Dhrystone 与 Whetstone 等 Benchmark 评测报告

姓名： 张煌昭
学号： 1400017707
学院： 元培学院
邮箱： zhang_hz@pku.edu.cn
手机： 17888838127

一． 工作背景和测评环境

本次实验通过使用 Dhrystone 和 Whetstone 对个人电脑进行测评，掌握了 Benchmark 的使用和评估方法，对于 Benchmark 的重要性和局限性也有了比较深入的认识和理解。

二． 测评环境

Table 1. 实验测评环境

项目	详细指标和参数
处理器型号及相关参数	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 8-way set-associative, 64 sets 32KB size, Level-1 D-/I-Cache 4-way set-associative, 1024 sets 256KB size, Level-2 Cache 16-way set-associative, 4096 sets 4096KB size, Level-3 Cache
内存	4GB Main Memory
外存	VBOX HARDDISK 1.0, 10GB device size 512byte logical/physical sector size
操作系统及其版本	Linux 4.10.0-28-generic 16.04.2-Ubuntu x86_64 GNU/Linux
编译器版本	GCC version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)

三． 测评步骤及要求

1. Dhrystone-2.1 Benchmark

1.1 于 Linux 环境下，使用 Dhrystone-2.1 提供的 Makefile 编译 Dhrystone；

1.2 分别采用 10^8，$3*10^8$，$5*10^8$，$7*10^8$，$9*10^8$ 作为输入次数，运行编译生成的程序，记录、处理相关数据，并做出解释；

1.3 对 Dhrystone 代码进行三行内的修改，使其运行结果不变而分数提升；

1.4 讨论采用 Dhrystone 进行测评存在哪些可以改进的地方，对其做出修改并说明，之后再次进行评测。

2. Whetstone-1.2 Benchmark

2.1 在 Linux 环境下分别采用-O0、-O2、-O3 选项对 Whetstone 程序进行编译并执行，记录评测结果；

2.2 分别采用 10^6、10^7、10^8、10^9 为输入次数，运行编译生成的可执行程序，记录、处理相关数据并做出解释；

2.3 通过使用新的编译选项等，进一步改进 Whetstone 的程序性能。

3. Wrk 4.0.2 HTTP Benchmark

3.1 在 Linux 环境下，下载编译 Wrk 源码，并测试执行；

3.2 运行 Wrk 测试不同线程数和连接数，在长时（3 min）和短时（30 s）下的 http 连接，记录处理测评结果并作出解释

4. CNN Benchmark

4.1 在 Linux 环境下，编译所给 CNN 代码，并测试执行；

4.2 运行 CNN，测试不同编译优化选项，记录、处理相关数据并做出解释。

四． 测评结果及分析

1. Dhrystone-2.1 Benchmark

1.1 编译 Dhrystone

dhrystone-2.1 目录下 make, 报错如下 :"dhry_1.c:48:17: error: conflicting types for 'times'", 按照报错提示，将 dhry_1.c 中的 times 函数声明修改为"extern clock_t times(struct tms *__buffer);"后，make 不再报错。最终生成 gcc_dry2，gcc_dry2reg，cc_dry2，cc_dry2reg 四个可执行文件。

1.2 运行 Dhrystone 测试

由于 Dhrystone 需要在进程内进行输入，不方便脚本执行，因此仿照 Dhrystone-2.2 对源码进行修改，将手动输入循环轮数改为读取命令参数。重新 make 后，编写 python 脚本，用规定的循环轮数运行生成的四个可执行文件。

运行结果入下表。

Table 2. Dhrystone 测试结果-1

循环	gcc_dry2		gcc_dry2reg		cc_dry2		cc_dry2reg	
轮数	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s
$1*10^8$	0.0628	$1.59*10^7$	0.0570	$1.75*10^7$	0.0465	$2.15*10^7$	0.0482	$2.08*10^7$
$3*10^8$	0.0593	$1.69*10^7$	0.0561	$1.78*10^7$	0.0479	$2.09*10^7$	0.0478	$2.09*10^7$
$5*10^8$	0.0595	$1.68*10^7$	0.0568	$1.76*10^7$	0.0476	$2.10*10^7$	0.0470	$2.17*10^7$
$7*10^8$	0.0575	$1.74*10^7$	0.0558	$1.79*10^7$	0.0475	$2.11*10^7$	0.0464	$2.16*10^7$
$9*10^8$	0.0581	$1.72*10^7$	0.0706	$1.42*10^7$	0.0474	$2.11*10^7$	0.0569	$1.76*10^7$

ms/run: Microseconds for one run through Dhrystone;

Dhry/s: Dhrystones per Second

根据上表绘制折线图如下，发现没有使用 register 编译参数的 gcc_dry2 和 cc_dry2 运行时，CPU Time 并没有随着循环数增加而出现明显的弯折，而使用了 register 参数的

gcc_dry2reg 和 cc_dry2reg 却在 $9*10^7$ 时 CPU Time 出现了明显的上升。此外，发现 gcc 选项编译的结果，比 cc 选项编译的结果，相同循环轮数的条件下，CPU Time 更长。

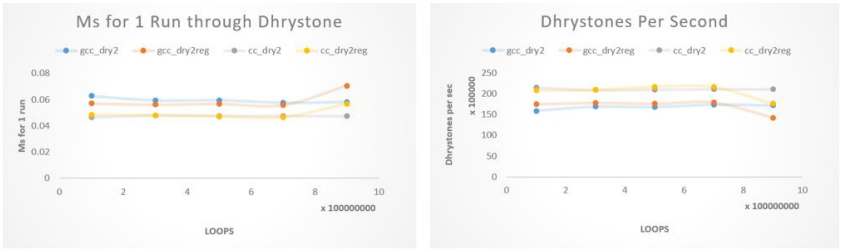

Figure 1. Dhrystone 测评结果-1

为了验证 register 参数带来的影响（即 CPU Time 在某一阈值之上会随循环数增加而增加，每秒 Dhrystone 数在某一阈值之上会随循环轮数增加而降低），再次进行如下实验：采用 $5*10^8$，$7*10^8$，$9*10^8$，$11*10^8$，$13*10^8$ 作为输入次数，运行编译生成的四个程序，记录结果如下表，并绘制曲线如下图。

Table 3. Dhrystone 测试结果-2

循环	gcc_dry2		gcc_dry2reg		cc_dry2		cc_dry2reg	
轮数	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s
$5*10^8$	0.0545	$1.83*10^7$	0.0557	$1.80*10^7$	0.0459	$2.18*10^7$	0.0463	$2.16*10^7$
$7*10^8$	0.0597	$1.67*10^7$	0.0562	$1.78*10^7$	0.0457	$2.19*10^7$	0.045	$2.22*10^7$
$9*10^8$	0.0569	$1.76*10^7$	0.0584	$1.71*10^7$	0.0457	$2.19*10^7$	0.0459	$2.18*10^7$
$11*10^8$	0.0573	$1.75*10^7$	0.0575	$1.74*10^7$	0.046	$2.17*10^7$	0.0451	$2.22*10^7$
$13*10^8$	0.0555	$1.80*10^7$	0.0554	$1.80*10^7$	0.0454	$2.20*10^7$	0.0453	$2.21*10^7$

ms/run: Microseconds for one run through Dhrystone;

Dhry/s: Dhrystones per Second

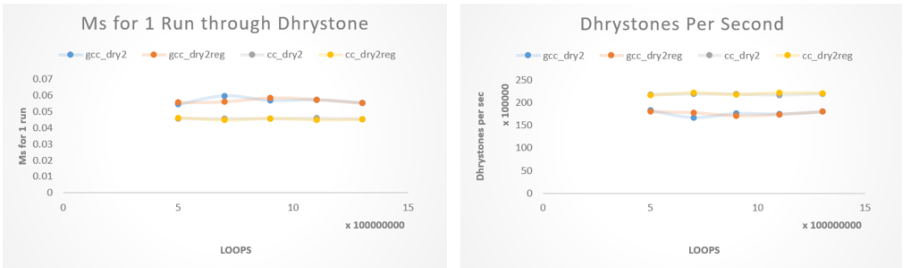

Figure 2. Dhrystone 测评结果-2

1. 3 修改 Dhrystone

利用 gprof 工具对 Dhrystone 进行剖视，发现 Proc_1 和 Proc_8 时间占比最高，因而考虑对这两个函数进行修改。阅读源码，发现 Proc_8 中存在一个 for 循环，该循环只会执行 2 次，考虑将其两次执行的语句直接顺序写出，从而避免循环的跳转预测错误引起的时间损失。

完成改动后重新编译，使用 gcc_dry2 和 cc_dry2 按照 1.2 的步骤进行实验，得到实验结果如下表，绘制曲线如下图。

Table 4. 修改 Dhrystone 代码测试结果

循环	Modified cc		Old cc_dry2		Modified gcc		Old gcc_dry2	
轮数	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s
$1*10^8$	0.045	$2.22*10^7$	0.0457	$2.19*10^7$	0.0552	$1.81*10^7$	0.067	$1.49*10^7$
$3*10^8$	0.0453	$2.21*10^7$	0.0468	$2.14*10^7$	0.0539	$1.85*10^7$	0.0561	$1.78*10^7$
$5*10^8$	0.0449	$2.23*10^7$	0.0461	$2.17*10^7$	0.0539	$1.86*10^7$	0.0548	$1.82*10^7$
$7*10^8$	0.0447	$2.24*10^7$	0.0461	$2.17*10^7$	0.053	$1.89*10^7$	0.0544	$1.84*10^7$
$9*10^8$	0.0447	$2.24*10^7$	0.047	$2.13*10^7$	0.0544	$1.84*10^7$	0.0563	$1.78*10^7$

ms/run: Microseconds for one run through Dhrystone;

Dhry/s: Dhrystones per Second

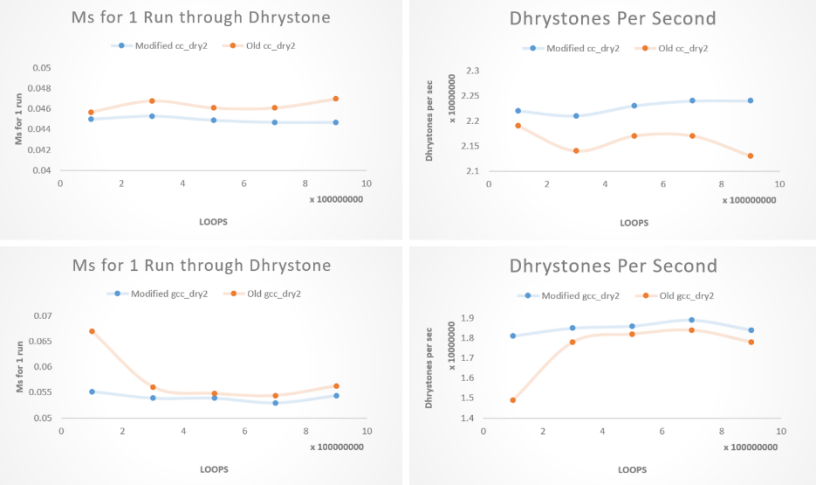

Figure 3. Modified Dhrystone 测评结果

以上图表说明此处针对 Proc_8 的循环语句的修改，有效地降低了 Dhrystone 的 CPU Time，该修改优化是成功的。

1. 4 改进 Dhrystone

在上一部分的试验中，已经发现 Dhrystone 中的循环和分支引起的时间损失会使得 CPU Time 上升，因而考虑到可以采用编译优化选项对跳转指令进行预测优化。查阅 gcc 编译器手册，发现从优化选项-O1 起（-O1，-O2，-O3），均会对分支指令进行预测。分别修改 Makefile 中的 GCCOPTIMIZE 选项为-O0，-O1，-O2，-O3 后 make，使用 gcc_dry2 进行与上面实验设置相同的实验，得到试验结果如下表，绘制曲线如下。

Table 5. Dhrystone gcc_dry2 在不同编译优化选项下的测试结果

循环	-O0		-O1		-O2		-O3	
轮数	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s	ms/run	Dhry/s
$1*10^8$	0.0578	$1.73*10^7$	0.0467	$2.14*10^7$	0.0467	$2.14*10^7$	0.046	$2.17*10^7$
$3*10^8$	0.0582	$1.72*10^7$	0.0476	$2.10*10^7$	0.0487	$2.05*10^7$	0.0478	$2.09*10^7$
$5*10^8$	0.0568	$1.76*10^7$	0.0473	$2.11*10^7$	0.0483	$2.07*10^7$	0.0488	$2.05*10^7$
$7*10^8$	0.0585	$1.71*10^7$	0.0483	$2.07*10^7$	0.0464	$2.16*10^7$	0.0481	$2.08*10^7$
$9*10^8$	0.058	$1.72*10^7$	0.0476	$2.10*10^7$	0.0471	$2.12*10^7$	0.0474	$2.11*10^7$

ms/run: Microseconds for one run through Dhrystone;

Dhry/s: Dhrystones per Second

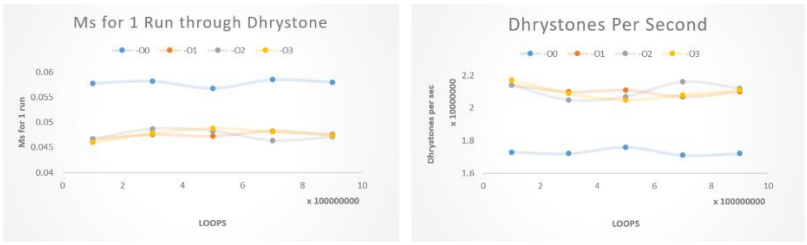

Figure 4. Dhrystone 在不同编译优化选项下的测评结果

观察以上图表可以发现-O1 选项相比于-O0 选项，CPU Time 有明显的下降，并且-O2，-O3 选项基本与-O1 选项相同。这说明-O1 优化选项内针对 Dhrystone 所做的优化是有效的。

2. Whetstone-1.2 Benchmark

2. 1 编译 Whetstone

对 whetstone.c 编写 Makefile，直接 make 从-O0 到-O3 的四种优化选项，并且可以通过修改 Makefile 中的 FLAGS 来添加其他的编译选项或优化选项。make 生成 wet0，wet1，wet2，wet3 四个可执行文件（数字对应优化选项）。

此外仿照 Dhrystone 的实验，编写 python 脚本方便试验中的脚本执行。

2. 2 运行 Whetstone 测试

使用 Makefile，make 后得到上述四个可执行文件，使用 python 脚本运行试验——wet0 到 wet3 各运行 10^5，10^6，10^7，10^8 轮（由于 10^9 轮在实验机器上已经运行 10 小时还没有结果，按照实验数据的规律，预计用时在万秒级别，为了省时考虑，去除了这一组实验，并添加规定各个单次实验的耗时上限为 10000 秒，超过这一时间则终止此次实验）。

得到实验结果如下表。

Table 6. Whetstone 测试结果

循环	wet0		wet1		wet2		wet3	
轮数	Dura	MIPs	Dura	MIPs	Dura	MIPs	Dura	MIPs
10^5	3	3333.3	2	5000.0	1	10000.0	1	10000.0
10^6	54	1851.9	35	2857.1	22	4545.5	22	4545.5
10^7	506	1976.3	277	3610.1	161	6211.2	152	6578.9
10^8	5056	1977.8	2772	3607.5	1611	6207.3	1533	6523.2

Dura: Duration per Iteration (sec); MIPs: C Converted Double Precision Whetstones (MIPs)

--: Meaning insufficient duration, the LOOP count should increase.

根据上表绘制曲线如下图，发现循环轮数对 Whetstone 检测有较大的影响，当循环轮数足够大时，Whetstone 检测趋于稳定；此外，-O0 到-O3 各个优化选项带来的影响基本符合预期，-O2 和-O3 几乎没有性能提升，因为-O3 与-O2 最主要的差别是对于 inline 的优化，而 Whetstone 中没有这一类函数。

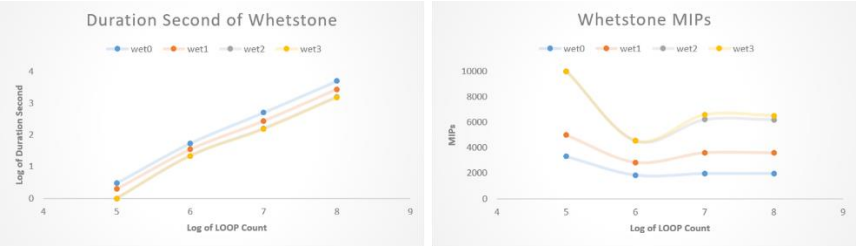

Figure 5. Whetstone测评结果

2. 3 提升 Whetstone 性能

由于 Whetstone 是针对浮点运算的测评，可以利用编译器针对浮点运算的优化选项进行优化。通过使用 man gcc 命令查看 gcc 手册，发现 gcc 编译器中针对浮点运算的选项有-ffloat-store 和-ffast-math。其中-ffloat-store 选项是迫使程序按照浮点国际定义进行存放(而非 Intel 的 80 位浮点数)；-ffast-math 对 math 库进行优化，大幅提升其速度，但也有可能会产生一些运算错误。

在此次试验中，使用-ffast-math 选项对 Whetstone 性能进行优化，使其运算速度大幅度提升。在 Makefile 中添加 FASTMATHFLAGS = -ffast-math 后 make，得到 fwet0，fwet1，fwet2 和 fwet3 四个可执行文件。使用 python 脚本重复上一部分的实验。

实验结果如下表。

Table 7. 使用-ffast-math 优化选项的 Whetstone 测试结果

循环	fwet0		fwet1		fwet2		fwet3	
轮数	Dura	MIPs	Dura	MIPs	Dura	MIPs	Dura	MIPs
10^5	3	3333.3	2	5000.0	--	--	--	--
10^6	31	3225.8	15	6666.7	2	50000.0	2	50000.0
10^7	308	3246.8	140	7142.9	20	50000.0	20	50000.0
10^8	3112	3213.4	1447	6910.9	203	49261.1	204	49019.6

Dura: Duration per Iteration (sec); MIPs: C Converted Double Precision Whetstones (MIPs);

--: Meaning insufficient duration, the LOOP count should increase.

根据上表绘制曲线图如下。

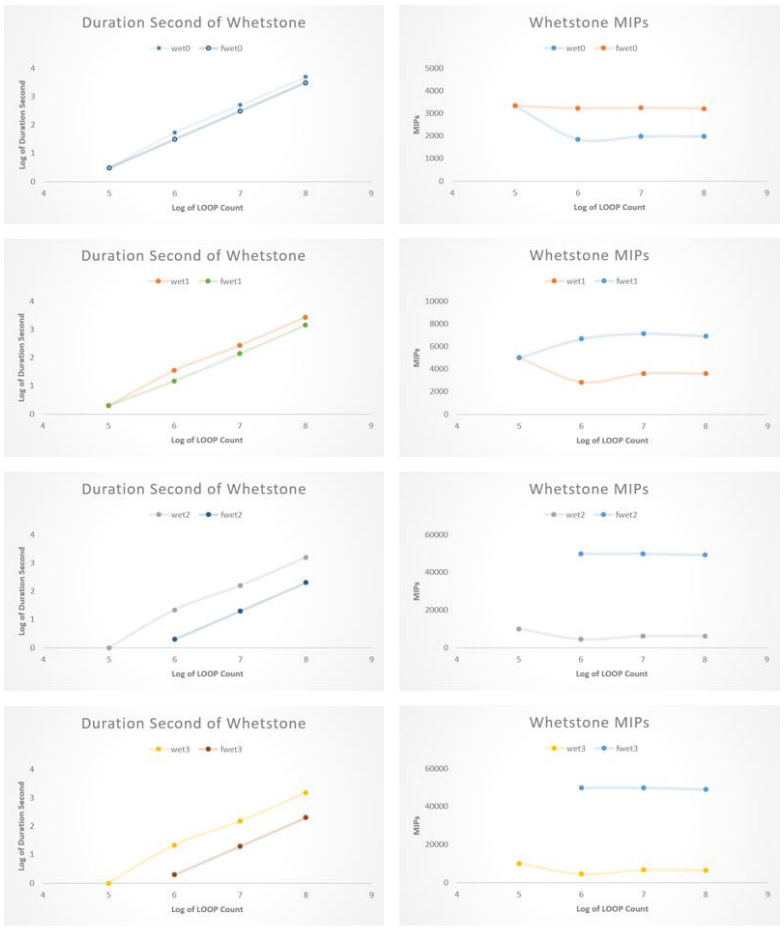

Figure 6. –ffast-math优化的Whetstone测评

发现使用-ffast-math 编译优化选项之后，-O0 至-O3 各个优化 Whetstone 运行时间明显降低，MIPs 数值明显上升；并且优化等级越高，增加-ffast-math 选项后的性能提升越高。说明-ffast-math 编译选项可以有效针对 Whetstone 进行性能提升。

3. Wrk 4.0.2 HTTP Benchmark

3. 1 编译 Wrk

使用 git 工具，用“git clone https://github.com/wg/wrk.git”命令获取 Wrk 开源源码，进入 wrk 目录 make，得到 wrk 可执行文件。

Wrk 一般有如下需要设定的参数：“-c”连接数 c，“-t”线程数 t，即用 t 个线程异步模拟 c 个连接；“-d”测试时间；“-T”超时时间 T，即一个连接时间超过 T 未响应，则认为连接失败。

使用 http://www.baidu.com 作为测试 http 连接的网址，得到测试报告如下图。

```
lc@lc-VirtualBox:~/下载/lab1/Lab1-1-src/wrk$ ./wrk -t8 -c100 -d30s http://www.ba
idu.com
Running 30s test @ http://www.baidu.com
  8 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   736.92ms  379.49ms   2.00s    73.96%
    Req/Sec    15.96     10.30    70.00     70.13%
  3254 requests in 30.03s, 48.07MB read
  Socket errors: connect 0, read 0, write 0, timeout 192
Requests/sec:    108.34
Transfer/sec:      1.60MB
```

测试报告中包括：线程的延迟和每秒处理请求数，具体有平均值，标准差，最大值，和一个标准差内占比；发送的总请求数，用时，读取内容大小；Socket 错误（连接错误，读写

错误，超时）数目；总计每秒发送请求数和每秒传送数据大小。

3.2 测试 Wrk

分别用 Wrk 测试 2，4，8 线程模拟 10，100，1000 连接数，测试时间为 10 s 和 3 min，总计 18 组实验，规定 T=5 sec。编写 python 脚本，以方便测试自动进行。

实验结果如下表。

Table 8. Wrk 测试结果

Test Deration = 10 sec; Test Website: https://www.baidu.com										
Th#	**Con#**	**Thread Latency**			**Thread Req/Sec**			**Err**	**Total**	**Total**
		Avg	**Stdev**	**Ratio**	**Avg**	**Stdev**	**Ratio**		**Req/s**	**Trans/s**
2	10	36.1	10.6	90.9%	139.6	18.3	74.2	None	276.1	4.1
2	100	218.8	262.6	90.6%	299.0	105.1	81.1%	None	537.1	8.0
2	1000	436.3	609.6	90.7%	218.7	1534.0	58.5%	3t	312.5	4.7
4	10	34.5	12.8	97.1%	58.6	9.7	76.0%	None	231.7	3.4
4	100	171.5	137.5	89.7%	161.3	43.1	81.4%	None	584.8	8.7
4	1000	483.6	487.3	87.9%	136.4	82.8	67.5%	None	417.2	6.4
8	10	41.6	21.9	89.5%	24.9	8.1	81.0%	None	195.2	2.9
8	100	232.4	227.1	88.7%	63.6	24.1	70.6%	None	456.0	6.8
8	1000	508.9	585.9	87.3%	60.1	49.0	60.8%	2t	299.2	4.6
Test Deration = 180 sec; Test Website: https://www.baidu.com										
Th#	**Con#**	**Thread Latency**			**Thread Req/Sec**			**Err**	**Total**	**Total**
		Avg	**Stdev**	**Ratio**	**Avg**	**Stdev**	**Ratio**		**Req/s**	**Trans/s**
2	10	38.4	30.1	97.2%	140.7	23.4	81.1%	None	279.0	4.1
2	100	178.0	202.9	92.5%	341.3	63.1	82.8%	3t	670.8	9.9
2	1000	601.7	573.2	89.7%	360.1	82.4	75.4%	3r527t	709.0	10.5
4	10	40.4	53.9	97.7%	57.6	11.4	66.5%	None	226.7	3.3
4	100	175.2	174.6	91.5%	167.2	33.9	69.8%	2t	662.3	9.8
4	1000	414.2	440.2	91.3%	180.1	65.4	67.9%	215t	706.3	10.4
8	10	34.3	8.1	96.3%	29.3	4.4	81.9%	None	234.2	3.5
8	100	162.3	152.9	91.1%	85.3	21.5	67.4%	None	678.1	10.0
8	1000	604.67	659.3	90.3%	85.0	62.4	66.3%	744t	616.9	9.1

Th#: Number of Threads; Con#: Number of Connections.

Thread Latency (ms); Ratio = Number of cases within area of one Stdev / Number of all cases.

Err: Number of Errors. c: Number of connection errors; r: Number of read errors; w: Number of write errors; t: Number of timeout errors.

Total Trans/s: Total Transfer per Sec (MB).

首先，发现测试时间对于测试精确度的影响很大，测试时间越长，线程延迟和请求的均值和方差越为准确；此外，异步线程数对于 http 传输的影响不大，而连接数的影响较大，且存在一个阈值，达到该阈值之后，几乎不再产生影响。

4. CNN Benchmark

4.1 编译 CNN

编写 Makefile，make 后得到可执行文件 cnn。使用命令./cnn 运行可执行文件，执行输出 CNN 卷积神经网络的运行时间。

4. 2 测试 CNN

修改 Makefile，添加编译优化选项 OPTIMIZEFLAGS 和编译浮点选项 FLOATFLAGS，使用 -O0 至-O3 和-ffast-math/-ffloat-store，分别运行编译生成的可执行文件，测试结果如下表。

Table 9. CNN Inference 测评结果

OPTIMIZEFLAGS	FLOATFLAGS	Time used (Sec)
-O0	None	18.2763
-O1	None	6.2362
-O2	None	3.3224
-O3	None	3.5503
-O0	-ffast-math	17.7976
-O1	-ffast-math	6.2635
-O2	-ffast-math	3.2531
-O3	-ffast-math	3.7998
-O0	-ffloat-store	24.9810
-O1	-ffloat-store	21.8967
-O2	-ffloat-store	7.8584
-O3	-ffloat-store	9.66453
-O0	-ffast-math + -ffloat-store	26.5218
-O1	-ffast-math + -ffloat-store	22.4704
-O2	-ffast-math + -ffloat-store	8.4429
-O3	-ffast-math + -ffloat-store	9.7428

根据上表，绘制曲线图如下，横坐标为优化等级（比如 0 表示-O0，1 表示-O1 等）

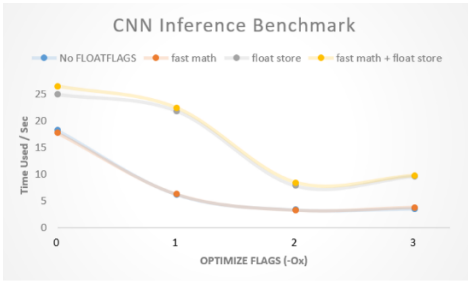

Figure 7. CNN Inference测评结果

结论如下：随着优化等级提高（-O0 到-O1 到-O2），CNN 运行时间越短；-O2 与-O3 优化等级几乎没有差别，因为-O3 针对链接优化和 inline 函数优化，而 CNN 项目中基本没有这一类优化。此外，浮点优化选项对于该 CNN 没有提升，-ffast-math 不造成影响，而-float-store 会降低 CNN 性能。

五． 小节

本次实验，针对 Whetstone Benchmark 和 Dhrystone Benchmark，以及其他不同的 Benchmark，进行了较为深入的了解和实验。明确了 Benchmark 的代表性作用，但也发现其易于被针对优化的特点。此外，对于 Linux 环境 Makefile 以及其它脚本的使用和编写更为熟

悉，对于 gcc/g++编译选项也更为熟悉。