实验四 使用 gem5 探索指令级并行

Step 0. 实验目的

1. 体验优化代码增加指令级并行性，加深对指令级并行的理解
2. 了解有序处理器的 ILP 限制
3. 通过实际优化代码和分析数据，加深对性能分析的理解和认识

Step 1. 重新编译

本次实验使用 arm 指令集架构，因为 ARM 提供了一个高性能有序处理器的实现。

使用 ARM 会增加编译代码的额外复杂性。我们必须交叉编译应用程序，而不是仅仅运行 gcc 或任何其他常用的编译器。

首先要把实验仓库中的压缩包解压，其中的 gcc 解压并添加 PATH.

使用以下命令编译 ARM 版的 gem5：

```
python3 `which scons` build/ARM/gem5.opt
```

Step 2. 阅读 daxpy

> 需要注意的是该程序会一个接一个地运行所有六个函数，并在每个函数之间使用 m5_dump_reset_stats() 来分离统计不同函数地性能表现。因此，在程序运行时，你将有八个不同的统计转储，体现在 stat.ini 中

daxpy(double *X, double *Y, double alpha, const int N)

$$Y[i] = alpha * X[i] + Y[i]$$

daxsbxpxy(double *X, double *Y, double alpha, double beta, const int N)

$$Y[i] = alpha * X[i] * X[i] + beta * X[i] + X[i] * Y[i]$$

stencil(double *Y, double alpha, const int N)

$$Y[i] = alpha * Y[i-1] + Y[i] + alpha * Y[i+1]$$

剩下三个函数与前三个函数功能相同，需要手动展开。

在主函数中，会生成向量 X 和 Y，长度均为 10000。

主函数会调用上述六个函数， 每次调用函数前都会调用 `m5_dump_reset_stats`，这个函数用来重置并且输出模拟器的统计信息。这样做是为了评估每个函数的性能，例如运行时间、CPU使用情况等。通过重置统计信息，可以确保每次调用的统计信息不会被其他调用影响。

Step 3. 编译 daxpy

修改 Makefile 中编译器的路径，然后 `make` 编译

Step 4. 运行样例程序

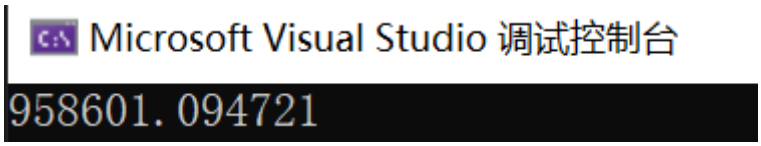
使用命令：

```
build/ARM/gem5.opt configs/example/arm/starter_se.py
/home/xxa/Desktop/daxpy/daxpy --cpu=hpi
```

运行结果：

```
958601.094721
exiting with last active thread context  @  960174000
```

将 `daxpy.cc` 的代码复制，删除其中 m5 相关内容， 实际计算内容保持不变，编译运行，结果：



对比可知，结果正确。

将输出的统计文件保存为 output0

Step 5. 重写三个函数的循环展开版本

每个函数循环展开两次，以第三个函数为例，上面是原函数，下面是循环展开版本：

```
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 stencil_unroll(double *Y, double alpha, const int N)
{
    int i;
    for (i = 1; i < N - 2; i += 2)
    {
        Y[i] = alpha * Y[i - 1] + Y[i] + alpha * Y[i + 1];
        Y[i + 1] = alpha * Y[i] + Y[i + 1] + alpha * Y[i + 2];
    }
    for (; i < N - 1; i++)
    {
        Y[i] = alpha * Y[i - 1] + Y[i] + alpha * Y[i + 1];
    }
}
```

可知每个函数中，仍然是串行计算 N 个公式，从 $Y[1]$ 计算到 $Y[N]$，只是之前是 N 轮循环，每轮循环计算一个 $Y[i]$，循环展开后循环次数减少，每轮循环计算的公式数增多，计算的内容不变。

在 Step 6 中会编译运行，通过运行结果也可以看出修改后不影响函数的执行结果。

Step 6. 编译运行

结果为：

```
958601.094721
exiting with last active thread context  @  955861000
```

与之前相同，可知我所做的修改不影响函数的执行结果。

将输出的统计文件保存为 output1

每个函数的要求统计指标：（循环展开次数为 2）

函数	CPI	simTicks	指令条数
daxpy	1.777964	35565500	80014
daxpy_unroll	1.824979	36508250	80019
daxsbxpxy	2.096470	62903000	120017
daxsbxpxy_unroll	2.117954	63549750	120021
stencil	1.962279	49055500	99997
stencil_unroll	2.190093	43806250	80008

- 指令条数参考指标 `system.cpu_cluster.cpus.numInsts`

通过运行结果可知（Step 6 的表格），三个函数在循环展开后 CPI 均增大，前两个函数发射的指令基本不变，而第三个函数发射的指令减少了 20%.

从运行时间上看，循环展开后，前两个函数的性能分别**下降**了约 2.6% 和 1.0%，而第三个函数的性能**提升**了 10.7%.

分析其原因：

提交的指令数变化：

对于前两个函数 `daxpy` 和 `daxsbxpxy`，循环展开后，每次迭代都做了两个元素的计算，因此看到的指令数基本不变，这是符合预期的。这是因为对于每一对元素，代码执行的计算和操作的数量没有改变，只是将它们在一个迭代中一起执行了。因此，总的指令数基本不变。

对于第三个函数 `stencil`，循环展开后，指令数有所减少，这是由于，循环展开前，在每次迭代中，都需要读取前后两个元素的值，即 `Y[i-1]` 和 `Y[i+1]`。在循环展开后，每次迭代处理两个元素，一些元素的值被重复使用，从而减少了一些读取操作。具体来说，`Y[i+1]` 在第 i 次迭代中被读取，然后在第 i+1 次迭代中又被读取。在循环展开后，这个值在一个迭代中就可以被重复使用，从而减少了一些读取操作。因此，看到的指令数减少了。

CPI 均增大：

循环展开后，编译后的汇编码长度增大，会导致 ICache Miss 率升高，这个可以由 `icache.demandMissLatency::total` 看出。

理论上循环展开后，循环的跳转条件变得复杂一些，会导致分支预测出错率升高。但是由于这里的循环比较简单，所以体现在 gem5 的输出指标上，展开前后的分支预测错误率没有大的变化

Step 7. 修改 HPI.py 文件

该文件位于 `configs/common/cores/arm/HPI.py`

为仿真处理器增加 SIMD/ 浮点处理单元，这将允许处理器并行执行更多的浮点操作，降低 CPI。

修改 class HPI_FUPool，在它的 funcUnits 的最后增加额外的三个 HPI_FloatSimdFU()

```
class HPI_FUPool(MinorFUPool):
    funcUnits = [HPI_IntFU(), # 0
        HPI_Int2FU(), # 1
        HPI_IntMulFU(), # 2
        HPI_IntDivFU(), # 3
        HPI_FloatSimdFU(), # 4
        HPI_MemFU(), # 5
        HPI_MiscFU(), # 6
        HPI_FloatSimdFU(),
        HPI_FloatSimdFU(),
        HPI_FloatSimdFU()
        ]
```

修改后重新使用 gem5 仿真执行 daxpy。

```
958601.094721
exiting with last active thread context  @  938533000
```

将输出的统计文件保存为 output2

Step 8. 使用 `-O3` 编译

修改 Makefile，编译运行：

```
958601.094721
exiting with last active thread context  @  3034337000
```

将输出的统计文件保存为 output3

Step 9. 回答问题

1. 如何证明展开循环后的函数产生了正确的结果?

首先，从代码层面上分析，之前是 N 轮循环，每轮循环计算一个 $Y[i]$。循环展开后循环次数减少，每轮循环计算的公式数增多，但是仍然是计算 N 个公式，从 $Y[1]$ 计算到 $Y[N]$，计算的内容不变。

从结果上来看，源代码的六个函数均会对 Y 向量做修改，最后会将 Y 的每个元素的和输出，所以可通过这个输出查看函数功能是否与之前（最早的版本是没有做循环展开的，结果一定正确）相同，运行后发现修改前后 C 代码的输出相同。

2. 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?

如果循环展开 2 次：通过运行结果可知（Step 6 的表格），三个函数在循环展开后 CPI 均增大，前两个函数发射的指令基本不变，而第三个函数发射的指令减少了 20%.（这一部分的原因分析请见 Step 6）。从运行时间上看，循环展开后，前两个函数的性能分别**下降**了约 2.6% 和 1.0%，而第三个函数的性能**提升**了 10.7%.

如果循环展开超过 4 次，：通过结果（表格和图像见第三题）可知，循环展开后，前两个函数的 CPI 均有上升，第三个函数的 CPI 下降，三个函数的提交指令条数减少。三个函数的运行时钟周期数均有减小。

循环展开减少了控制 hazard，尤其是对于循环控制的部分。循环展开减少了循环条件检查和跳转指令的数量，从而减少了控制 hazard。

3. 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？

以下是展开 2 次、4 次、8 次、16 次、32 次的运行结果。注意，此时编译优化为 `-O1`，且没有增加硬件。

循环展开 2 次：

函数	CPI	simTicks	指令条数
daxpy	1.777964	35565500	80014
daxpy_unroll	1.824979	36508250	80019
daxsbxpxy	2.096470	62903000	120017
daxsbxpxy_unroll	2.117954	63549750	120021
stencil	1.962279	49055500	99997
stencil_unroll	2.190093	43806250	80008

循环展开 4 次：

函数	CPI	simTicks	指令条数
daxpy	1.781163	35629500	80014
daxpy_unroll	2.008741	35161000	70016
daxsbxpxy	2.094220	62835500	120017
daxsbxpxy_unroll	2.264793	62292000	110018
stencil	1.962369	49057750	99997
stencil_unroll	1.886102	44797750	95006

循环展开 8 次：

函数	CPI	simTicks	指令条数
daxpy	1.781263	35631500	80014
daxpy_unroll	2.130091	34622500	65016
daxsbxpxy	2.094153	62833500	120017
daxsbxpxy_unroll	2.351473	61736750	105018

函数	CPI	simTicks	指令条数
stencil	1.962279	49055500	99997
stencil_unroll	1.923966	42093500	87514

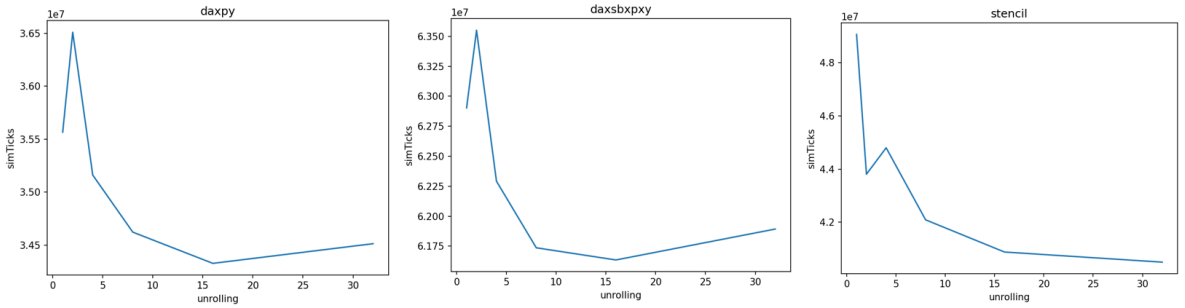
循环展开 16 次：

函数	CPI	simTicks	指令条数
daxpy	1.781263	35631500	80014
daxpy_unroll	2.196430	34328000	62516
daxsbxpxy	2.094153	62833500	120017
daxsbxpxy_unroll	2.404856	61635250	102518
stencil	1.962279	49055500	99997
stencil_unroll	1.952029	40885250	83780

循环展开 32 次：

函数	CPI	simTicks	指令条数
daxpy	1.781263	35631500	80014
daxpy_unroll	2.251957	34513500	61304
daxsbxpxy	2.094029	62829250	120016
daxsbxpxy_unroll	2.443818	61892750	101305
stencil	1.962079	49050500	99997
stencil_unroll	1.978049	40504500	81908

作图：



所以，综合来看，循环展开 16 次左右效果最好。

三个函数表现不同，前两个函数展开 16 次效果最优，但是第三个函数在展开 32 次后运行时间仍有所下降，只是下降幅度较小。

若没有展开足够多，会由于控制 Hazard 导致性能不佳。若展开的次数过多，会导致生成的指令数量过于庞大，不能全部装入指令 Cache，导致指令 Cache 命中率下降，从而影响性能。

4. 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?

循环展开的版本的统计指标见第三题。

增加硬件后，原函数和循环展开 2 次的函数表现为：

函数	CPI	simTicks	指令条数
daxpy	1.777939	35565000	80014
daxpy_unroll	1.824979	36508250	80019
daxsbxpxy	2.013156	60403250	120017
daxsbxpxy_unroll	2.045125	61364500	120021
stencil	1.962279	49055500	99997
stencil_unroll	2.136074	42725750	80008

CPI 和运行时间略有下降，降幅均小于 4%.

而对于循环展开效果比较明显的 16 次展开，增加硬件后，循环展开的函数的统计指标为：

函数	CPI	simTicks	指令条数
daxpy_unroll	2.196430	34328000	62516
daxsbxpxy_unroll	2.319700	59452750	102518
stencil_unroll	1.944581	40729250	83780

CPI 和运行时间略有下降，最明显的降幅也仅为 3.5%

添加更多硬件会减少结构相关。结构相关是由于硬件资源不足而造成的 hazard，增加硬件可以减少结构相关。

增加硬件也会减少数据相关。由于处理器可以并行执行更多的操作，这会减少数据相关。数据相关发生在一条指令依赖于另一条指令的结果，而这个结果尚未产生。增加硬件会使得一些结果更早地计算出来。

5. 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?

`simTicks`，模拟运行的时钟周期数。在上面的运行结果中可以看出，即使一些修改使得 CPI 升高，但同时可能由于提交的指令数减少，所以最终的运行时间也会减少；同时提交的指令数可能基本不变，但由于 CPI 增大，实际运行时间也会增加。所以单纯的 CPI 和提交的指令数均无法全面地反映函数的性能。

而运行时间则可以直观地反映函数的表现。

6. 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。

开 `-O3` 优化后，原函数和展开次数为 2 的循环展开函数的统计指标为：

函数	CPI	simTicks	指令条数
daxpy	1.822922	18239250	40022
daxpy_unroll	1.934023	19354250	40029
daxsbxpxy	2.075205	28546000	55023
daxsbxpxy_unroll	2.067217	28440250	55031
stencil	2.208287	33123750	59999
stencil_unroll	2.935602	33038000	45017

对比开 `-O1` 优化的版本：

函数	CPI	simTicks	指令条数
daxpy	1.777964	35565500	80014
daxpy_unroll	1.824979	36508250	80019
daxsbxpxy	2.096470	62903000	120017
daxsbxpxy_unroll	2.117954	63549750	120021
stencil	1.962279	49055500	99997

函数	CPI	simTicks	指令条数
stencil_unroll	2.190093	43806250	80008

可以看到，开 `-O3` 优化后，不同函数的 CPI 虽有增减，但是提交的指令数基本减为之间的一半，运行时间也有较大幅度的减少。

对比发现，开 `-O3` 优化的原函数，也要比手动循环展开的函数性能要好，而且是大幅度优于手动循环展开的版本。

现代编译器通常具有强大的优化能力，包括自动循环展开、矢量化、函数内联等。这意味着在很多情况下，编译器可以自动执行一些手动优化可能会执行的操作。在这些情况下，手动优化可能没有太大意义，而且可能使代码变得更难理解和维护。

不过本次实验代码比较简单，对于一些复杂的代码，编译器可能难以正确地理解其逻辑并进行有效的优化。在这些情况下，手动优化可能会带来显著的性能提升。

但对于本次实验而言，编译器优化代码已经足够了。

附录

- `daxpy.cc`：原函数和循环展开 2 次的函数
- `daxpy4.cc`、`daxpy8.cc`、`daxpy16.cc`、`daxpy32.cc`：原函数和循环展开次数分别为 4、8、16、32 的函数。
- `output1`：循环展开 2 次后，gem5 模拟器的输出
- `output2`：增加硬件后，gem5 模拟器的输出
- `output3`：增加硬件且开启 `-O3` 优化后，gem5 模拟器的输出
- `HPI.py`：增加硬件时修改的函数，位于`configs/common/cores/arm/HPI.py`
- `PB20061343_徐奥_lab4.pdf`：实验报告