实验4 使用gem5探索指令级并行

1 改写后的`daxpy`文件

由于`N = 10000`，可对三个函数均手工循环展开5次。对于函数`stencil`，多次循环展开5次之后的余项不进行展开，按照原来的执行方式进行。

```
#include <cstdio>
#include <random>

#include <gem5/m5ops.h>

void daxpy(double *X, double *Y, double alpha, const int N)
{
    for (int i = 0; i < N; i++)
    {
        Y[i] = alpha * X[i] + Y[i];
    }
}

void daxsbxpxy(double *X, double *Y, double alpha, double beta, const int N)
{
    for (int i = 0; i < N; i++)
    {
        Y[i] = alpha * X[i] * X[i] + beta * X[i] + X[i] * Y[i];
    }
}

void stencil(double *Y, double alpha, const int N)
{
    for (int i = 1; i < N-1; i++)
    {
        Y[i] = alpha * Y[i-1] + Y[i] + alpha * Y[i+1];
    }
}


void daxpy_unroll(double *X, double *Y, double alpha, const int N)
{
    for (int i = 0; i < N/5; i++)
    {
        int k = 5 * i;
        Y[k] = alpha * X[k] + Y[k];
        k++;
        Y[k] = alpha * X[k] + Y[k];
        k++;
        Y[k] = alpha * X[k] + Y[k];
        k++;
        Y[k] = alpha * X[k] + Y[k];
        k++;
        Y[k] = alpha * X[k] + Y[k];
    }
}
```

```
void daxsbxpxy_unroll(double *X, double *Y, double alpha, double beta, const int
N)
{
    for (int i = 0; i < N/5; i++)
    {
        int k = 5 * i;
        Y[k] = alpha * X[k] * X[k] + beta * X[k] + X[k] * Y[k];
        k++;
        Y[k] = alpha * X[k] * X[k] + beta * X[k] + X[k] * Y[k];
        k++;
        Y[k] = alpha * X[k] * X[k] + beta * X[k] + X[k] * Y[k];
        k++;
        Y[k] = alpha * X[k] * X[k] + beta * X[k] + X[k] * Y[k];
        k++;
        Y[k] = alpha * X[k] * X[k] + beta * X[k] + X[k] * Y[k];
    }
}

void stencil_unroll(double *Y, double alpha, const int N)
{
    int iter = N/5 - 1;
    int first_remain = (N/5 - 1)*5 + 1;
    for (int i = 0; i < iter; i++)
    {
        int k = iter * 5 + 1;
        Y[k] = alpha * Y[k-1] + Y[k] + alpha * Y[k+1];
        k++;
        Y[k] = alpha * Y[k-1] + Y[k] + alpha * Y[k+1];
        k++;
        Y[k] = alpha * Y[k-1] + Y[k] + alpha * Y[k+1];
        k++;
        Y[k] = alpha * Y[k-1] + Y[k] + alpha * Y[k+1];
        k++;
        Y[k] = alpha * Y[k-1] + Y[k] + alpha * Y[k+1];
    }
    for (int i = first_remain;i < N-1; i++)
    {
        Y[i] = alpha * Y[i-1] + Y[i] + alpha * Y[i+1];
    }
}

int main()
{
    const int N = 10000;
    double *X = new double[N], *Y = new double[N], alpha = 0.5, beta = 0.1;

    //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);
    }

    m5_dump_reset_stats(0, 0);
    daxpy(X, Y, alpha, N);
```

```
    m5_dump_reset_stats(0, 0);
    daxpy_unroll(X, Y, alpha, N);
    m5_dump_reset_stats(0, 0);
    daxsbxpxy(X, Y, alpha, beta, N);
    m5_dump_reset_stats(0, 0);
    daxsbxpxy_unroll(X, Y, alpha, beta, N);
    m5_dump_reset_stats(0, 0);
    stencil(Y, alpha, N);
    m5_dump_reset_stats(0, 0);
    stencil_unroll(Y, alpha, N);
    m5_dump_reset_stats(0, 0);

    double sum = 0;
    for (int i = 0; i < N; ++i)
    {
        sum += Y[i];
    }
    printf("%lf\n", sum);
    return 0;
}
```

2 模拟输出结果统计

`simTicks`	1st	2nd (additional `HPI_FloatSimdFu()`)	3rd (`-O3` optimization based on 2nd)
`daxpy`	35552500	35552500	18248500
`daxpy_unroll`	37231750	37231750	17254000
`daxsbxpxy`	62844750	60345000	28388000
`daxsbxpxy_unroll`	64430500	62086750	18177250
`stencil`	49055750	49055750	33124000
`stencil_unroll`	43738750	43739250	37740250

`cpi`	1st	2nd (additional `HPI_FloatSimdFu()`)	3rd (`-O3` optimization based on 2nd)
`daxpy`	1.777314	1.777314	1.823847
`daxpy_unroll`	2.012010	2.012010	2.871599
`daxsbxpxy`	2.094528	2.011215	2.063719
`daxsbxpxy_unroll`	2.260303	2.178081	1.816091
`stencil`	1.962289	1.962289	2.208303
`stencil_unroll`	2.033297	2.033320	2.693520

committedInst	1st	2nd (additional HPI_FloatSimdFu())	3rd (-O3 optimization based on 2nd)
daxpy	80014	80014	45022
daxpy_unroll	74019	74019	30034
daxsbxpy	120017	120017	60023
daxsbxpy_unroll	114021	114021	45036
stencil	109995	109995	69998
stencil_unroll	86045	86045	56050

总结：

1. 循环展开后，指令条数都有不同程度的减少；
2. 除第三次实验中的`daxsbpxy`函数外，循环展开都使得CPI有一定程度的增大；
3. 函数的执行时间同时受指令条数和CPI的影响（在clock cycle大小不变的前提下），故当循环展开带来的指令条数的减小不足以抵消CPI增大的负面影响时，函数的执行时间反而更长；
4. 实验结果显示，编译器优化的效果显著好于手动循环展开。

3 问题解答

1

Question:

如何证明展开循环后的函数产生了正确的结果？

Answer:

在`daxpy`中，存在如下代码

```
    double sum = 0;
    for (int i = 0; i < N; ++i)
    {
        sum += Y[i];
    }
    printf("%lf\n", sum);
```

我们可以利用`printf`的结果对循环展开的正确性进行一定程度上的验证。若循环展开后的打印结果和初始（即各unroll函数中的内容和原函数中一样）的打印结果一致，则可以在一定程度上认为循环展开没有影响结果的正确性。

2

Question:

对于每一个函数，循环展开是否提升了性能？循环展开减少了哪一种hazard？

Answer:

在进行了手动循环展开后，各个函数的指令条数都有不同程度的减少。这是因为循环展开减少了可能存在的data hazard，使得存在data hazard的两条指令间可以插入一些不冲突的指令，减少流水线停顿的次数。同时，在进行了循环展开后，CPI很大可能会变大。当循环展开带来的指令条数的减小不足以抵消CPI增大的负面影响时，函数的执行时间反而更长。在初始的条件下，我们可以看到，在手动循环展开5次后，从最终的执行时间来看，只有`stencil`的性能得到了提升，而另外两个函数的性能反而有一定程度的下降。

3

Question:

你应该展开循环多少次？每个循环都一样吗？如果你没有展开足够多或展开太多会影响程序性能吗？

Answer:

在本次实验中，由于`N = 10000`，可对三个函数均手工循环展开5次。对于函数`stencil`，多次循环展开5次之后的余项不进行展开，按照原来的执行方式进行。当展开次数过少时，指令条数的减少可能不太明显，不足以抵消CPI增大的负面影响。只要指令调度得当，展开次数越多，指令条数的下降将更加显著，而CPI的增长在达到一定量后会趋于平缓，此时程序的性能将会得道更大的改善。

4

Question:

增加硬件对循环展开版本的函数和原函数有什么影响？添加更多硬件会减少哪种或哪些hazard？

Answer:

在实验中，增加硬件后函数的指令条数没有发生变化，`daxsbxpxy`、`daxsbxpxy_unroll`和`stencil_unroll`的CPI有少量的下降，导致这三个函数最终的执行时间有小幅度的改善。增加硬件可以减少浮点执行时的structure hazard，即硬件可以允许更多条浮点指令同时执行，而不需要在RS中持续等待。尤其当循环展开次数较多而浮点指令执行时间较长时，该效果更加显著。

5

Question:

选择你认为合适的指标比较四个版本函数的性能表现，为什么选择该指标？

Answer:

执行时间是比较各版本函数性能表现的一个更加合适的指标。在实验中，我们可以看到，指令条数和CPI的变换往往是逆相关的。这意味着，如果将指令条数和CPI中的任意一个作为评价性能的指标的话，对该指标的优化可能另一指标的大幅度恶化，从而导致最终的性能反而变差。同时，由于函数的执行时间同时受指令条数和CPI的影响（在clock cycle大小不变的前提下），将其作为性能的评价指标可以兼顾两方面的影响，从而更加客观和可靠。

6

Question:

你认为本次实验中你所进行的手动循环展开优化有意义吗？还是说编译器优化代码就已经足够了？说明理由。

Answer:

在已经进行较高编译器优化的前提下，手动循环展开的优化意义不大。从实验结果上看，进行`-O3`优化后，所有函数的性能都相比`-O1`优化的情况有了大幅度的提升。同时，在已经进行`-O3`优化后，手动循环展开的函数与原函数的性能差距不大，且在`-O3`优化下存在部分函数（`stencil`）在手动循环展开后性能反而有一定程度的损失。综上所述，在进行了较高程度（`-O3`）的编译器优化后，再进行手动循环展开的意义较小，不建议进行。