高效能巨量資料與人工智慧系統 HW3

B05611047 化學四 鍾宜樺

1. 調整 NROW,NCOL。調整 thread

(NROW, NCOL = 1000, 2000, 4000；thread = 2, 4, 8, 16；執行時間整理成表格)

表格一、Elapsed time *(Unit : sec)*

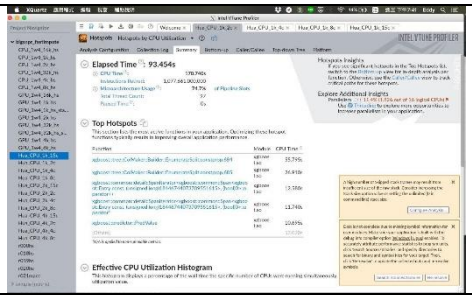
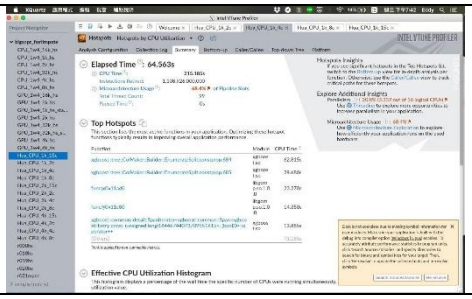
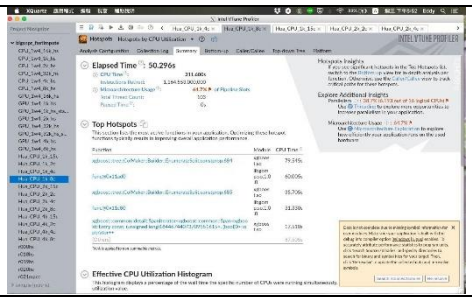
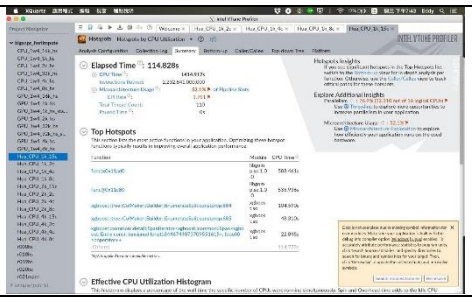
NROW \ cores	2	4	8	15
1000	93.454	64.563	50.296	114.828
2000	326.940	219.801	175.391	338.815
4000	1107.774	790.471	597.319	713.402

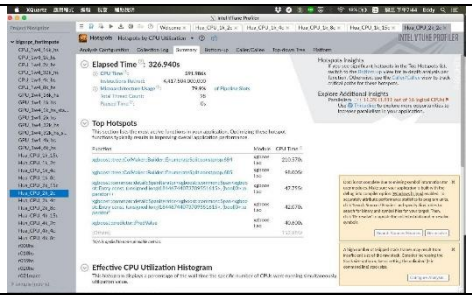
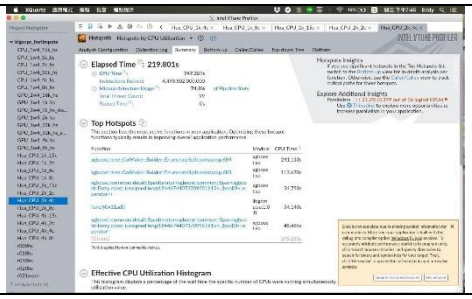
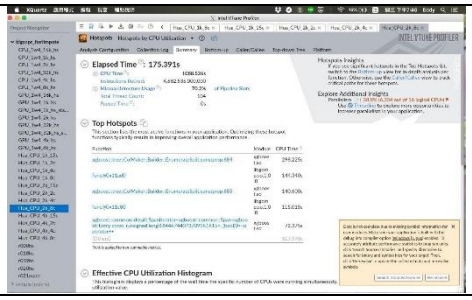
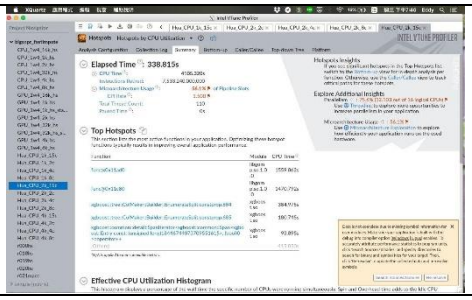
表格二、CPU time *(Unit : sec)*

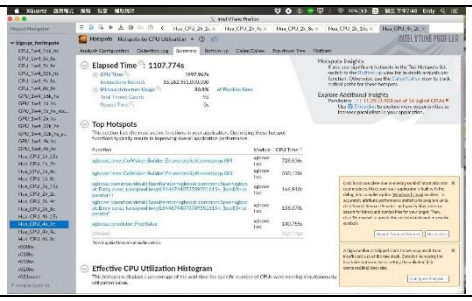
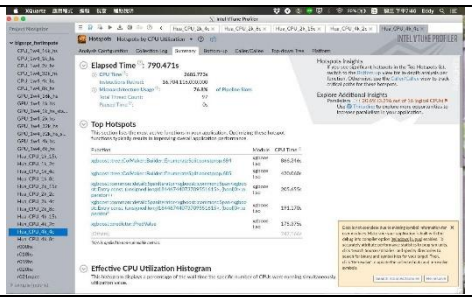
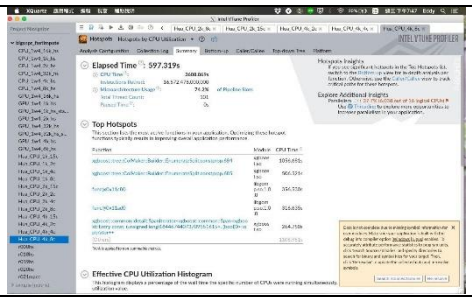
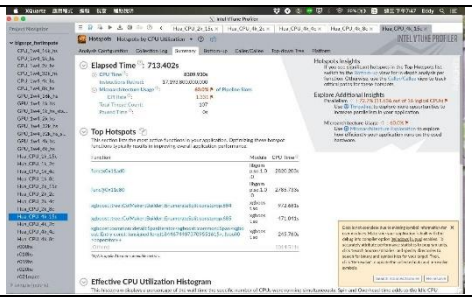
NROW \ cores	2	4	8	15
1000	170.740	215.185	411.600	1414.917
2000	591.986	747.351	1088.526	4105.320
4000	1997.967	2605.773	3608.069	8309.930

2. 找出前 5 耗時的 function (可以使用截圖)，並敘述觀察到什麼。

圖片：

NCOL=NROW=1000, 2 cores	NCOL=NROW=1000, 4 cores
	
NCOL=NROW=1000, 8 cores	NCOL=NROW=1000, 15 cores
	

NCOL=NROW=2000, 2 cores	NCOL=NROW=2000, 4 cores
	
NCOL=NROW=2000, 8 cores	NCOL=NROW=2000, 15 cores
	

NCOL=NROW=4000, 2 cores	NCOL=NROW=4000, 4 cores
	
NCOL=NROW=4000, 8 cores	NCOL=NROW=4000, 15 cores
	

表格三、各個 thread 跟 core V.S. Top Hotspots

function idx		1k_2c	1k_4c	1k_8c	1k_15c	2k_2c	2k_4c	2k_8c	2k_15c	4k_2c	4k_4c	4k_8c	4k_15c
A	xgboost::constrop.684	55.795	62.815	79.545	104.59	210.57	241.11	298.225	384.875	728.636	866.346	105.681	972.681
B	xgboost::constrop.685	26.91	29.68	35.705	48.31	98.005	113.67	140.6	180.745	350.31	420.06	506.521	471.041
C	func@0x11ad0		23.27	60.005	588.461			144.34	1559.862			316.635	2820.203
D	func@0x11c80		14.35	31.33	535.936		54.14	115.015	1470.792			356.53	2785.733
E	xgboost::0pertor++	12.58	13.885	17.51	22.845	47.755	54.75	72.375	93.895	166.94	205.655	264.75	245.76
F	xgboost::operator*	11.74				42.07	48.465			158.37	191.17		
G	xgboost::PredValue	10.695				40.6				140.755	175.735		
	總秒數	170.74	215.185	411.6	1414.917	591.986	747.351	1088.526	4105.32	1997.967	2605.773	3608.069	8309.93

表格四、將時間除以總 CPU time 取得相對消耗的時間%數，將其相加後得 Sigma，並根據此加總作為找出前 5 耗時的 function 的排序根據，結果如下：

function idx		1k_2c	1k_4c	1k_8c	1k_15c	2k_2c	2k_4c	2k_8c	2k_15c	4k_2c	4k_4c	4k_8c	4k_15c	Sigma	Rank
A	xgboost::constrop.684	0.326783	0.291912	0.193258	0.07392	0.355701	0.322619	0.273971	0.09375	0.364689	0.332472	0.02929	0.11705	2.775416	1
B	xgboost::constrop.685	0.157608	0.137928	0.086747	0.034143	0.165553	0.152097	0.129165	0.044027	0.175333	0.161204	0.140386	0.056684	1.440875	4
C	func@0x11ad0		0.10814	0.145785	0.415898			0.132601	0.379961			0.087757	0.339377	1.60952	2
D	func@0x11c80		0.066687	0.076118	0.378776		0.072443	0.105661	0.358265			0.098815	0.335229	1.491993	3
E	xgboost::0pertor++	0.073679	0.064526	0.042541	0.016146	0.080669	0.073259	0.066489	0.022872	0.083555	0.078923	0.073377	0.029574	0.70561	
F	xgboost::operator*	1.097709				0.071066	0.064849			0.079266	0.073364			1.386254	5
G	xgboost::PredValue	0.062639				0.068583				0.070449	0.067441			0.269112	

經由上面統計，經過排序最耗時的是 functions 大概有：

1. xgboost::tree::ColMaker::Builder::EnumerateSplit.constprop.684
2. func@0x11ad0
3. func@0x11c80
4. xgboost::tree::ColMaker::Builder::EnumerateSplit.constprop.685
5. xgboost::common:detail::SpanIterator<xgboost::common::Span<xgboost::Entry const,(unsigned long)18446744073709551615>,(bool)()>::operator*

B. 觀察到什麼：

觀察一、我們可以從表四發現，在 core 數比較少的時候 function F 和 function G 的耗時比率會相對高。然而，隨著 cores 數的上升他們的占比卻下降了。取而代之的是 function C 跟 function D 的占比逐漸上升。我們覺得這個原因可能是因為 function F 和 function G 是平行度較小的 function，因此在 cores 較小的時候站比較高，故當 cores 數上升後，系統則換成使用平行度較高的 functions 分別為 function F 和 function G。另外還有 function A 跟 function B，在固定的矩陣大小下，他們的時間占比都會隨著 core 數上升而下降，原因可能是這兩種 function 在相同矩陣大小下的工作都是相同的。因此當 core 數上升時，這個固定大小的工作量就會被分配給很多 core 去執行，故當 core 數上升時，他們的時間占比會下降。

觀察二、我們可以從上表四中發現，矩陣大小在相同的 core 數下時，並不會影響各 function 的時間占比。像是看 1k_2c、2k_2c、4k_2c 的各個 function 的時間占比，我們就很清楚可以發現，這個小觀察。

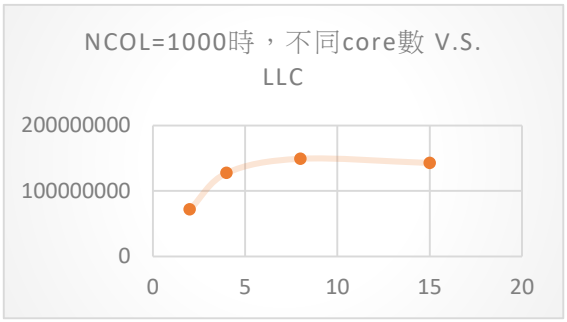
3. Memory access (LLC)

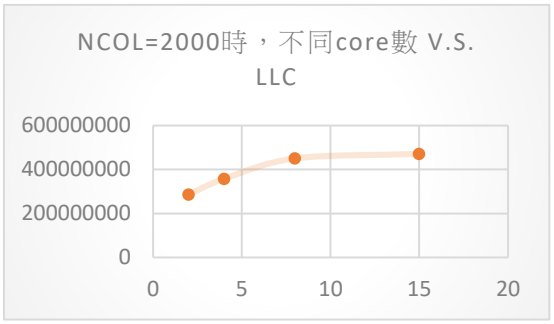
(不同大小的 matrix 和不同的 thread 數量(同上)、將不同 matrix shape 的 LLC miss count 整理成表格。敘述觀察到什麼。)

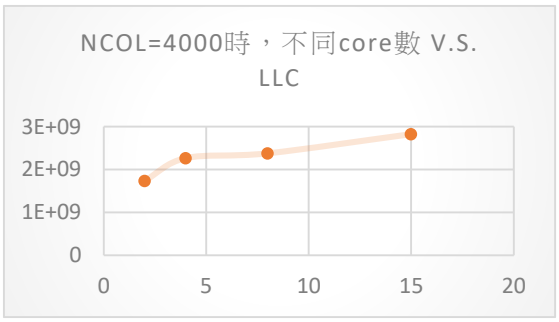
A. 表格二、LCC miss count

NROW \ cores	2	4	8	15
1000	71658020	127413938	149155999	142731897
2000	286185778	356993402	449779012	470435329
4000	1734238198	2262959104	2375723042	2823202617

B. 觀察到什麼：



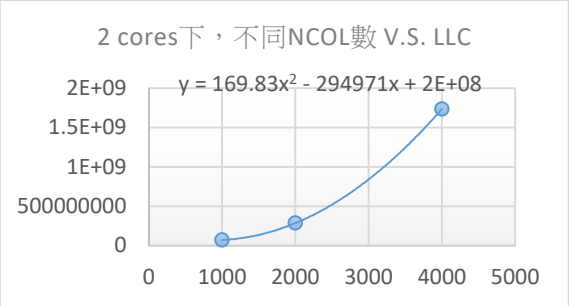


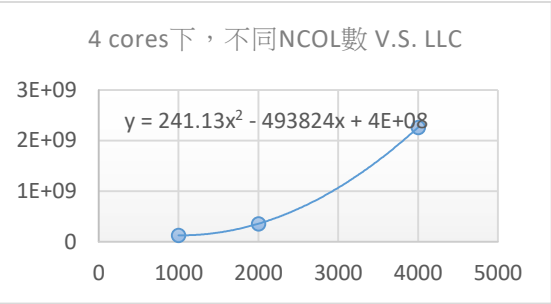


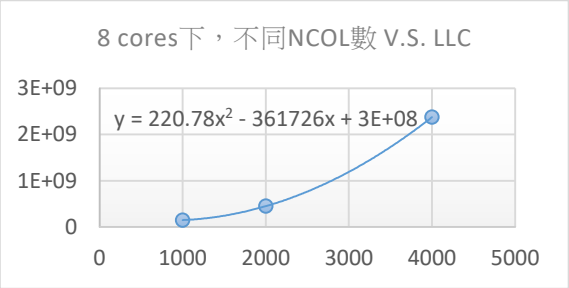
觀察一、不同的 thread 數會不會對 LLC 造成影響

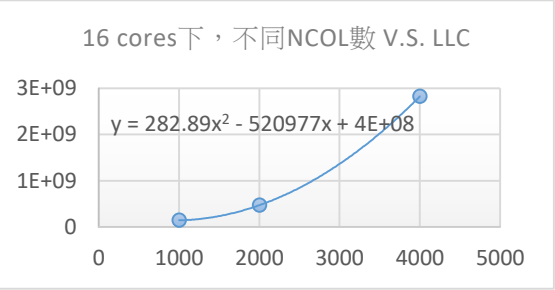
從上面三個表中，我們可以發現到在相同的矩陣大小下，隨著 threads 數的上升 LLC 也大致上也會跟著上升。原因可能是因為，當 core 數上升的時候，每個 core 都有 cache miss 的時候，總 LLC 的數量就會隨著 core 數上升而上升，故會有表中的效果。

另外我們可以發現，cores 數上升到一定程度的時候，趨勢線都會趨於平緩。我們猜測原因是因為 blocking 的關係。Blocking 在做的事情就是將總工作量切成較小的份量分配給每個 thread 去執行，且希望這個切出來的 block 的大小能夠盡量符合 cache size，以達到減少 cache miss 的效果，提升 code 的效能，達到 optimize 的效果。因此從上三表中我們可以發現，當切的數量到達一定的點的時候，cache miss 上升的速率就會趨於平緩，也就是如圖表中的結果，當 thread 數到達一定的數量後，隨著 thread 數的上升，LLC 的數量會趨於平緩。









觀察二、不同的矩陣大小，會不會對 cache miss 造成影響

從上面四個表中很明顯的可以發現到，隨著矩陣大小的上升，LLC 的數量也隨之上升。且從四條趨勢線的公式可發現 LLC 大概和矩陣大小成正比，即當 NCOL 從 1000->2000 時，LLC 變大約四倍。原因可能是因為 cache miss 本來就跟資料量呈正比的關係。

觀察三、為什麼要觀察 last level cache miss

LLC 的成本是所有的 cache miss 裡面最高的，所以他的數量有可能大大的影響效能。尤其是當程式有大樣的矩陣運算時。承上面觀察二，矩陣的數量和 LLC 成正比關係，所以當矩陣越大的時候，LLC miss 就會越多，因此在需要大量運算的狀況下，像是大數據的處理的時候，觀察 LLC miss 就會是很重要的評估效能的一個重點。