通分支判别是在运行时判断，故编译后的代码要长，效率也不如条件编译。

```
void arm_f() {
    printf("ON_ARM \n");
}
void  x86_f(){
    printf("ON_X86 \n");
}
```

```
#define ON_ARM 1
#ifdef ON_ARM
void arm_f() {
    printf("ON_ARM \n");
}#endif
#ifdef ON_X86
void  x86_f(){
    printf("ON_X86 \n");
}#endif
```

分支语句优化-移除分支语句

如果在程序设计时，编程人员能够将各分支路径的计算结果放到一张表中，并将分支条件转化为表中值对应的索引，那么就可以将分支跳转转化为访问表中元素，这是查表法移除分支的主要思想。

```
int score = 0;
printf("请输入你的成绩：");
scanf_s("%d", &score);
if (score >= 90) //score属于（0...100）
printf("A");
else if (score >= 80)
printf("B");
else if (score >= 70)
printf("C");
else
printf("D");
```

```
int score = 0;
printf("请输入你的成绩：");
scanf_s("%d", &score);
char s[] = { 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'C', 'B', 'A'
};
printf("%c", s[score / 10]);
```

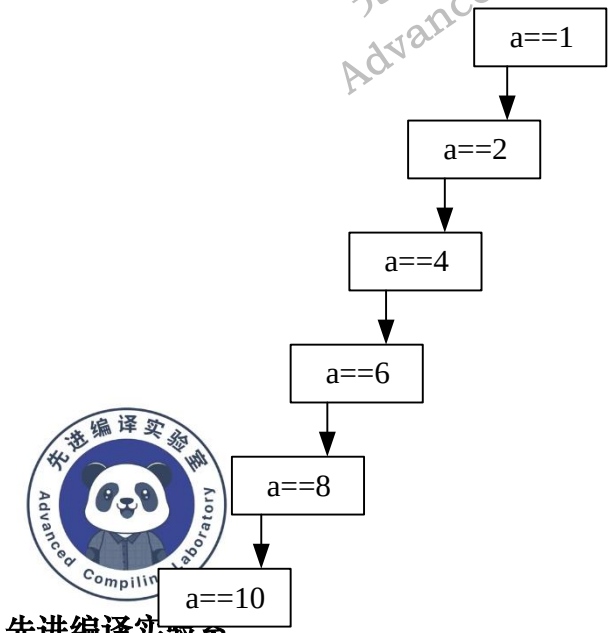
分支语句优化-平衡分支判断

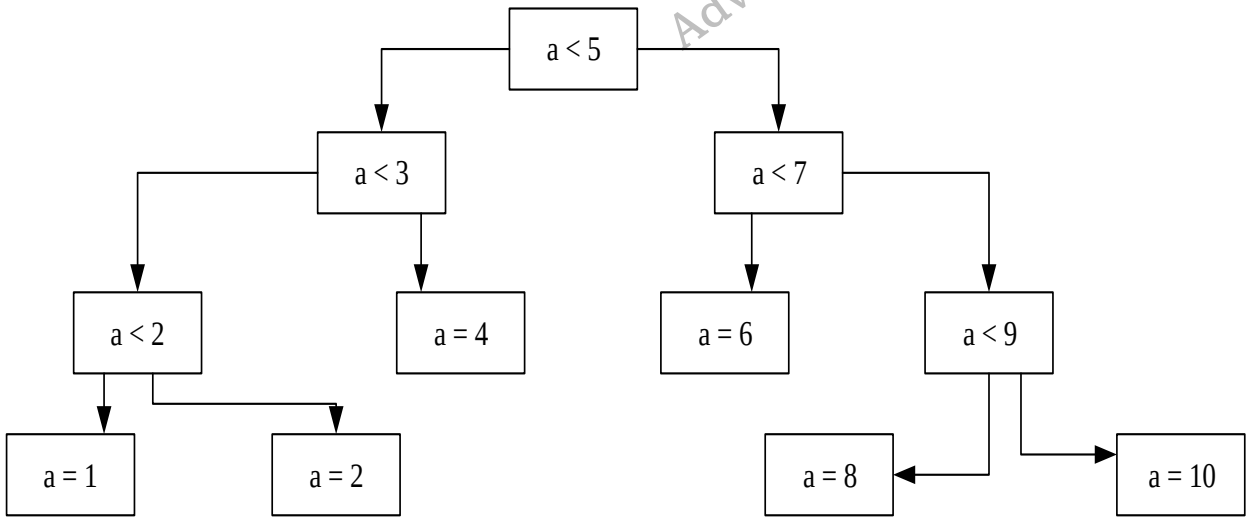
C语言中的switch运算符是程序员经常使用的一种语法，包含大量的分支，在一些程序中switch运算符可以含有数千个设置值，若直接实现这种需求的话，所得到的逻辑树会特别高，以下面switch分支语句为例。

```
switch (a){
  case 1 :fun1();
  case 2 : fun2();
  case 4 : fun4();
  case 6 : fun6();
  case 8 : fun8();
  case 10 : fun10();
}
```

分支语句优化-平衡分支判断

左图为分支语句对应的判断逻辑树，该代码所对应的优化逻辑树的高度为6，当a的值为10时需要6次判断，可以通过平衡判断分支的方法对分支语句进行优化，优化后逻辑判断树如右图所示，当a的值为10时需要4次判断，即平均仅需要4次比较操作就能完成判断。





参考资料

[1]薛联凤,秦振松. 编译原理及编译程序构造[M].南京东南大学出版社:, 201301.296.

AdvancedCompiler

Tel: 13839830713