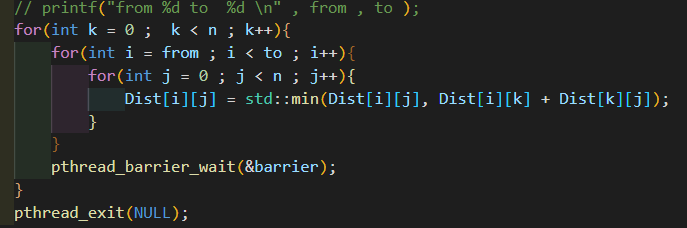
|  |  |  |  |
| --- | --- | --- | --- |
| Hw3 report | | | |
| 學號: 109062233 | 姓名: 蘇裕恆 |  | 繳交時間: |

**Implementation :**

1. Which algorithm do you choose in hw3-1?



在3-1裡面，我使用的是很naïve的FLOYD-WARSHALL algorithm，並使用pthread作為相應的解法。因為主要來說都是floyd – warshall 也比較好判斷。

1. How do you divide your data in hw3-2, hw3-3?

兩種方法都差不多，基本上就是把切成64\*64的方塊並對其執行block-floyd –warshall algorithm.

1. What’s your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor,

#blocks, #threads)

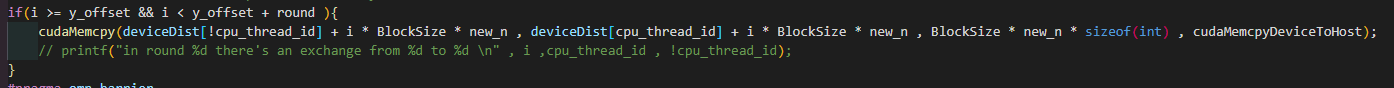
我不知道blocking factor 與blocks的差別，我選擇的是64作為blocksize。

因為gtx 1080的配置，在share memory裡面總共有49152 bytes. 我在phase 3 裡面的實作需要三個share memory block。同時，考慮到warp的性質是以32個資料作為單位，因此考慮block與threads接為32之倍數。

Thread 在HW3-2視為32\*32 ，在HW3-3為 16\*64 ( 原本hw3-3是參考我的第九版code所測計的 因此thread為 16\*64 ( 若變為 32\*32 將有更好的performance 但是因為已經全過了所以就沒有更改。)

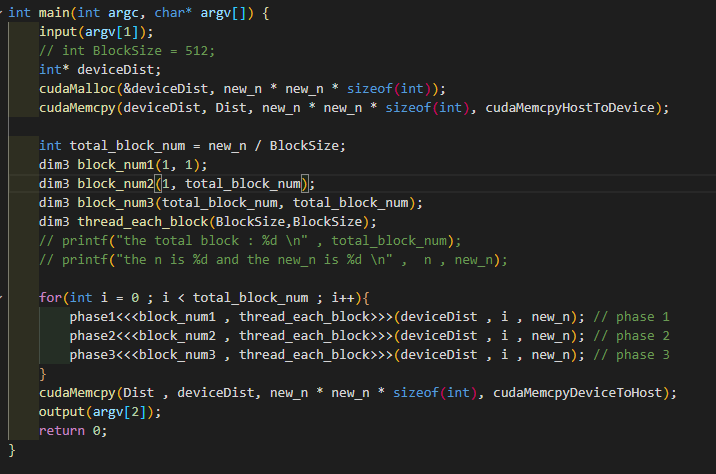
1. How do you implement the communication in hw3-3?

我在hw3-3裡面，使用的是openmp作為base ( 我不是很清楚這題想要問什麼 但如果是問communication function的話我使用的是cudaMemcpy



1. Briefly describe your implementations in diagrams, figures or sentences.

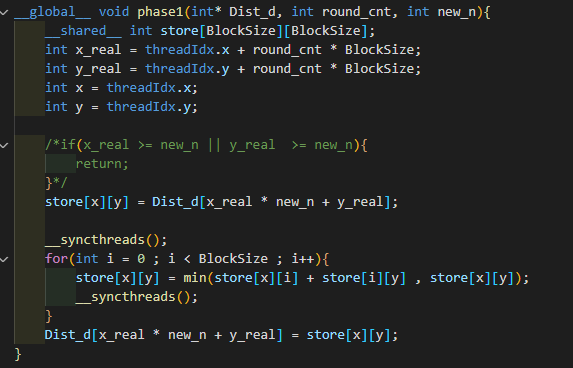
在hw3-2裡面，我總共implement了九個版本。因為後面的版本過於優化以至於基本上很難看懂，因此以下為第一版的code。



總共有三個phase，而一開始時候，會將block設為一個，並執行一開始使的計算

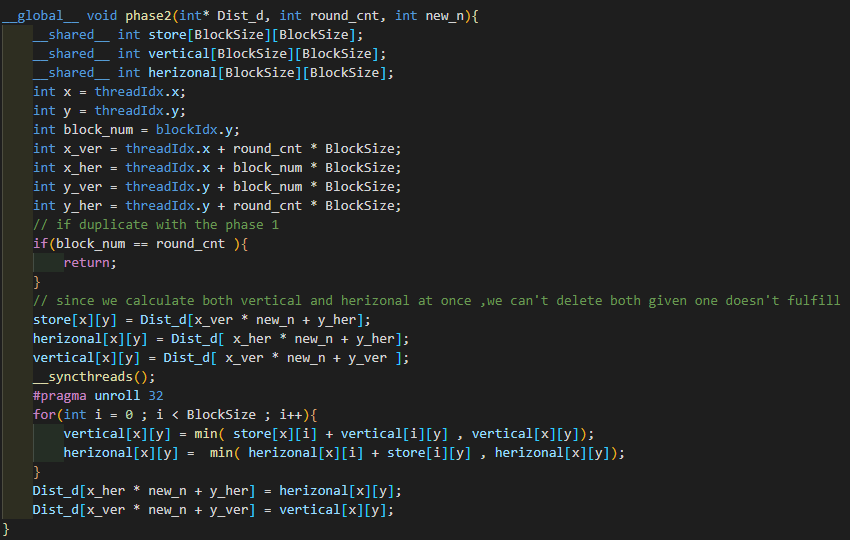


概念上來說，就是將naïve floyd-warshell 在小範圍內實作

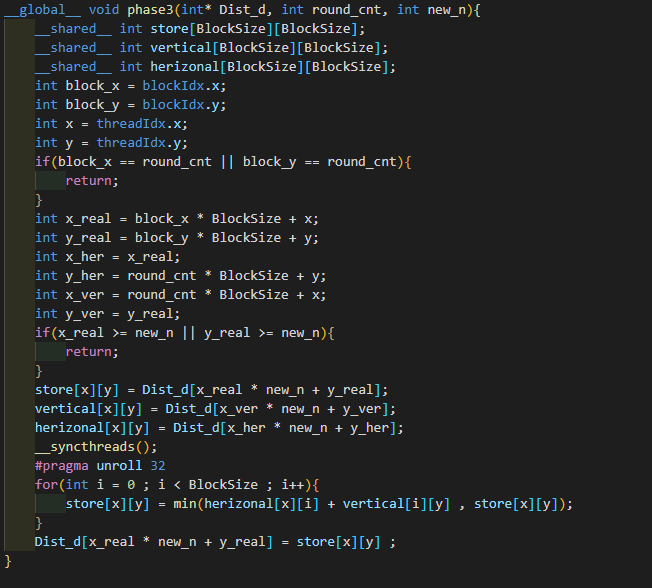


注意在每次run完以後就必須像是hw3-1的實作一樣將thread 同步 (因為data 有dependency)。





在phase二這邊我們使用一個(1, total\_block\_num) 作為block的形狀。同時，我的實作會同時將row跟column算完並回傳。基本上來說vertical就是column。



在phase3這邊，我們將row跟column找出來，並透過spec給的方式做reduction。注意: 此階段與上一階段接在每個block做同步。基本上就是找pivot row跟column來作為base並計算出target最小值。

至於hw3-3，基本上作法與hw3-2類似，唯一的差別在於我將phase 3 做了優化 ( 因為profile以後發現phase3 占了8.90%以上的運行時間，所以基本上優化都是由那邊進行的)

也有一部份是因為不需要過多的優化就可以將hw3-3全部通過，因此並沒有特別在做優化。

優化方法如下 : round為每個gpu需要計算的點數 ( 行數 ) 並且會有一個叫做 y\_offset來說從哪裡開始計算。最後，為了確保正確性，在計算之前都必須將device data做同步化來確保計算實的正確性。



我有不少優化，我將在之後一一闡述。

**Profiling Results (hw3-2)**

Provide the profiling results of following metrics on the biggest kernel of your program

using NVIDIA profiling tools. NVIDIA Profiler Guide.

我使用p22k1作為profiling result的baseline。

|  |  |  |  |
| --- | --- | --- | --- |
|  | Min | Max | Average |
| occupancy | 0.924393 | 0.925843 | 0.94892 |
| sm efficiency | 99.95% | 99.97% | 99.96% |
| shared memory load throughput | 3232.6GB/s | 3307.6GB/s | 3272.7GB/s |
| shared memory store throughput | 263.89GB/s | 270.01GB/s | 267.16GB/s |
| global load throughput | 197.92GB/s | 202.50GB/s | 200.37GB/s |
| global store throughput | 65.972GB/s | 67.502GB/s | 66.789GB/s |

**Experiment & Analysis & Discussion:**

Explain how and why you do these experiments? Explain how you collect those

measurements? Show the result of your experiments in plots, and explain your

observations.

Methodology

1. System Spec (If you run your experiments on your own machine)

Please specify the system spec by providing the CPU, RAM, storage

and network (Ethernet / InfiniBand) information of the system.

沒有，我使用的是hades

1. Blocking Factor **(hw3-2)**

Observe what happened with different blocking factors, and plot the trend in

terms of Integer GOPS and global/shared memory bandwidth. (You can get the

information from profiling tools or manual) (You might want to check nvprof and Metrics Reference)

以下的圖表我將不會使用我的final version ( 因為他已經過度優化，無法隨意調整blocksize )

因此 我會將整個結果建立在 version 7 ( 可以過到p23k1) 而不是最終版本。

而已下的資訊是建立在c19.01

測量方法為先將inst\_int 數量除以 total time。Memory bandwidth 的部分則是將 load/store 的加總在除以時間。

可以看到global memory因為blocking factor增大而遞減。推測是因為主要運算都是在shared memory，而我們在phase3主要只有將global的東西寫入與寫出，而blocking size變大會造成需要讀寫的次數變少進而造就這個結果。

可以看到並沒有很大的差別，這邊想不太太好的解釋方式。可以看到在雖然有增加，但之後就沒有太大的變化，推測是因為這個版本寫得沒有很好造成如此的現象。

但我們還是可以看到GOPS會因為blocking factor增加而增加。主要是因為shared memory的速度真的很快，因此在運算上如果一次般較大的資料過來，並不會成為bottleneck。因此可以看到隨著blocking factor增加而增加。

C. Optimization **(hw3-2)**

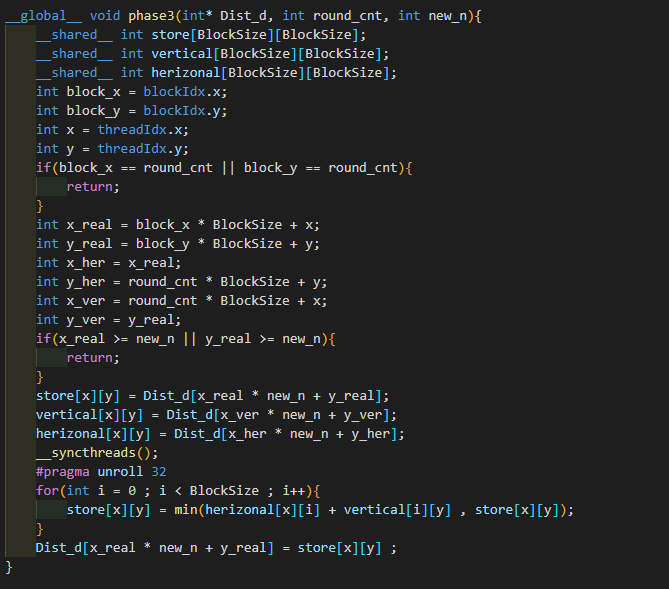
1. Shared memory

2. Coalesced memory access

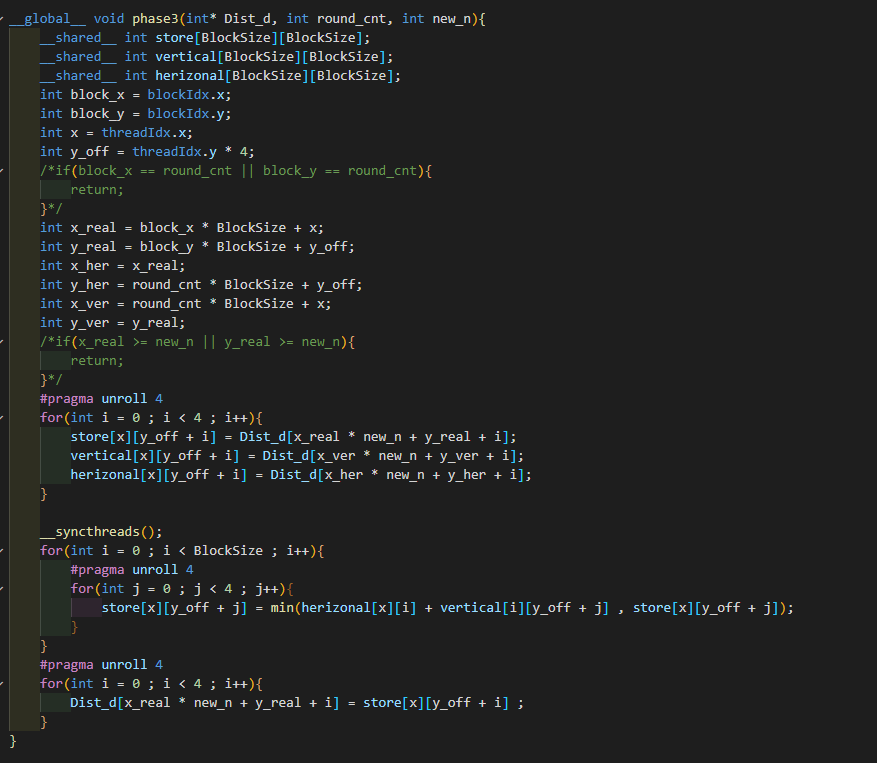
3. No parathesis

4. Large blocking factor

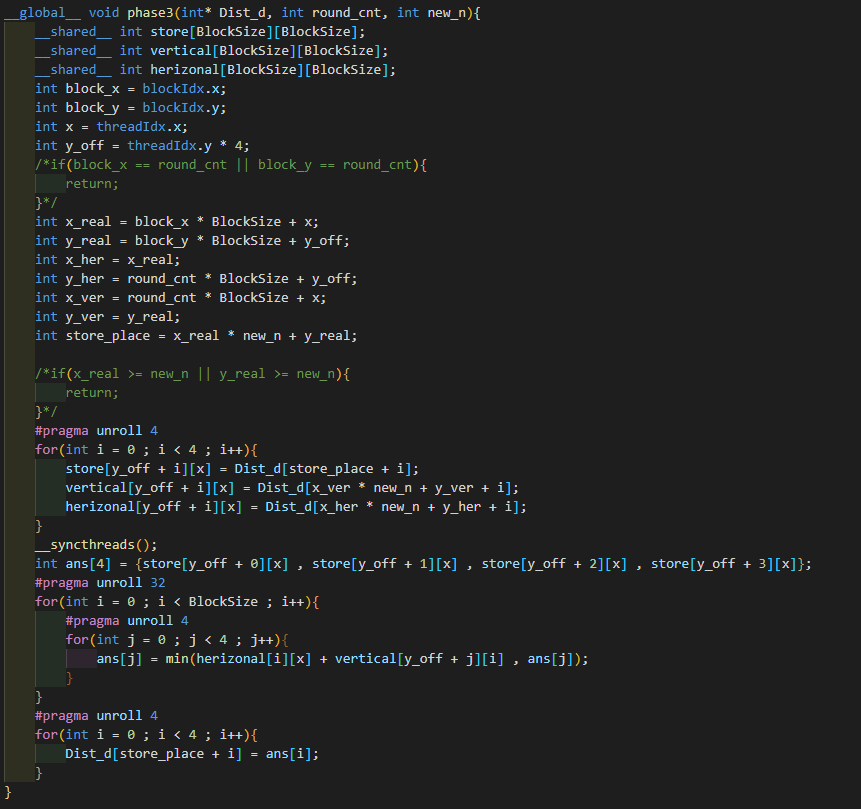
其實主要來說，最好的方式就是shared memory好好調教。這邊就以我自己的版本為例，因為我無法像speculation那樣的寫 ( 我認為這不是很好劃分 像是shared memory的access pattern算不算一種coalesced memory access ? (bank conflict )) 因此主要以我以下的為主。接下來主要講的phase 3



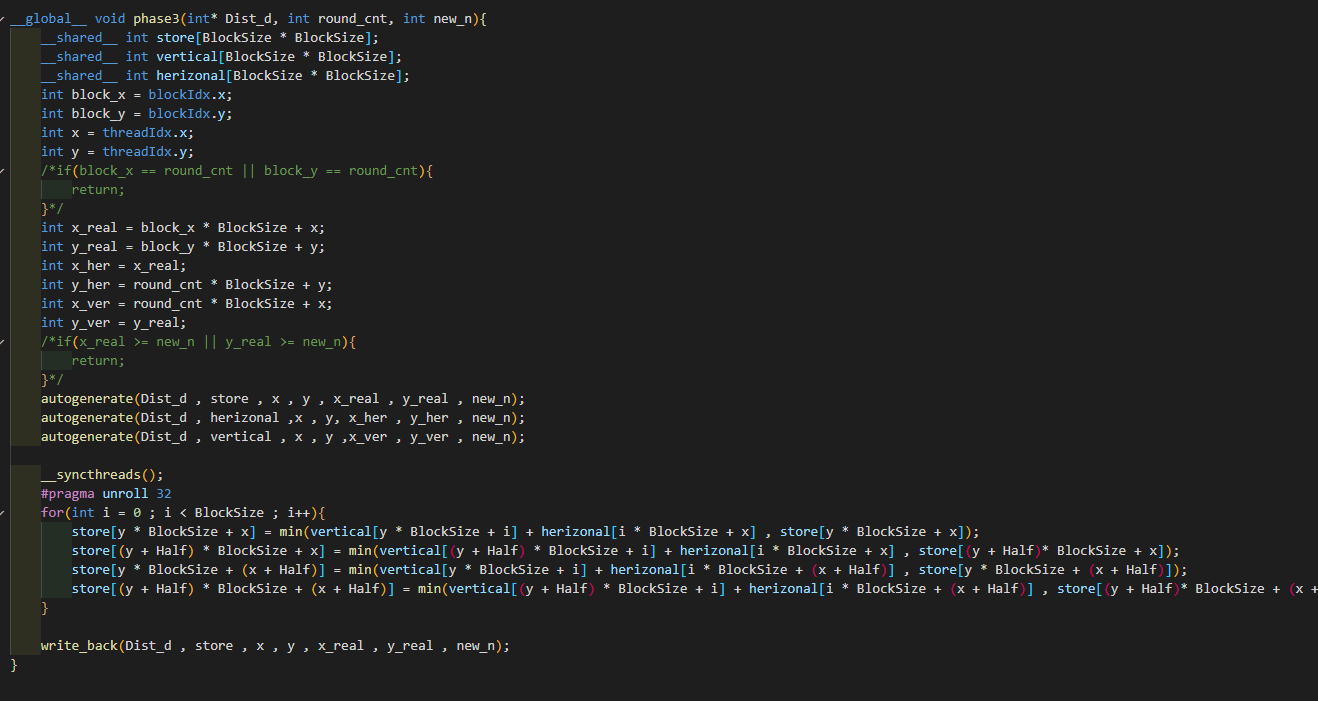
在一版中，我使用的技巧是沒有很嫻熟的share memory與判斷式。同時在input與output 有使用coalesced memory access ( 這邊快了1.5倍左右 )



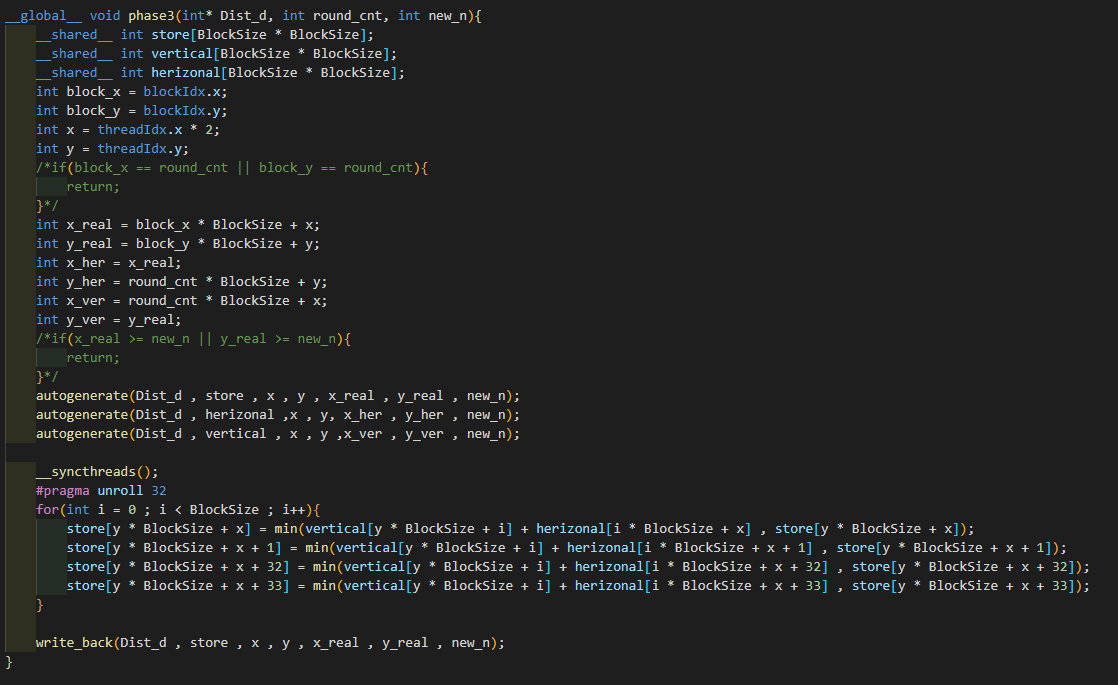
在第二版中，我將一個thread需要運算的東西調整為原來的四倍，並同時將if else的statement拿掉 ( 因為不想要發生branch ) 這個版本變快了兩倍。同時為了更好的access memory因此將原本的(32,32) 改為 (16,64) 這樣的話就可以一次讀取 x , x+1,x+2 , x+3 而不是 x , x + 32 , ( y + 32) \* n + x …



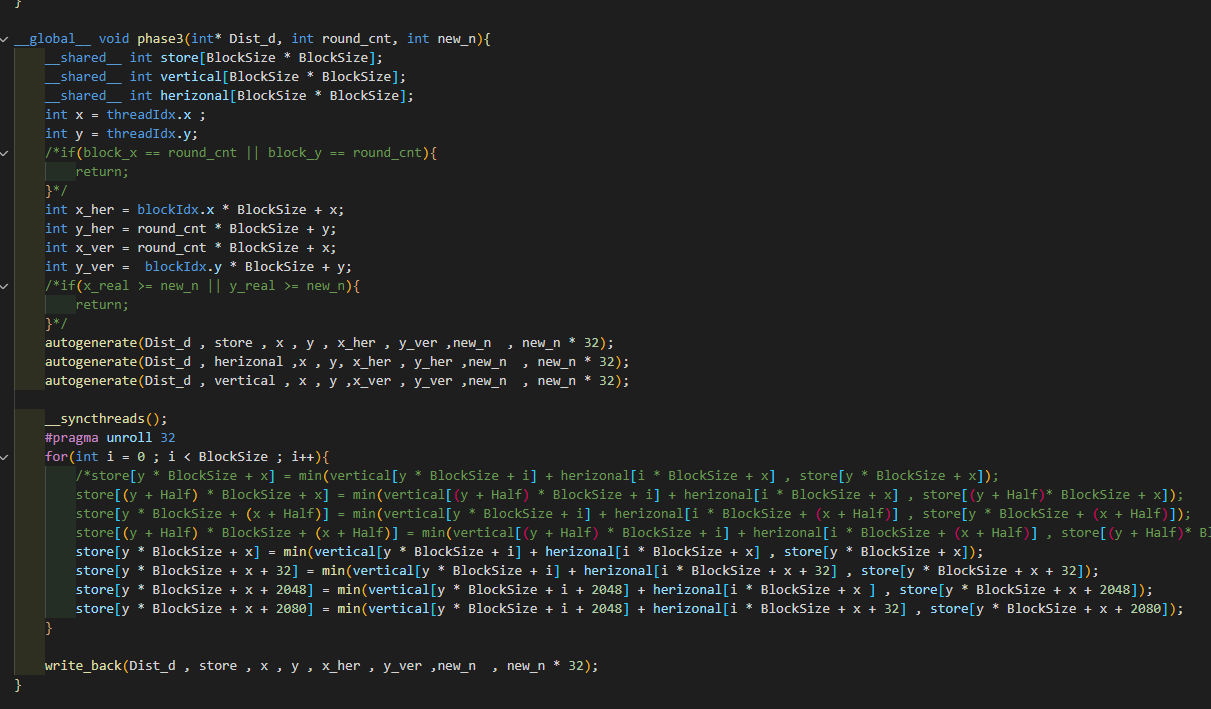
這是第五版，因為3.4.5版本中最後總共improved了2~3倍左右，因此這邊統一說明。主要的猜測shared memory access產生了bank conflict。所以就索性對調了一下access pattern(因此我將單純store的行列對調)發現他整體會快了兩倍左右。



Version 7這邊是將blockSize改為原本的 (32,32) 來測試效果。發現access解決bank conflict又可以增加兩倍的時間。



Version 9 ( 差了一筆 ) 是一直測試access pattern。並發現用(16,64) 可以加速不少 ( 兩倍 ) 因此就在這個情況下一直在做input/output的speed up。



最後一個version最特別，因為我在試的時候發現其實 ( x << 6 ) 會比 x \* 64 還要慢。將原本 y \* blockSize + x 變成 temp並將全部改成temp不會加速多少。因此嚴重懷疑parathesis在gpu computing裡面造成的速度下降。 因此，就索性將原本很慢的 (32,32) 來使用並直接使用 數字來代表access block，最終又增快了 1.2 倍。((hurray))

因此我不會對optimization作圖 ( 因為太難界定甚麼東西會快幾倍了 ) 。而且在programming中都是可能一次改一小部分，並沒有所謂一次將整個shared memory都改道到完善。

Weak scalability :

我的作法只有將phase 3 分段，因為floyd-warshell是一個dependency很高的算法，因此每次都會需要將資料傳給對方以確保答案的重要性。因此，大概估了一下我的scalability是1.7倍左右。

Time Distribution:

**Experience & Conclusion**

1. What have you learned from this homework?

Cuda programming 好像在通靈。有時候改一下就發現突然變快了。有時候改一典認為會變快的東西結果竟然變慢了。同時，要考慮蠻gpu硬體架構，這是之前軟體課沒有教過的東西，其實還蠻新奇的。

1. FeedBack : spec其實用字有點奇怪 像是block factor與 block差異點在哪裡? 還有一些communication ? 其實不太懂