Parallel Programming

Homework 4 Report

Student ID: 105062635
Name: 吳浩寧

I. DESIGN

1.) Single-GPU

採用老師上課中所介紹的 Block Algorithms，因 Floyd-Warshall 問題使用 adjacency matrix 來儲存距離資訊，因此我們可以將矩陣切割成數個較小的 blocks，如此就能將計算所需資訊都放到 GPU 的 shared memory 中，藉此提升資料利用率，並降低 CGMA ratio。

首先，我將 Block Algorithms 切出的每個 block，對應到 GPU 的一個 block 進行計算；並且將計算的三個階段：Self-dependent blocks、Pivot-row and pivot-column blocks、Other blocks，分成三個 kernel functions，因為每階段的計算都需要用到前一階段的資料，因此 blocks 之間須要藉此才能同步。

由於主要的計算量是在第三個階段，因此這次的改良方式以處理這一階段為主。此階段需要自己這個 block，還有上階段 pivot row 和 pivot column 各一個 block，因此共需 3 個 blocks，這次使用的機器 K20 與 M2090 的 shared memory 大小皆為 48kB，一個 int 為 4 bytes，因此若想減少 global memory 的存取次數，最大的 block 邊長可以使用$\sqrt{48 \times 1024/(3 \times 4)} = 64$，因此當輸入的 blocking factor≦64，我皆會先將資料從 global memory 複製到 shared memory 再計算，當 blocking factor>64，則不作處理直接用 global memory 進行計算。

不過一個 block 中最多只有 1024 個 threads，因此每個 thread 必須計算多個位置，我的分配方式是由一個 block 的左上角開始，向右向下分給每個 thread，一開始我將 thread 位置更新的計算，放到每個位置更新完距離的地方，不過發現如此產生的 overhead 非常大，由於當邊長≦64 時每個 thread 最多只會負責 4 個位置，因此一開始就預先計算每個位置，並宣告 int x1~x4，y1~y4，存到每個 thread 限量 63 個的 register 來儲存，便能大幅提升速度。

最後，為了降低從 host 複製資料到 device 的時間，使用 pinned memory 來配置 host 端的記憶體，最後一輪第二階段計算完後，便能先用一個 stream 來傳計算完的最後一列，再將第三階段的 kernel function 切割成數部分，新增 block offset 參數來決定要從第幾列開始算起，將整個矩陣分給最多 16 個 streams 來處理，如此先完成的部分便能提早進行資料傳輸，並將計算與傳輸部分重疊。

2.) Multi-GPU implementation with OpenMP

此版本須使用 2 個 threads 來操控 2 片 GPU，由於前 2 階段使用時間較少，因此各個 thread 都須自行計算，避免不必要的傳輸，第三階段則將矩陣分為上下兩半分給 2 個 threads，再用用 cudaMemcpyPeerAsync 來交換 GPU 計算的結果，由於 GPU 間的資料傳輸 overhead 很大，須想辦法用 pipeline 的方式，利用 stream 重疊計算與傳輸的時間，方法與第一個版本差不多，

3.) Multi-GPU implementation with MPI

此版本與 OpenMP 的差異，首先由於這次使用的 GPU 版本似乎不支援 GPUDirect，因此無法將 device address 直接當作參數傳入 MPI function，而得先複製到 host memory 再呼叫，因而須將計算、H2D、D2H data transfer、MPI_Isend、MPI_Irecv 皆加入 pipeline，來掩蓋傳輸更大的 overhead。

而我呼叫的順序如下 MPI_Irecv→FW3→D2H→cudaStreamQuery→MPI_Isend→MPI_Waitany→H2D，除了 cudaStreamQuery 與 MPI_Waitany 外，每個 function 都是 non-blocking 的。每個 process 從一開始便等待訊息，並開始執行計算，由於 FW3、D2H 使用同一個 stream，因此 D2H 會等待 FW3 作完才開始。接著使用 cudaStreamQuery 進行 busy-waiting，等待任一個 stream 完成工作後，便會呼叫 Isend 傳送該部分的結果，接著 CPU 透過 MPI_Waitany 檢查一開始的 MPI_Irecv 狀態，並等待直到接收到一份資料，使用此時閒置的 stream 來執行 H2D，將資料寫回 device memory 中。

2. PERFORMANCE ANALYSIS

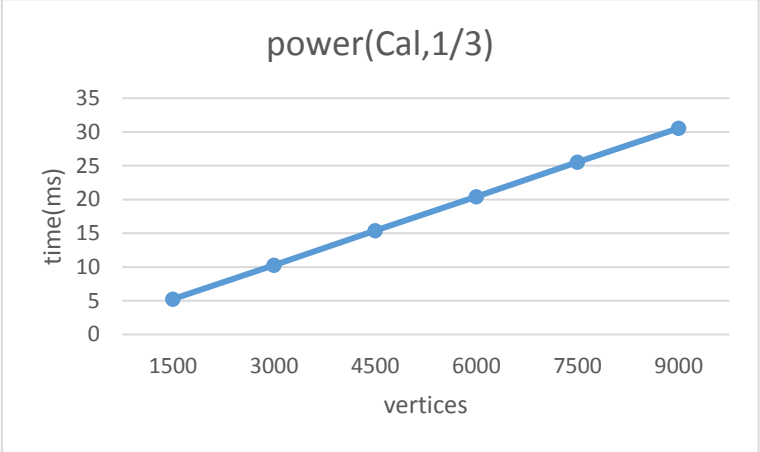
Weak Scalibility

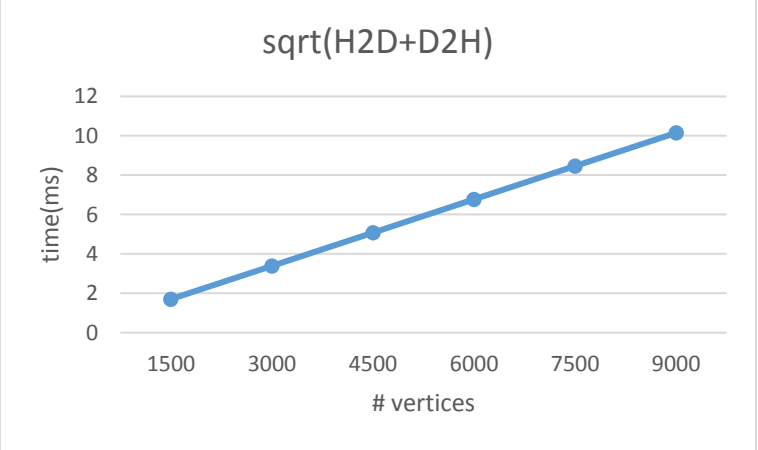
使用 HW3 的./graph_ge 來產生本次所需的 directed graph，使用節點數為 1500*N (N=1~6)，邊的數量平方與節點數成正比，邊上最大的權重為 100。各版本都使用 blocking factor 64，在 M2090 上進行計算。OpenMP 與 MPI 版本皆使用 nodes=1，ppn=2 的設定。

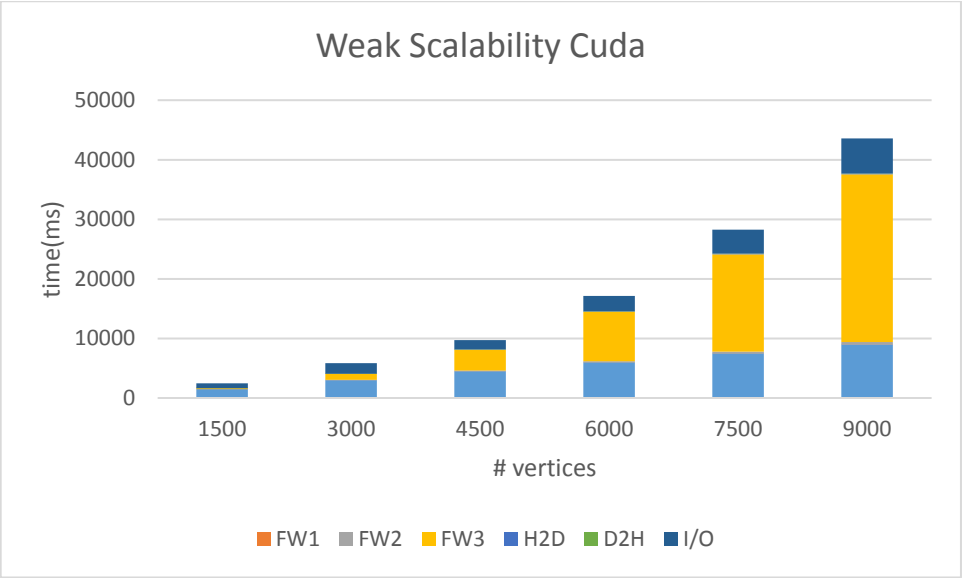
Cuda

	FW1	FW2	FW3	H2D	D2H	I/O	total
1500	3.1357	10.662	130.59	1.4864	1.4177	829.97	977.2618
3000	6.295	42.458	1028.19	5.931	5.5621	1085.286	2873.722
4500	9.4507	95.53	3527.94	13.308	12.5	1587.881	5246.61
6000	12.547	169.15	8304.47	23.667	22.182	2584.734	11116.75
7500	15.77	264.93	16368.9	36.984	34.601	4035.35	20756.54
9000	18.884	380.69	28150	53.169	49.812	5915.813	34568.37

由上圖可以觀察出，由於 Floyd-Warshall 的複雜度為 V 的 3 次方，因此從最主要的計算部分 FW3，可以看出時間符合此原則，佔全部時間的比例也越來越高。由於資料使用 adjacent matrix 來儲存，所需要的空間為 V 的 2 次方，可以看出花在資料傳輸的 H2D、D2H 差不多符合此規則，



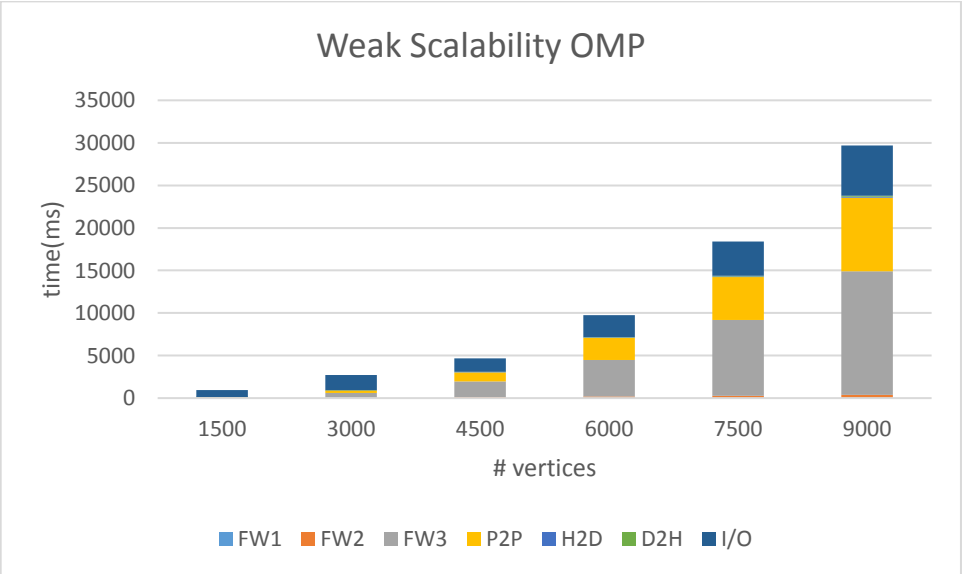




OpenMP

	FW1	FW2	FW3	P2P	H2D	D2H	I/O	total
1500	3.1378	10.672	68.153	40.928	2.8527	2.6864	829.97	958.3999
3000	6.296	42.461	545.36	317.47	11.365	10.74	1085.286	2718.978
4500	9.4484	95.531	1844.96	1077.25	25.514	24.158	1587.881	4664.742
6000	12.583	169.15	4296.39	2587.73	45.383	42.979	2584.734	9738.949
7500	15.768	264.92	8902.22	5055.42	71.035	67.14	4035.35	18411.85
9000	18.885	380.64	14503.3	8667.19	102.07	96.688	5915.813	29684.59

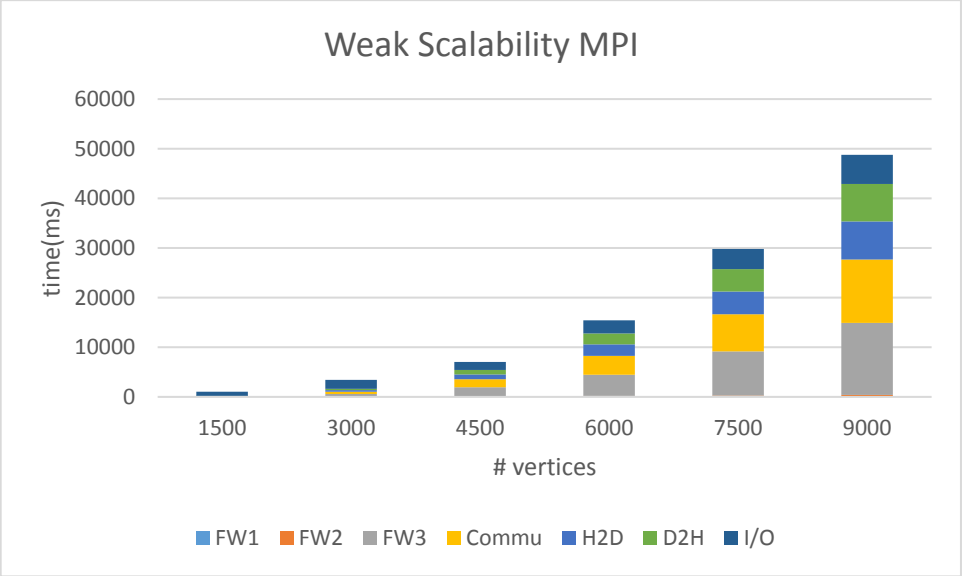
可以看出計算所花的時間快了 cuda 版本差不多一倍，不過多出了 peer to peer 的資料傳輸時間，不過由於成長幅度較緩因此整體速度在資料量變大實也會有更顯著的加速。

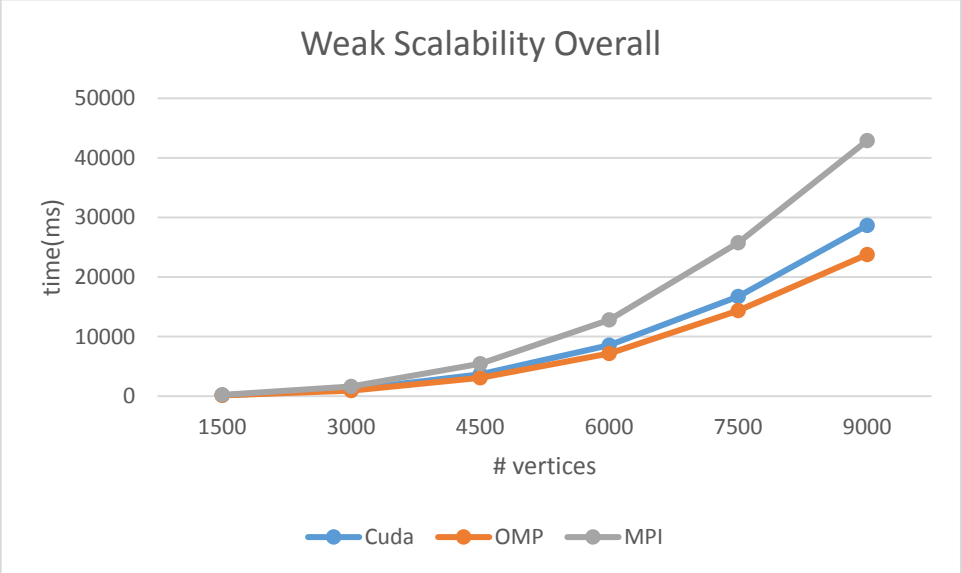


MPI

	FW1	FW2	FW3	Commu	H2D	D2H	I/O	total
1500	3.1366	10.661	68.147	53.744	38.115	35.708	829.97	1039.482
3000	6.2967	42.479	544.98	467.373	286.9	278.61	1085.286	3411.925
4500	9.4611	95.511	1843.97	1595.576	963.95	949.84	1587.881	7046.189
6000	12.585	169.12	4293.96	3803.341	2298.71	2233.43	2584.734	15395.88
7500	15.772	264.88	8896.2	7476.354	4603.85	4488.53	4035.35	29780.94
9000	18.893	380.63	14497.5	12774.25	7667.42	7549.6	5915.813	48804.1

我使用 MPI_Wtime 來計算資料進行傳輸的時間。可以看出 MPI 版本，由於每輪計算都必須進行 H2D 與 D2H 的傳輸，還有 node 之間的傳輸，使得傳輸的 overhead 高出許多，即便計算所花的時間也快了單 GPU 版本一倍，不過整體時間還是稍稍慢了些。

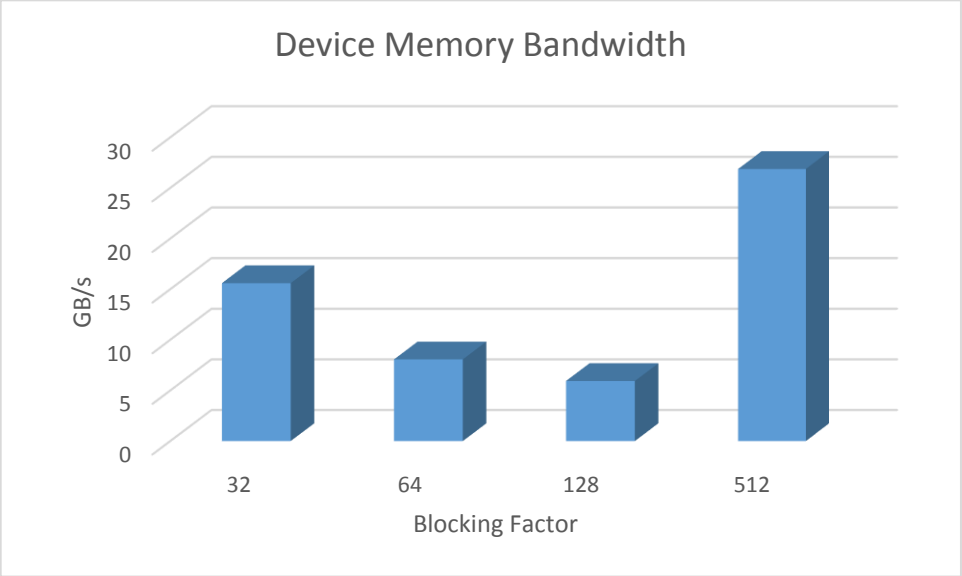


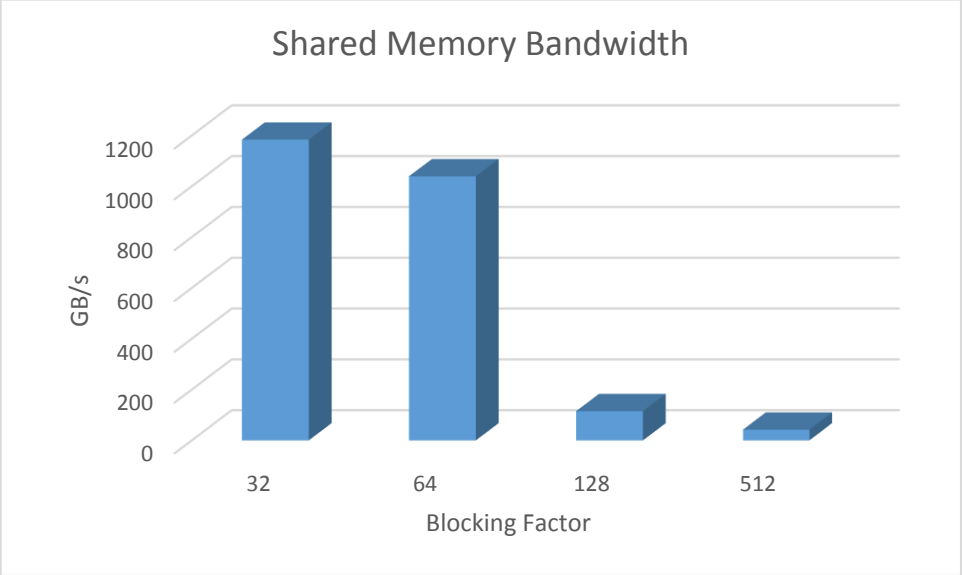


從整體的時間看來，weak scalability 呈區線增長，以 OpenMP 的版本最為理想，MPI 的版本最不理想。

Bandwidth

使用 nvvp 中的 examine individual kernel→perform kernel analysis→perform memory bandwidth analysis 來檢查我第三階段計算使用的 bandwidth，皆使用 testcase/in5 作為測資，使用 omp 版本在兩台 K20 上跑。





由於我在 blocking factor>64 時完全沒有使用到 shared memory，可以看出 shared memory 在此時大幅減少，而雖然 blocking factor=32 時頻寬最大，不過對於同樣 64x64 的 block，複製 global memory 到 shared memory 次數最多達 4 倍，因此我的城是整體效能還是在 blocking factor=64 時最佳。

3. EXPERIENCE

這次作業讓我對 GPU 的硬體架構更加了解，包括 grid、blocks、threads 之間彼此的關係，與如何去操作他們，由於受到數量的限制，因此必須考量如何有效的分配工作；更體會到 GPU 的 memory hierarchy 下，不同類型的記憶體存取速度上的差異，雖然 register、shared memory 的存取速度遠快於 local memory 與 global memory，但空間卻是有限的，因此得決定哪些資料要放在 share memory，才能最有效率地完成計算。

此外，還學習到如何用 asynchronous 的 function call 來達成 pipeline 的效果，藉此提升計算速度，與如何使用一些工具，來得到 GPU 的各項性能資訊、取得程式運行時的各項數據等等，總之很有收穫，對 GPU 算是有更進一步的認識。