程序优化

施朱鸣

11 月 4 日

Outline

优化编译器

优化区块

处理器级优化

优化编译器

减少重复计算同一个数

```
void set_row(double *a, double *b,
    long i, long n)
{
     long j;
     for (j = 0; j < n; j++)
          a[n*i+j] = b[j];
}
```

→

```
     long j;
     int ni = n*i;
     for (j = 0; j < n; j++)
          a[ni+j] = b[j];
```

用简单的计算代替

乘除法开销大，视情况用累加或者位移运算实现

$16 * x \rightarrow x << 4$

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

用简单的计算代替

分解复杂算式，把重复计算的数提出来

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

优化区块

编译器优化的局限性

- 为了不编译错，编译器只进行安全的优化
- 优化只在局部（procedures）进行
- 基于静态信息的优化
- 编译器不知道内存的实际情况

消除低效率循环

编译器看不到跨越函数的情况，不敢优化

“边带效应”：做了编译器没意识到的事情

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

消除不必要的访存

利用局部变量，但如果 B 和 A 相关可能会造成错误，编译器不敢优化

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

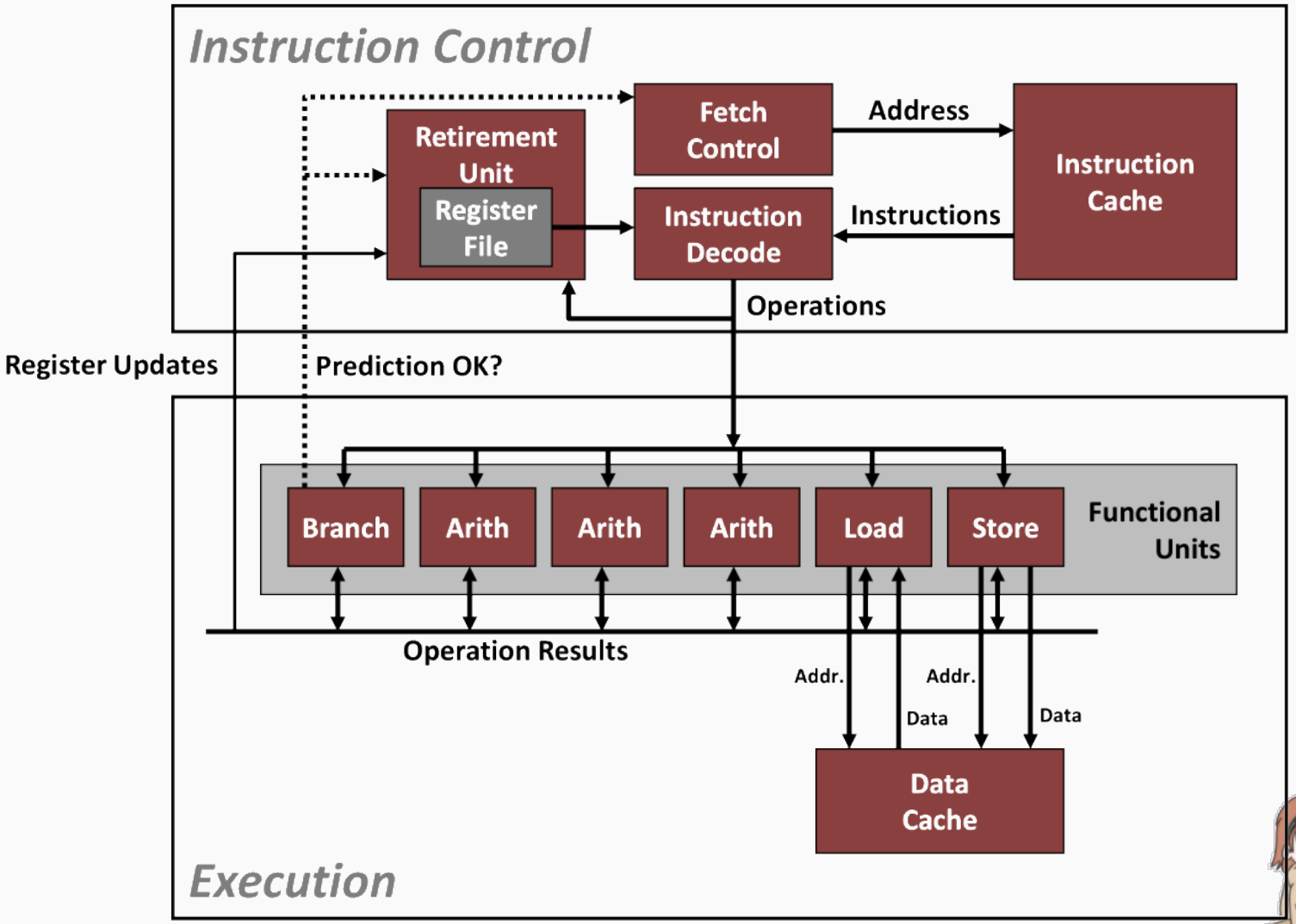
```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

减少不必要的访存

- 一段内存有多个别名时，如果编译器进行优化可能发生错误，所以要人工进行优化。
- C 中指针指来指去，编译器感到害怕

处理器级优化

超标量处理器


Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instruction Decode
Instructions
Operations
Register Updates
Prediction OK?
Branch
Arith
Arith
Arith
Load
Store
Functional Units
Operation Results
Addr.
Data
Addr.
Data
Data Cache
Execution

发射、延迟与容量

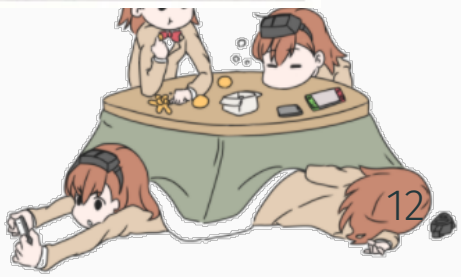
- 执行乱序，提交顺序
- 发射时间：两个连续同类运算之间需要的最小周期
- 延迟：完成运算需要的总时间
- 容量：能执行该运算的功能单元数量
- 吞吐量：发射时间的倒数

	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

发射、延迟与容量

- 执行乱序，提交顺序
- 发射时间：两个连续同类运算之间需要的最小周期
- 延迟：完成运算需要的总时间
- 容量：能执行该运算的功能单元数量
- 吞吐量：发射时间的倒数

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3～30	3～30	1	3～15	3～15	1



循环展开

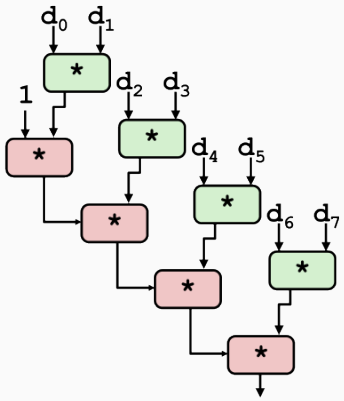
增大步长，减少循环次数，从而减少循环控制的开销

```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x OP d[i]) OP d[i+1];
}
/* Finish any remaining elements */
```

使用分配律重新结合

“让流水线流动起来”

但是有不可交换的风险（比如交错级数求和）



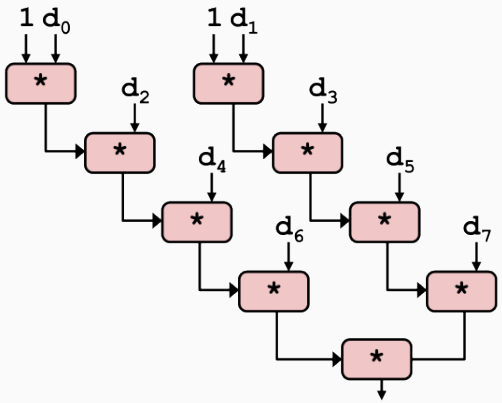

```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = x OP (d[i] OP d[i+1]);
}
/* Finish any remaining elements */
```

多个变量累积

“让流水线流动起来”

但要付出最后重新结合的代价，切的太多最后代价大

同样有不可交换的风险（比如交错级数求和）




```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 OP d[i];
    x1 = x1 OP d[i+1];
}
/* Finish any remaining elements */
```

循环展开和变量累积

展开 L 级，累积 K 个变量

风险：

- 寄存器溢出：累积变量数量超过可用寄存器数量（通常被吞吐上限制约）
- 有不可交换的风险，如交错级数奇偶项分开求和溢出

$$\sum_{i=1}^{\infty}(-1)^n\frac{1}{n}$$

- 最后处理 K 个变量开销不可忽略，可能得不偿失

各个方法的比较

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

各个方法的比较

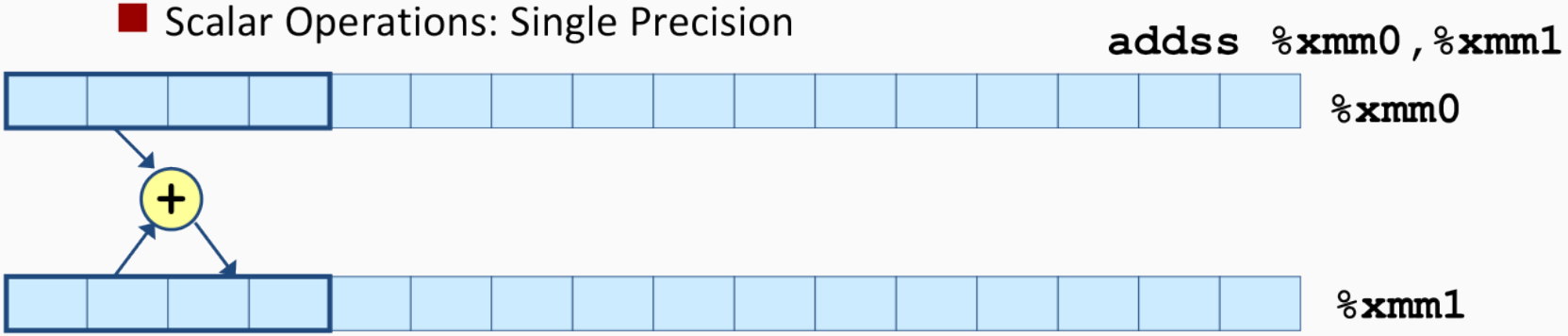
FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

各个方法的比较

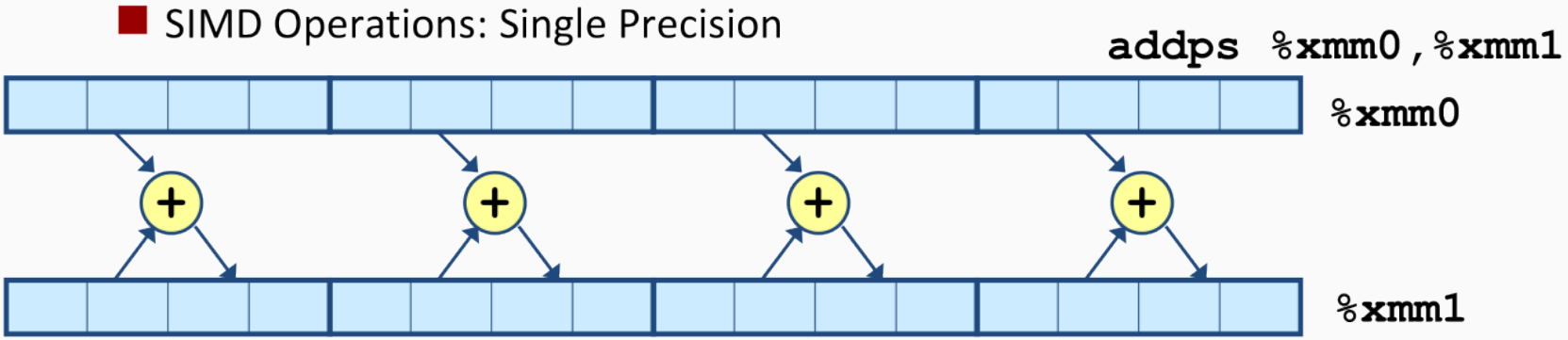
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

利用 AVS 向量计算

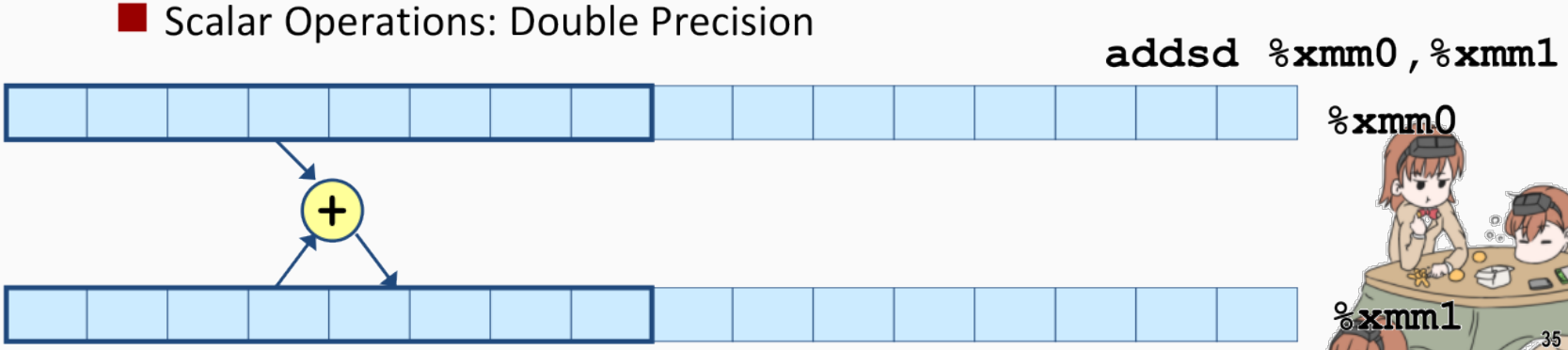
- Scalar Operations: Single Precision

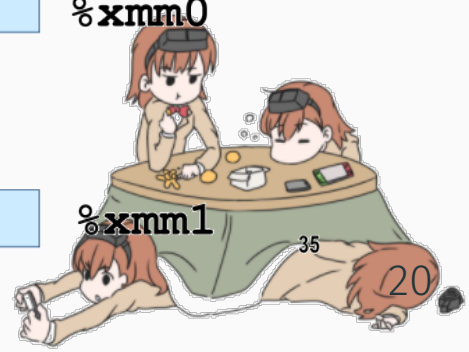


- SIMD Operations: Single Precision



- Scalar Operations: Double Precision





致谢

致谢

祝大家期中顺利，谢谢聆听！