计算机体系结构gem5实验四实验报告

PB21111681 朱炜荣

一、验证循环展开不影响代码的正确性

我的想法是创建一个diff函数，来比较不同函数执行结果之间是否相同。

```
void diff(double *Y, double *Y_unroll, const int N, int times)
{
    switch(times){
        case 1:
            printf("daxpy:");
            break;
        case 2:
            printf("daxsbxpxy:");
            break;
        case 3:
            printf("stencil:");
            break;
        default: break;
    }
    for (int i = 0; i < N; i++)
    {
        if (Y[i] != Y_unroll[i])
        {
            printf("Error: Y[%d] = %lf, Y_unroll[%d] = %lf\n", i, Y[i], i,
Y_unroll[i]);
            return;
        }
    }
    printf("Correct\n");
}
```

首先，我们将Y_unroll的各个元素赋值成与Y相同，然后修改unroll函数调用的数组为Y_unroll，然后再执行完两个函数之后使用我们定义的diff函数判断结果是否相同。

```
//std::random_device rd;
std::mt19937 gen(0);
std::uniform_real_distribution<> dis(1, 2);
for (int i = 0; i < N; ++i)
{
    X[i] = dis(gen);
    Y[i] = dis(gen);
    Y_unroll[i] = Y[i];
}

m5_dump_reset_stats(0, 0);
daxpy(X, Y, alpha, N);                          //2
m5_dump_reset_stats(0, 0);
daxpy_unroll(X, Y_unroll, alpha, N);            //3
m5_dump_reset_stats(0, 0);
diff(Y, Y_unroll, N, 1);
m5_dump_reset_stats(0, 0);
daxsbxpxy(X, Y, alpha, beta, N);                //5
m5_dump_reset_stats(0, 0);
daxsbxpxy_unroll(X, Y_unroll, alpha, beta, N);  //6
m5_dump_reset_stats(0, 0);
diff(Y, Y_unroll, N, 2);
m5_dump_reset_stats(0, 0);
stencil(Y, alpha, N);                           //8
m5_dump_reset_stats(0, 0);
stencil_unroll(Y_unroll, alpha, N);             //9
m5_dump_reset_stats(0, 0);
diff(Y, Y_unroll, N, 3);
```

测试结果如下：

```
daxpy:Correct
daxsbxpxy:Correct
stencil:Correct
```

二、展开循环的次数对性能的影响

展开循环之后性能展示如下所示

	CPI	numCycles	numInsts
daxpy	1.782900	142657	80014
daxpy_unroll	2.018038	141303	70020
daxsbxpxy	2.096461	251611	120017
daxsbxpxy_unroll	2.265867	249293	110021
stencil	1.963149	196309	99997
stencil_unroll	1.870726	182407	97506

通过观察numCycles的对比，我们发现循环展开的函数总体性能都得到了提升，执行相同的函数逻辑，所用的时钟周期数更少。

循环展开之后，主要消除的是控制hazard，每个展开的循环减少了循环控制指令的执行次数（如跳转指令），从而减少了分支预测失败的风险。而同样也减少了数据hazard，主要涉及RAW依赖，因为展开独立的命令可以使得寄存器分配更加合理，冲突减少。

对于每个循环，我都展开了4次，一般来说，展开次数应该是硬件能有效处理的指令数目，可以是2、4、8等等。如果没有展开的足够多或者展开的太多也会影响程序的性能，比如daxpy函数在展开2次、4次、16次和32次的性能比较。

	CPI	numCycles	numInsts
0	1.782900	142657	80014
2	1.825052	146037	80018
4	2.018038	141303	70020
16	2.200272	137561	62520
32	2.248618	137856	61307

不难发现，关于daxpy函数并非展开越多性能越好，但是下降之后的性能依旧大于因为本身这个函数的可并行性就非常高，所以会有这样的结果，而daxsbxpxy函数逻辑基本与daxpy相同，所以不再做新的测试，下面我们直接来看看stencil这个存在相互影响的函数效果如何，这里我们展开次数为2、4、16、32次的性能比较。

	CPI	numCycles	numInsts
0	1.963149	196309	99997
2	1.805861	207665	114995
4	1.870726	182407	97506
16	1.953700	164904	84406
32	1.986391	163325	82222

我们不难发现循环展开的次数过多，性能反而会有所上升，但是上升的效果不明显，而且中间可能存在更低点，由于逻辑较为复杂和数据相关过多，再加上循环展开可能与之前不同，所以我依旧认为展开更多性能会有所下降。

总体来看，目前测试的展开条件下，daxpy最优的展开次数为16次。太多的展开次数反而对得到下降的性能优化。

三、增加硬件对循环展开版本的函数和原函数的影响

通过添加硬件之后，具体的性能指标如下所示：

	CPI	numCycles	numInsts
daxpy	1.777939	142260	80014
daxpy_unroll	2.017209	141245	70020
daxsbxpxy	2.015040	241839	120017
daxsbxpxy_unroll	2.184410	240331	110021
stencil	1.963139	196308	99997
stencil_unroll	1.851937	180575	97506

对比增加硬件之前的各个性能指标，我们发现程序的CPI有所上升，但是性能上升的效果并不明显，因为循环中的逻辑本身就比较简单，增加的计算单元可能在大多数的时候都没有被派上用场，结构hazard有所下降，但是对应的数据hazard由于计算单元的增加会有所上升。

四、启动O3优化对函数性能的影响

在这里依旧是类似第二部分讨论的那样，对daxpy和stencil函数进行讨论分析。

	CPI	numCycles	numInsts
daxpy	1.821248	72890	40022
daxpy_unroll_4	2.021578	60710	30031
daxpy_unroll_16	2.209312	49778	22531
daxpy_unroll_32	2.263897	49036	21660
daxsbxpxy	2.063501	113540	55023
daxsbxpxy_unroll_4	1.583847	75285	47533
stencil	2.240021	134399	59999
stencil_unroll_4	3.427694	137214	40031
stencil_unroll_16	4.575626	149019	32568
stencil_unroll_32	4.771285	147948	31008

通过比较启动O3优化前后的性能，我们发现实际上手动的循环展开优化在有些程序上依旧可以使得程序性能得到一定的提升，但是提升的程度并不明显。也有可能产生负效果吗，比如你的展开不如编译器展开的智能甚至有可能产生了更多的数据相关冲突。实际上，现代编译器通常能够进行许多低级优化，包括循环展开、向量化等，手动优化可能效果有限。我们可以根据实际场景的不同需求来考虑是否进行手动循环展开优化。

对于同一个函数的不同版本，我觉得依旧应该使用numCycles的大小来表示函数的性能，因为这个指标是最直接、最重要的数据。我们对于一个结构完成相同的逻辑得到结果，肯定是希望能够越快越好，所以体现在gem5中也就是执行相同的逻辑所用的时钟周期数。