

Project 1

Introduction

這個 project 主要的目的是要讓我們熟悉本堂課重要的計算機性能分析工具：sniper，以及模擬和分析不同記憶體階層和 cache size 所帶來的不同效能。

Install Sniper and test the result

首先，我們要透過 Virtual Box 來安裝 sniper，因此先到 Virtual Box 的官網下載 Windows 版本的應用程式，接著下載並解壓縮 sniper.rar。到 Virtual Box 後，新增一個虛擬機器，虛擬硬碟檔案使用 sniper 裡面的 sniper.vmdk，將記憶體、硬碟大小等配置完後虛擬機器即建立完成。

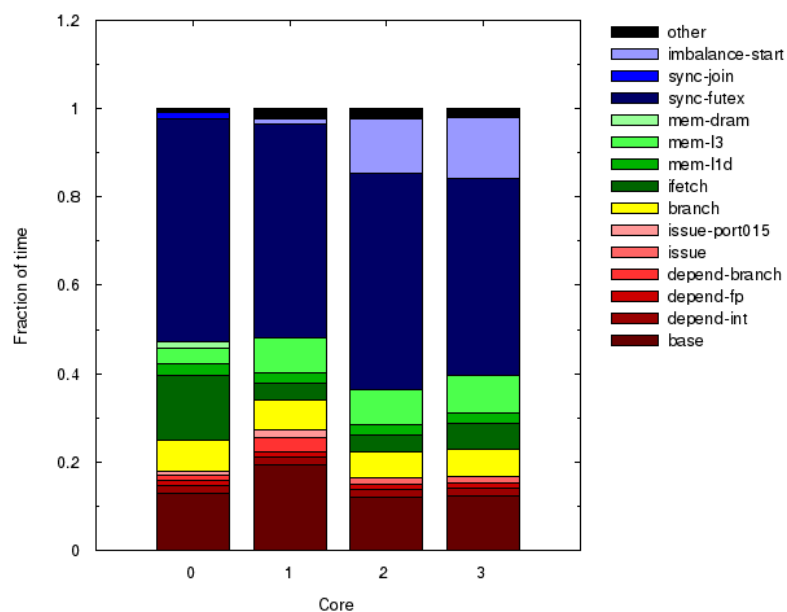
- 進入 sniper 後，我們要先準備好 sniper 的分析工具
 1. 解壓縮 pin-2.13-61206-gcc.4.4.7-linux.tar.gz 和 boost_1_65_1.7z
 2. 將這兩個檔案移動到路徑 Downloads/sniper-6.0/
- 接下來，更新 sniper 內的套件
 1. `$ sudo apt-get update` (更新套件)
 2. `$ sudo apt-get -y dist-upgrade` (升級套件)
 3. `$ sudo apt-get clean` (清理套件)
- 安裝 sniper
 1. `$ make clean`
 2. `$ make -j 4`
 3. `$ cd test/fft`
 4. `$ make run` (測試安裝是否有成功)
- 安裝 benchmarks (sniper 的測試工具)
 1. 解壓縮 sniper-benchmarks.tbz 並放到路徑 Downloads/sniper-6.0/
 2. `$ cd benchmarks/`
 3. `$ export GRAPHITE_ROOT=/home/sniper/Downloads/sniper-6.0`
 4. `$ export BENCHMARKS_ROOT=$(pwd)`
 5. `$ make` (安裝)

- 執行 sniper

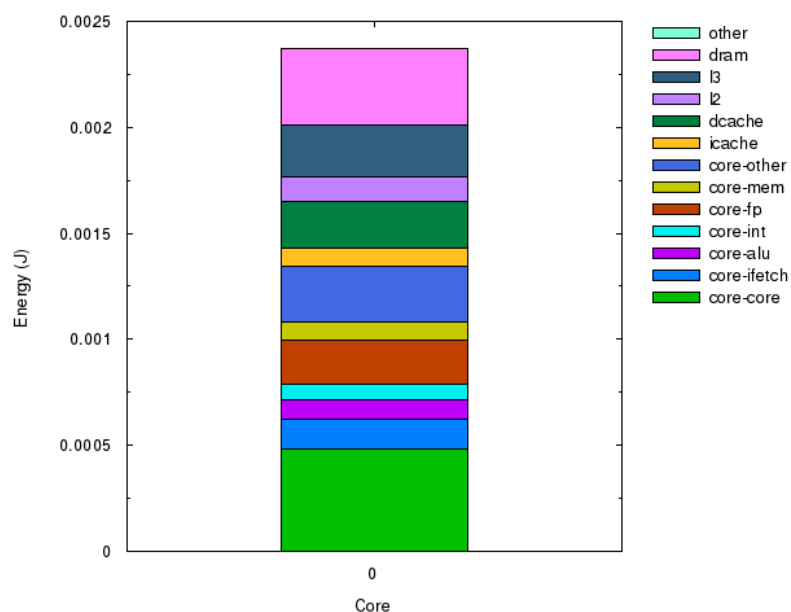
1. 確認目前路徑為 `~Downloads/sniper-6.0/benchmarks`
2. `$./run-sniper -p splash2-fft -i test -n 2 -c gainestown` (跑 sniper)
-p: suite name-benchmark name
-n: numcores
-i: inputs size
-c: passed to sniper/run-sniper unmodified
3. 結果會在路徑 `~Downloads/sniper-6.0/benchmarks` 內，檔名為 `sim.out`

- 執行 `cpistack.py`、`mcpat.py` 和 `gen_topology.py` 得到結果

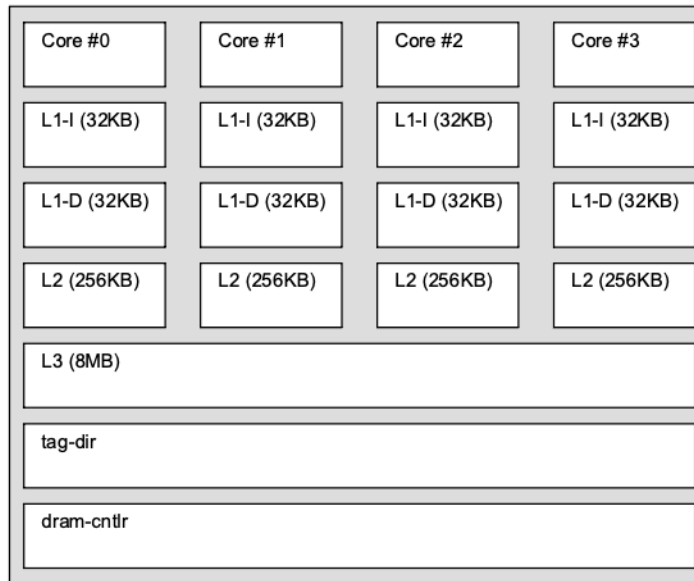
1. \$../tools/cpistack.py (量測各個部分執行時間的比例)



2. \$../tools/mcpat.py (量測能耗)



3. \$../tools/gen_topology.py (量測記憶體階層拓樸)



Problem 1

此問題主要是要讓我們練習透過更改 command line 的 -n 參數來改變 core 數，並得到不同 core 的 instructions、cycles 和 CPI，加上分析能量消耗最高的 3 個 components

- \$./run-sniper -p splash2-fft -i test -n 4 -c gainestown (4 cores 的結果)
- \$./run-sniper -p splash2-fft -i test -n 8 -c gainestown (8 cores 的結果)
- 透過 \$../tools/cpistack.py、\$../tools/mcpat.py 和 \$../tools/gen_topology.py 得到三張執行結果的圖

分析結果：

在 sim.out 檔案中包含了 instructions、cycles 和 IPC 等資訊，其中 CPI 又等於 cycle / instruction。我們可以觀察到當 core 數愈多，總指令會變多，而每個 core 所分擔的指令也會隨著 core 數的增加而減少。總 cycle 數會隨著 core 的增加而上升，因此 CPI 也會隨著提高。

最高能耗的部分，可以發現在 core4 和 core8 中 dram 的能耗分別佔了第二和第一名，屬於非常耗能的部分。而 l3 也都是前幾耗能的部分。

Problem 2

此問題主要是要讓我們去更改 sniper 內的配置檔案

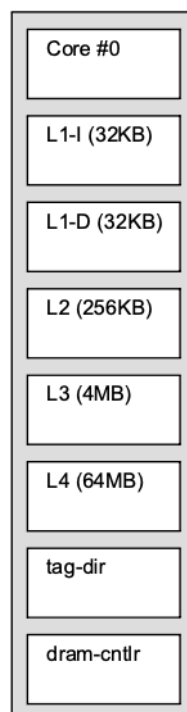
(~Downloads/sniper-6.0/config/gainestown.cfg 和

~/Downloads/sniper-6.0/config/nehalem.cfg)以變更 cache size，藉此分析不同 core 不同 cache size 的效能差異。

Core1 + 4MB L3 cache	Core2 + 4MB L3 cache
Core1 + 8MB L3 cache	Core2 + 8MB L3 cache
Core1 + 16MB L3 cache	Core2 + 16MB L3 cache

分析結果：Layer 3 cache 的大小並不會對 CPI 造成顯著的影響。而 core 數從 1 變成 2 使得 instructions 平均交給兩個 cores 處理，但效率卻沒有到提升兩倍，這是 CPU cores 的性能瓶頸。而變成 2 cores 後，單一 core 的 CPI 下降，miss rate 上升，這是因為 core 數變多，使得耗能增加。

最後還有透過更改配置檔案模擬了 4 層(新增 L4 cache)的記憶體階層架構



Project 2

Introduction

這個 project 主要是要讓我們在 sniper 分析單執行緒和多執行緒的效能，以及透過 hotspot 來分析系統的溫度。

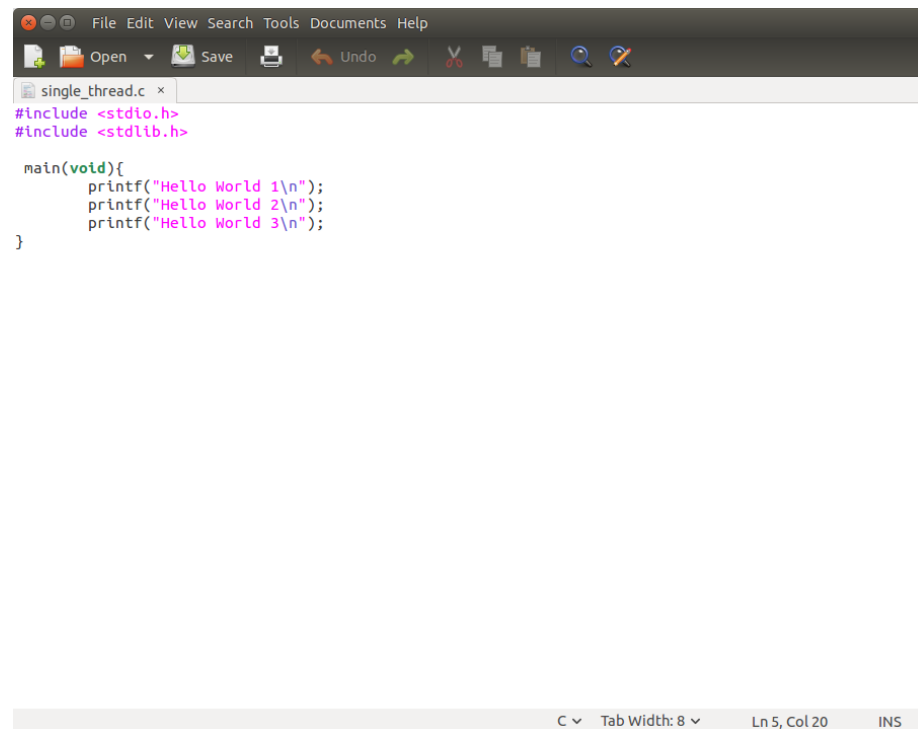
Single and multiple thread cross compiler check

這個部分我們要測試單執行緒和多執行緒是否能跨編譯器正確執行。

- 測試單執行緒編譯
 1. 在~Downloads/sniper-6.0/project2 下編譯 hello.c (`$ gcc hello.c -o hello`)
 2. 在~Downloads/sniper-6.0 下透過 sniper 執行剛剛編譯的執行檔 (`./run-sniper ./project2/hello`)
- 測試多執行緒編譯
 1. 在~Downloads/sniper-6.0/project2 下編譯 hello_thread.c (`$ gcc hello_thread.c -o hello_thread -pthread`)
 2. 在~Downloads/sniper-6.0 下透過 sniper 執行剛剛編譯的執行檔 (`./run-sniper -n 4 ./project2/hello_thread`)

Problem 1

Single thread C code

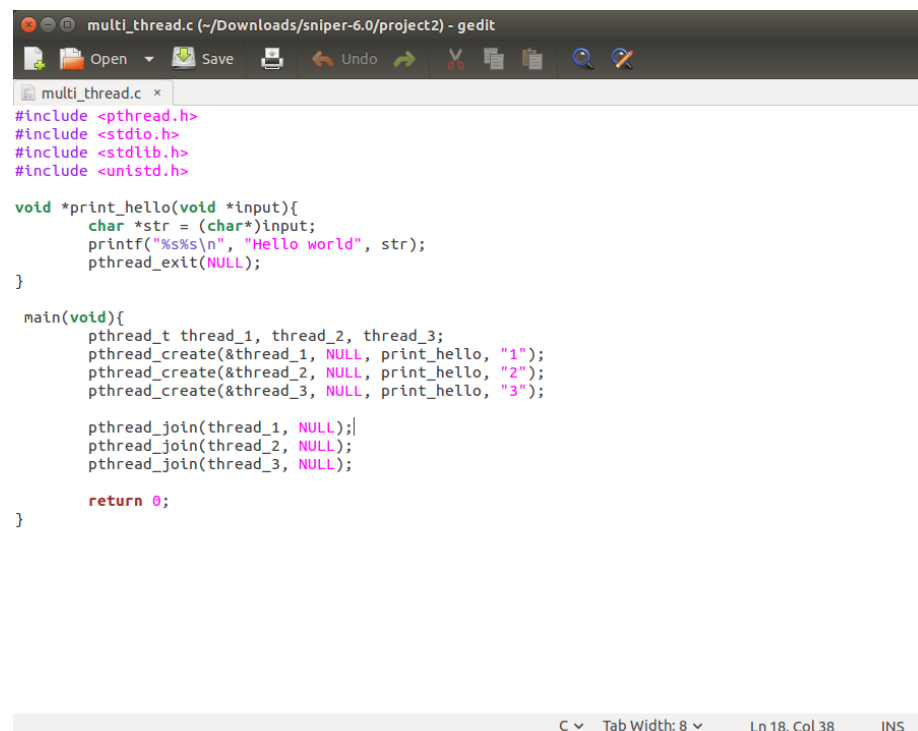
A screenshot of a code editor window titled 'single_thread.c'. The editor has a menu bar with 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. Below the menu is a toolbar with icons for opening, saving, printing, undo, redo, cut, copy, paste, find, and replace. The code is as follows:

```
#include <stdio.h>
#include <stdlib.h>

main(void){
    printf("Hello World 1\n");
    printf("Hello World 2\n");
    printf("Hello World 3\n");
}
```

The status bar at the bottom shows 'C', 'Tab Width: 8', 'Ln 5, Col 20', and 'INS'.

Multiple thread C code

A screenshot of a code editor window titled 'multi_thread.c (~/.Downloads/sniper-6.0/project2) - gedit'. The editor has a menu bar with 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. Below the menu is a toolbar with icons for opening, saving, printing, undo, redo, cut, copy, paste, find, and replace. The code is as follows:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *print_hello(void *input){
    char *str = (char*)input;
    printf("%s\n", "Hello world", str);
    pthread_exit(NULL);
}

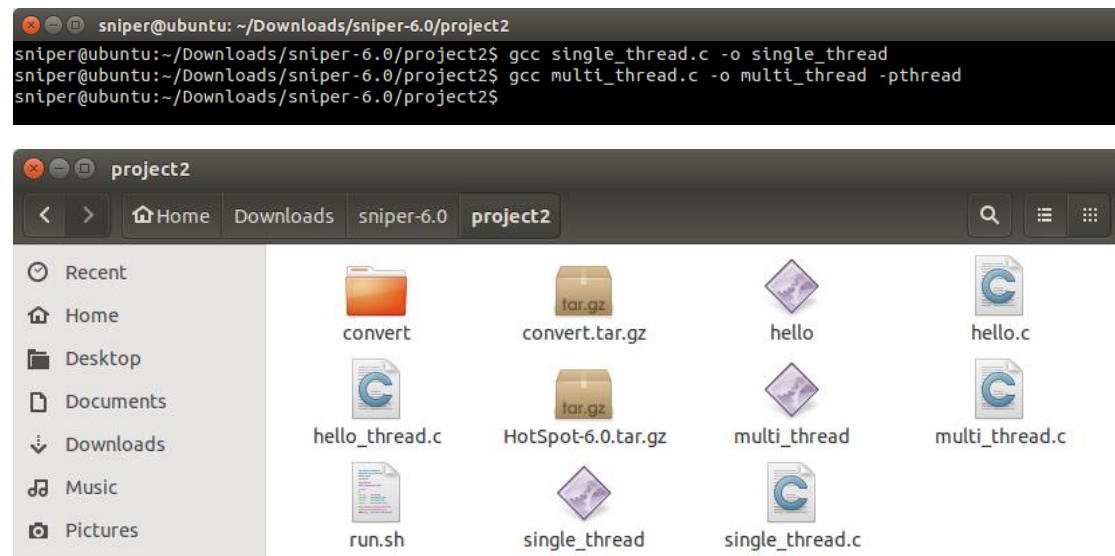
main(void){
    pthread_t thread_1, thread_2, thread_3;
    pthread_create(&thread_1, NULL, print_hello, "1");
    pthread_create(&thread_2, NULL, print_hello, "2");
    pthread_create(&thread_3, NULL, print_hello, "3");

    pthread_join(thread_1, NULL);|
    pthread_join(thread_2, NULL);
    pthread_join(thread_3, NULL);

    return 0;
}
```

The status bar at the bottom shows 'C', 'Tab Width: 8', 'Ln 18, Col 38', and 'INS'.

能夠正確編譯



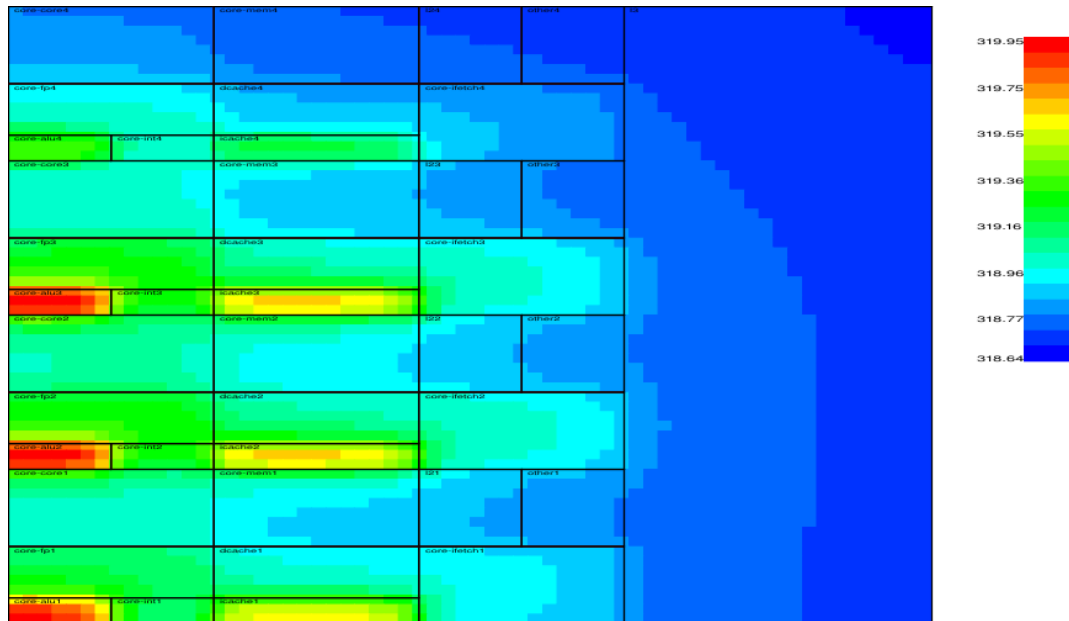
分析結果：

以跑上面 C code 的結果來說，因為 Multi-thread 的 program 需處理的指令數較多，導致 Total Instruction、Total Cycle 數量明顯增多。計算下來，單執行緒的平均 CPI 較低，多執行緒則因為工作大部分都集中在一個 core 執行，造成工作量分配不均，導致平均 CPI 增加。

Problem 2

這個部分是要透過 hotspot 去量測溫度。

- 首先解壓縮 hotspot
 1. \$ cd ~/Downloads/Hotspot-6.0
 2. ~/Downloads/Hotspot-6.0 \$ make
- 到 benchmark 資料夾中透過 McPAT 產生可視化的逐步可視圖
\$ /run-sniper -p splash2-fft -d ../../dataoutput-sniper/ -n 4 -c gainestown --viz --power
- 最後執行 script 得到 power trace 檔案(png、svg、gif)



分析結果：從圖中可以發現溫度最高的地方都集中在 **core-ALU** 的部分。**ALU** 為算術邏輯單元 (arithmetic logic unit)，是電腦不可或缺的部分，當指令載入指令記憶體後，會被送往暫存器，接著就會由 **ALU** 進行指令的運算，無論是算數、儲存、載入或跳躍等，都需要 **ALU** 做加減乘除的運算，他的負擔算是 **data flow components** 內負擔較大的，因此，比起 **data cache**、**core-ifetch** 等，**ALU** 是相對忙碌的，溫度也是 **components** 裡面最高的，達到了 319.95 度。

Project 3

Introduction

這個 project 是要讓我們更進一步的分析不同數量的 cores 和不同的 cache (階層、size、data access time 或 tag access time 等)所帶來的相異效能。

Benchmark Setting

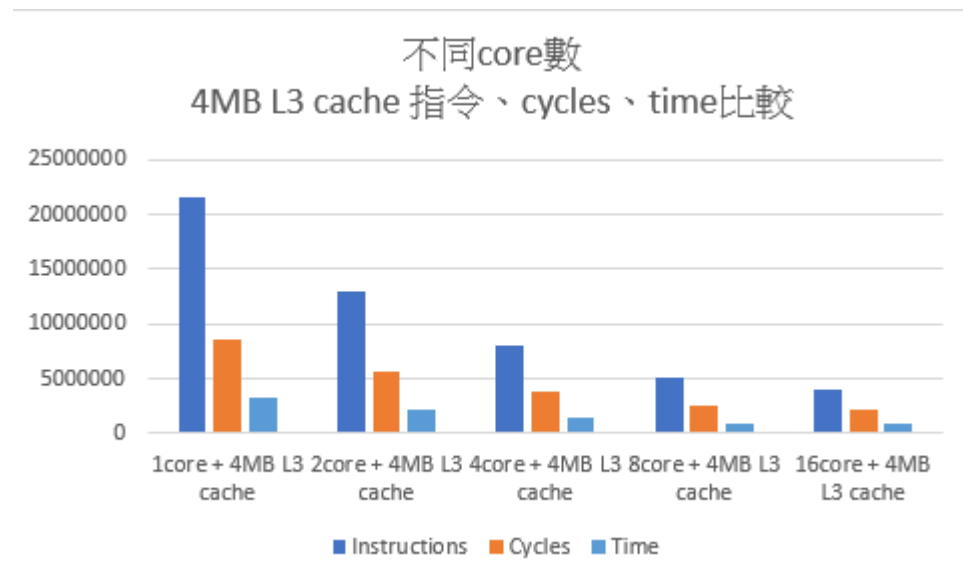
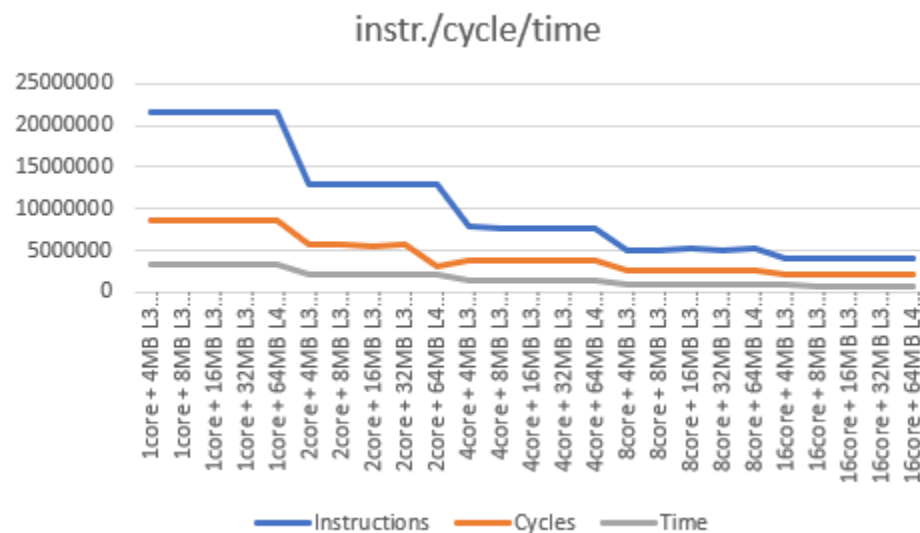
- 藉由更改~/Downloads/sniper-6.0/config/gainestown.cfg 的配置來改變 cache size 和有幾層 cache 等
- 本次使用了 Cholesky benchmark 來量測
- `$/run-sniper -p splash2-cholesky -i test -n 2 -c gainestown`

Project Problem

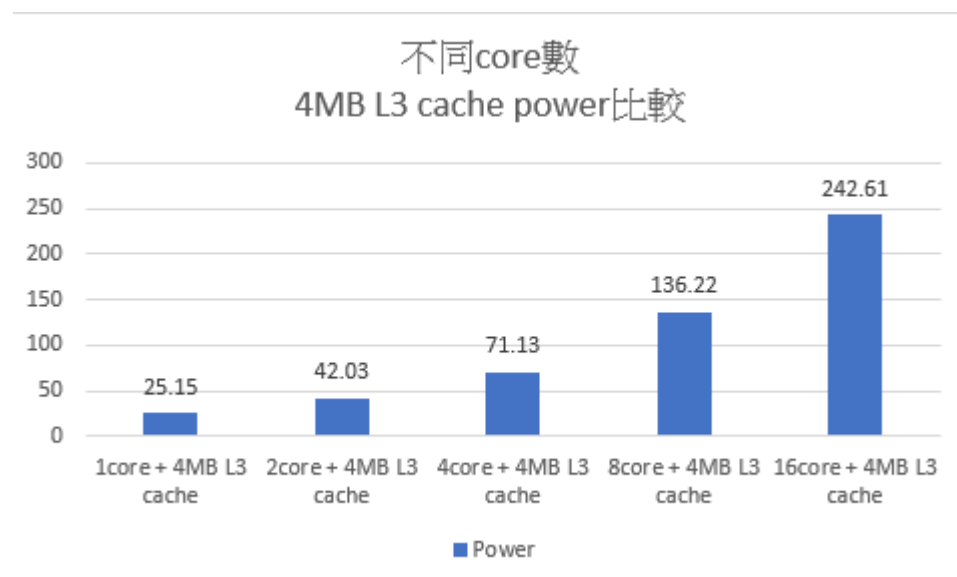
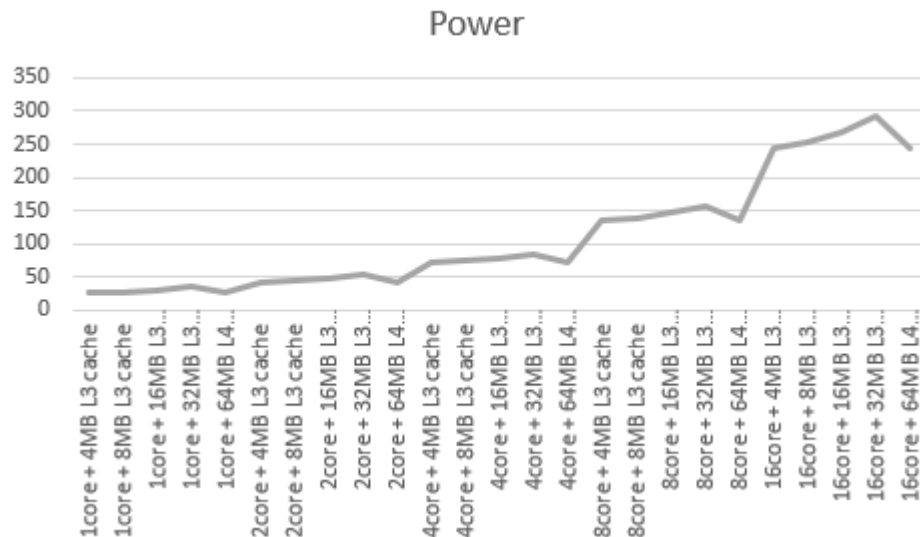
- 本次模擬了只有 L3 cache 的情況(size 分別為 4MB、8MB、16MB 和 32MB) 以及新增 L4 cache 的記憶體階層(4MB L3 cache + 64MB L4 cache)，然後再分別用 1、2、4、8、16 個 core 去跑模擬。

規格	core	cache size	Instructions	Cycles	IPC	CPI	Time	Power	Energy	Average Latency
1core + 4MB L3 cache	1	4MB L3	21515134	8636194	2.49	0.401401	3246690	25.15	0.08	65.35
2core + 4MB L3 cache	2	4MB L3	12922613.5	5621992	2.3	0.435055	2113531	42.03	0.09	56.43
4core + 4MB L3 cache	4	4MB L3	7950499.75	3881581	2.0475	0.489174	1459242	71.13	0.1	14537
8core + 4MB L3 cache	8	4MB L3	5110854.75	2542548	2.00875	0.499192	955845.6	136.22	0.13	64.09
16core + 4MB L3 cache	16	4MB L3	3971215.25	2149952	1.846875	0.542259	820752.7	242.61	0.2	132.91
1core + 8MB L3 cache	1	8MB L3	21511695	8642731	2.49	0.401769	3249147	26.78	0.09	65.35
2core + 8MB L3 cache	2	8MB L3	12914777.5	5626091	2.295	0.435635	2115072	43.69	0.09	56.73
4core + 8MB L3 cache	4	8MB L3	7745011.75	3756889	2.0575	0.485549	1412365	73.5	0.1	12598.04
8core + 8MB L3 cache	8	8MB L3	5100423.25	2572747	1.98375	0.506134	967198.4	138.46	0.13	59.5
16core + 8MB L3 cache	16	8MB L3	3981593.125	2110152	1.885625	0.531044	793290.6	252.31	0.2	65.31
1core + 16MB L3 cache	1	16MB L3	21516534	8646661	2.49	0.401861	3250625	29.72	0.1	65.33
2core + 16MB L3 cache	2	16MB L3	12916231	5607119	2.305	0.434118	2107940	46.8	0.1	58.74
4core + 16MB L3 cache	4	16MB L3	7747256.75	3764840	2.0575	0.486459	1415354	76.55	0.11	12436.29
8core + 16MB L3 cache	8	16MB L3	5130183.625	2551229	2.01	0.498977	959108.8	146.1	0.14	62.79
16core + 16MB L3 cache	16	16MB L3	3990906.438	2089320	1.91	0.524558	785459	266.95	0.21	65.96
1core + 32MB L3 cache	1	32MB L3	21521742	8652133	2.49	0.402018	3252682	35.34	0.11	65.36
2core + 32MB L3 cache	2	32MB L3	12950177.5	5629272	2.3	0.4347	2116268	52.46	0.11	60.51
4core + 32MB L3 cache	4	32MB L3	7757947.75	3749588	2.0675	0.483805	1409620	82.69	0.12	12484.09
8core + 32MB L3 cache	8	32MB L3	5117644.5	2565502	1.99625	0.502912	964474.5	157.12	0.15	60.98
16core + 32MB L3 cache	16	32MB L3	3971842.563	2074831	1.9125	0.523659	780011.9	291.08	0.23	63.53
1core + 64MB L4 cache	1	64MB L4	21522599	8652531	2.49	0.402021	3252831	25.13	0.08	65.32
2core + 64MB L4 cache	2	64MB L4	12910062.5	3095404	2.295	0.239313	2115751	41.98	0.09	58.14
4core + 64MB L4 cache	4	64MB L4	7749556	3761912	2.0575	0.485924	1414253	71.68	0.1	12492.95
8core + 64MB L4 cache	8	64MB L4	5172215.875	2573467	2.00875	0.499205	967468.9	135.89	0.13	58.26
16core + 64MB L4 cache	16	64MB L4	3972889.313	2108258	1.885	0.531873	792578.6	245	0.2	62.72

本次模擬結果以 **cores** 的數量為主軸，以指令數量、**cycle** 數和執行時間來看，隨著 **core** 數量的增加，單個 **core** 負責的指令和 **cycle** 較少，而執行時間也較短，這是因為多核的系統能將指令平均分配到不同 **core** 執行，有效的降低了每個 **core** 的工作量以及整體的執行時間，若 **core** 數相同，不同 **cache size** 對於性能的影響並沒有很大。



至於能耗的部分，消耗的能量也會隨著 core 數的增加而上升，也會隨著 cache size 的大小而上升。而我們可以發現折線圖在有 L4 cache 的地方會有些微將低，這是因為是 4MB L3 cache + 64MB L4 cache，能耗還是主要由 L3 cache 的大小來決定。



如果只考慮 time 的話，16cores + 32MB L3 cache 的時間最短，然而，多核且容量較大的配置會帶來較高的能耗(power)。若有 L4 cache，則可以使整體的效能最好，執行時間短且 time/power 低，是為最佳的選項。但如果只有 L3 cache，則多核的選項能帶給我們比較好的整體性能。

