**AAHLS Final Project**

**Scalable matrix matrix multiplication**

方爾秋 謝睿宇

* **Paper Introduction**

這篇Paper主要著重在configurable的MMM架構上，除了透過tiling方式適應各種不同size的input(matrix)之外，也在此基礎上盡可能的完全榨乾任何computation和memory resource，並實現最好的computation intensity。Paper Introduction主要分為3個部分，分別是resource、I/O以及hardware implementation。

1. **Resource**

Design overview

一張含有 文字, 方案, 圖表, Rectangle 的圖片

AI 產生的內容可能不正確。

Configurable MMM的hierarchy可以分為兩大部分，往下又能再細分成四個區塊，分別是memory tile、block tile、compute tile以及processing element。兩大部分則是memory以及computation。首先我們先看computation部分，在computation中最小的單位是compute unit，一個compute unit支援one word的MAC(乘加器)，PE包含個compute unit，再往上是compute tile，它包含個PE。這邊就是所有parallel processing compute unit數量()。再往上是block tile，他是指一次parallel processing所需要的block ram數量。由於這是比較抽象的tile，故這裡的block ram數量不需要是整數。而memory tile包含個block tile，其中代表目標平台的block ram數量，而是所有compute unit 數量，這邊是粗略假設一個compute unit需要一個block ram。

Computation Resource

接著我們可以看computation resource的限制，假設一個compute unit需要的hardware resource是 ，而目標平台最大的hardware resource是，再考慮單一個PE所需要Control overhead是，我們就能得出 。如果再考慮routing congestion問題，給定不同PE之間一條Bus width是wp，我們就會有額外兩條不等式。

透過上述不等式能推出，以此推估不考慮data movement的computation time T = F /( f )。f是clock frequency。

一張含有 文字, 螢幕擷取畫面, 字型, 圖表 的圖片

AI 產生的內容可能不正確。

Memory Resource

令是一個compute unit所需的bandwidth，是一個block ram的bandwidth，由於block tile是支援compute tile parallel processing 次的block ram數量，實際數值可以透過以下公式算出:

而memory tile內包含的block ram數量也可透過下列公式算出:

因為一個block ram可以儲存 個words，因此一個block tile可以支援compute tile 次parallel processing，進而可以求得一個memory tile可以支援次block tile。在最好的情況下，是的整數倍，而最糟情況下 = 2 – 1。

1. **I/O**

在做I/O model之前，我們需要先透過DAG了解MMM的memory使用情況，DAG如下圖所示。

一張含有 文字, 圖表, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

可以看到A 是一個i\*k的matrix，而B是一個k\*j的matrix，而C則是i\*j的matrix。此外，也能觀察到intermediate term總共有i\*j\*k項。故我們的目標就是盡可能減少intermediate term write back。具體實現方法可以採用red blue pebble game概念的model作分析。

下圖是一個red blue pebble game的示意圖。紅色代表fast memory(on-chip)，而藍色代表slow memory(off-chip)。一個node只有在他所有parents node都是紅色時才能變紅色，也就是只有在fast memory (compute)後才能存在fast memory(compute result)。Red blue pebble game的核心重點是在有限的fast memory和無限的slow memory下如何盡量減少顏色轉換的次數(data movement from slow to fast or fast to slow)，也就是提升computation intensity。

一張含有 行 的圖片

AI 產生的內容可能不正確。

這篇paper參考COSMA[2]的work並將其稍作修改使其適用於configurable FPGA上，首先 𝑆 = · ，代表on-chip memory能儲存的總word數。= = ，代表subcomputation也就是一個memory tile的x和y的input數量。目標是在不使用超過有限的fast memory的情況下: , ，盡可能提升computation intensity，也就是

一張含有 字型, 行, 白色, 筆跡 的圖片

AI 產生的內容可能不正確。最終結果是:



1. **Hardware implementation**

Routing congestion

如果input是透過broadcast方式連接所有compute unit，並且要在一個cycle內完成運算，總體的fan out會是。為了緩解這項問題，paper採用systolic array方式。

一張含有 文字, 圖表, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

一張含有 文字, 字型, 圖表, 方案 的圖片

AI 產生的內容可能不正確。一張含有 文字, 螢幕擷取畫面, 字型, 圖表 的圖片

AI 產生的內容可能不正確。如下圖，透過systolic array方式架構PE array(compute tile)可以將每個PE的connect減少到6個data bus，而為了去進一步適配不同hardware platform比如可能有non-uniform或hierarchy結構，需要將two dimension array變成one dimension如下圖所示，具體細節由另一位同學在code部分做介紹。

接著是matrix A的column-wise read問題。由於outer product需要讀取matrix A的部分column vector，這種column-major access會造成I/O效率低下，因此需要採用on the fly transposition方式，具體細節由另一位同學在code部分做介紹。

再來則是partial C write back問題，先前的work是透過使用double buffer方式解決，但這也會使得fast memory只剩一半的容量，這會導致computation intensity地降低，因此這篇paper採用sequential execution方式，這引入額外的write back overhead。這之中需要 cycles來write back，以及個cycles compute。如果的話，write back時間可以忽略不計。

最後是data dependency問題，如果target data type是floating point的形式，由於**FPGA** 並不原生支援浮點數累加這類操作，會使得MAC需要多個cycles才能輸出結果，為了應對這個情況，code中引入一種巧妙的分解方式。每個外積 (outer product) 被分解成個內部記憶體瓦片 (inner memory tiles)。每個inner memory tiles的計算結果會被存入不同的、fast memory位置。因為這些結果是存到不同位置，所以distant增加，每次衝突之間有個週期的間隔。只要distant大於累加的延遲時間，這種分離就能確保pipeline正常運行，從而提高效率。

1. **Throughput Drop**

在throughput plot中可以觀察到一些特殊點會有一個明顯的drop，這是因為隨著input增加，需要的memory tiles數量突然增加，以下圖為例子

一張含有 Rectangle, 行, 正方形, 螢幕擷取畫面 的圖片

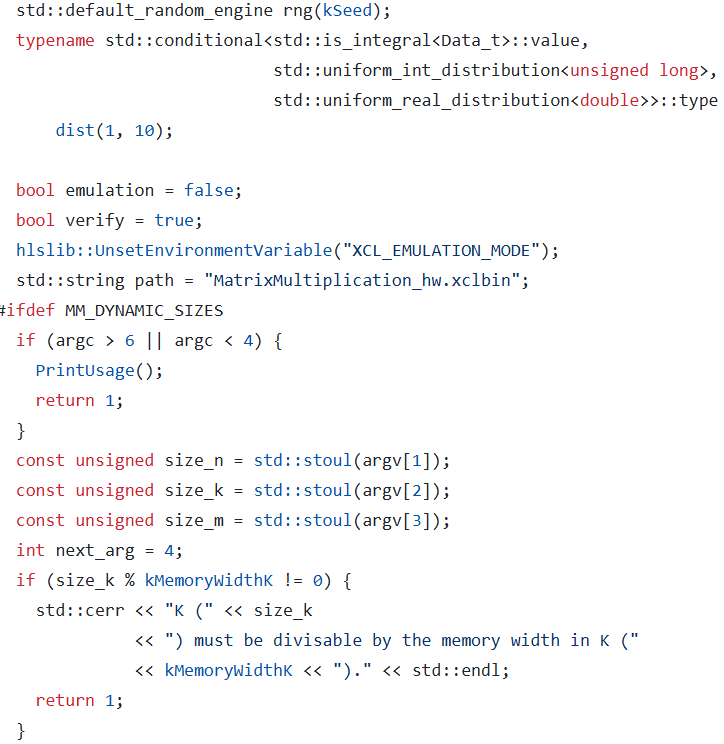
AI 產生的內容可能不正確。一張含有 行, Rectangle, 平行, 螢幕擷取畫面 的圖片

AI 產生的內容可能不正確。

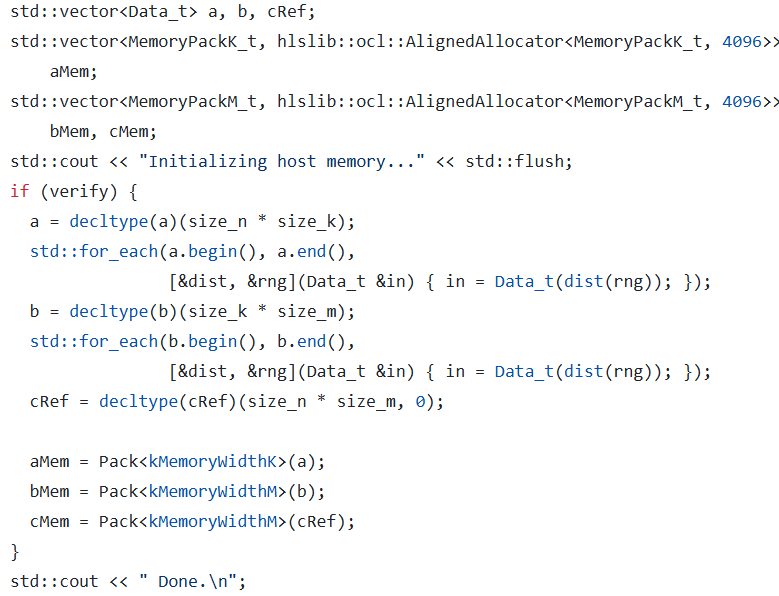
最大的方框是輔助線，實際memory tile size只有一個小方框，可以看到隨著input size增加，memory tile從原先1個變為4個，這導致很明顯的throughput drop，具體模擬結果會由後面同學做進一步講解。

* **Code Explanation**

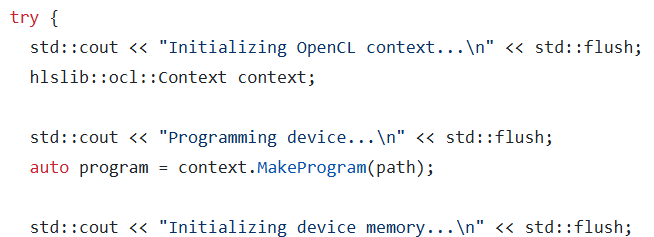
1. **RunHardware.cpp**

****

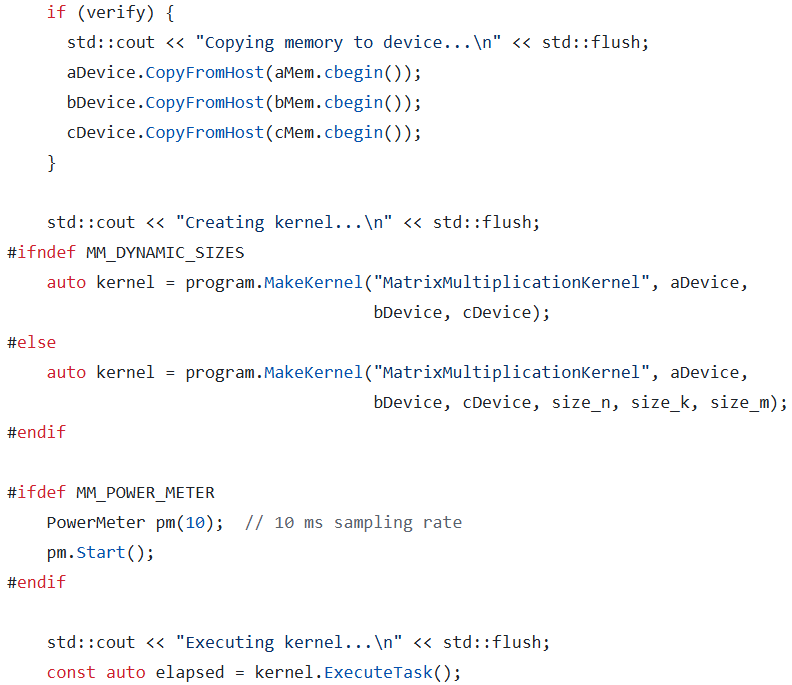
解析指令並初始化參數，從指令列讀入 N, K, M，並檢查 K/M 是否能被記憶體寬度整除（memory alignment），否則報錯。



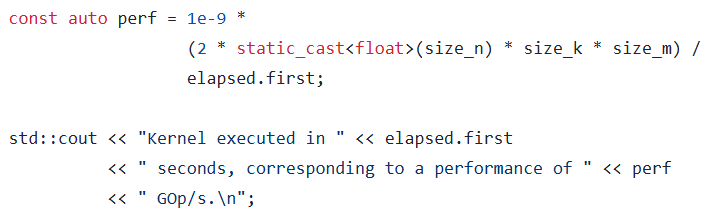
初始化矩陣a, b, cRef，其中a和b矩陣皆使用隨機數初始化矩陣的內容，而cRef初始化為0。



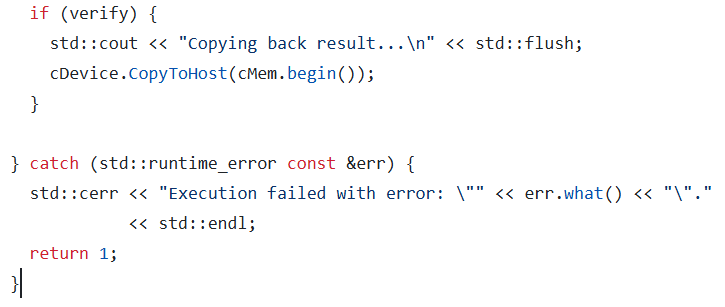
初始化 OpenCL 。



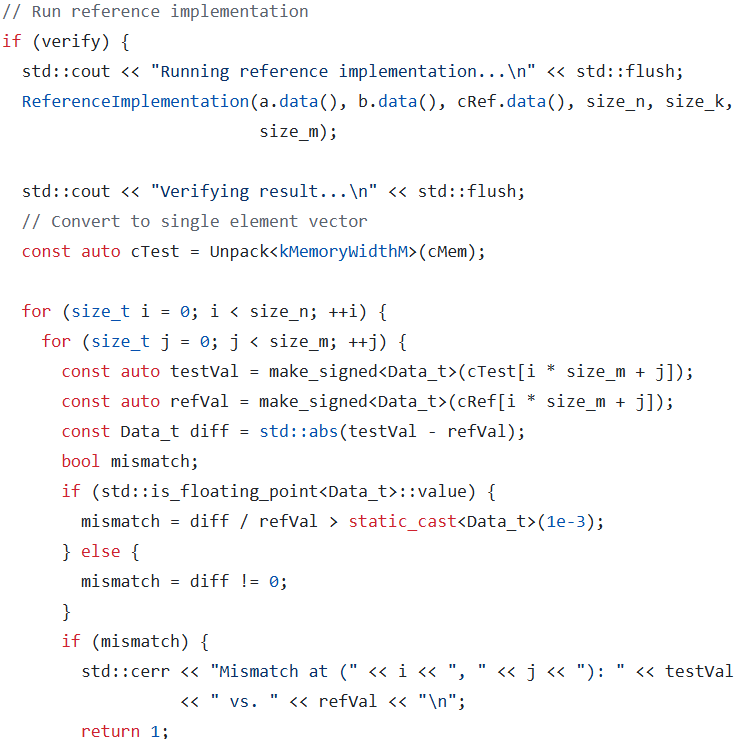
建立kernel實體並複製矩陣a, b, cRef，由Host傳到kernel後開始執行kernel，同時測量時間。



計算performance。

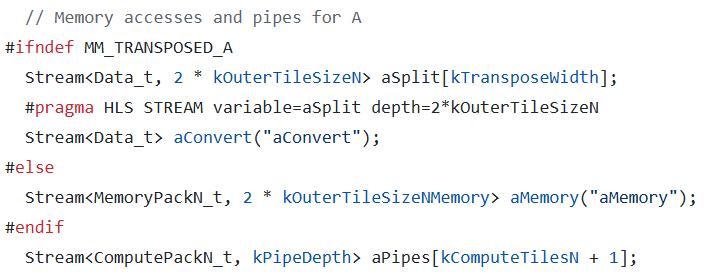


將kernel計算完的結果回傳到Host。

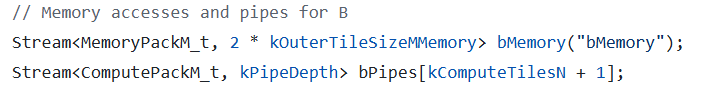


驗證結果是否正確。

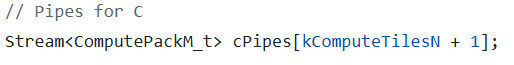
1. **TOP.cpp**

****

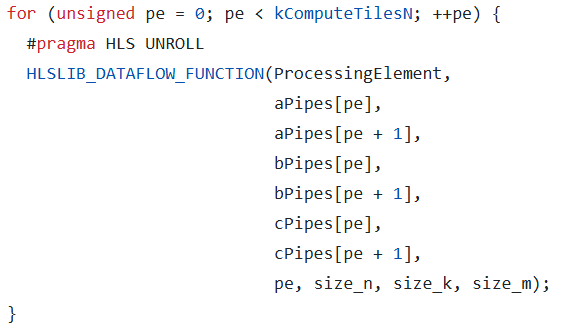
如果transpose A矩陣並無被定義，則利用aSplit[]把原始 A 拆成多條 stream，分開送給後面做 transpose，並同時轉換memory witdth。如果有定義transpose A矩陣，則直接將A讀入並存於aMemory。最後利用aPipes[]把A的資料送給每個PE使用。



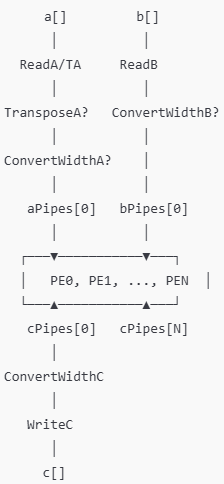
將B讀出後存於bMemory，之後利用bPipes[]把資料送給PE使用。



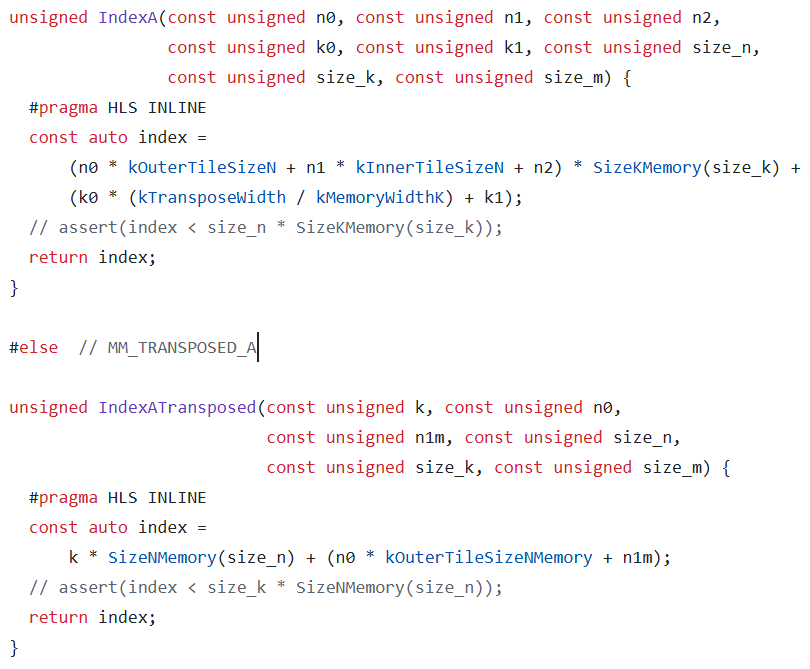
C的話則直接用cPipes[]把PE執行完的結果送回memory。



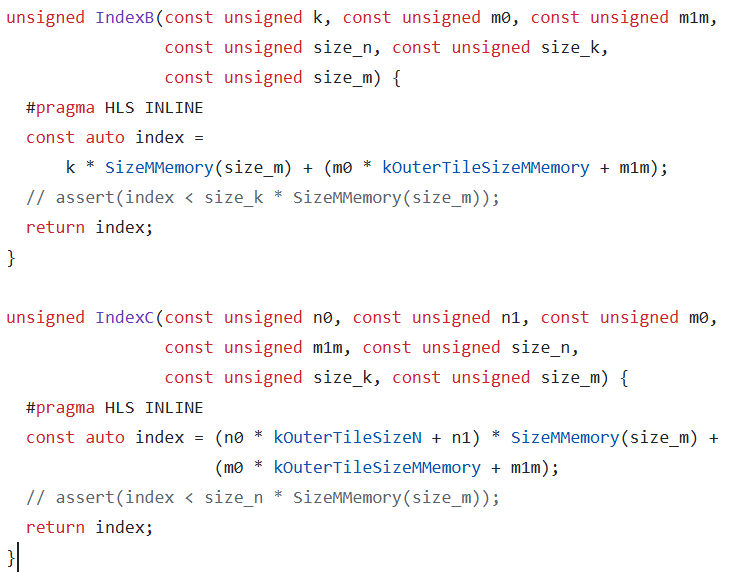
每個 PE 拿到一部分 A 和整個 B 的資料，然後算出對應 C 的一部分，把結果往後傳。這種串接方式很適合 pipeline 資料流運算。



1. **Memory.cpp**

****

對於未轉置矩陣A，index = n \* stride\_k + k，適用於column-major。而對於已轉置矩陣，則是以row-major，因此index = k \* stride\_n + n。



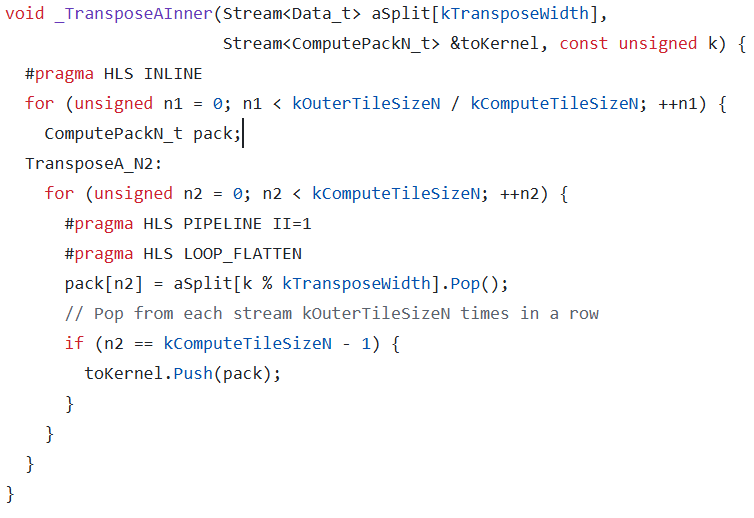
而對於B和C因為不會有轉置的狀況，因此都是以column-major為主。



ReadAInner讀出單個 tile 中某個元素的「memory pack」（如 128-bit / 256-bit）並拆分送入多個 stream，一次處理一筆 memory packed data。ReadAInnerLoop控制 tile 的迴圈，呼叫 ReadAInner 來完整讀完一塊 tile 中的資料，每次會掃過 n2（tile 內部 row）與 k1（tile 內部 col）。

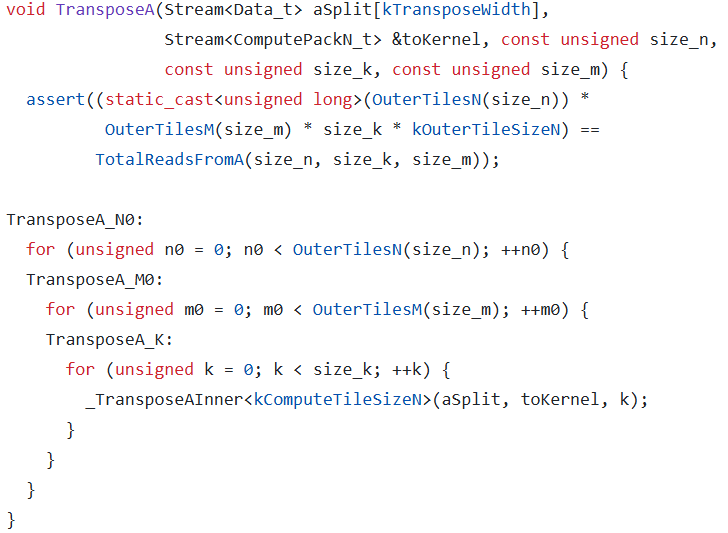


從 memory 讀取打包過的矩陣 A，拆解後推入 stream aSplit[]。n0(outer tile row index), mo(outer tile column index), k0(index of k dimension), n1(number of inner tile)。最後呼叫ReadAInnerLoop處理 tile 內所有資料，並將 A 的每一 row 依照其 column（k0）拆入對應的 stream aSplit[k1 \* kMemoryWidthK + w]。

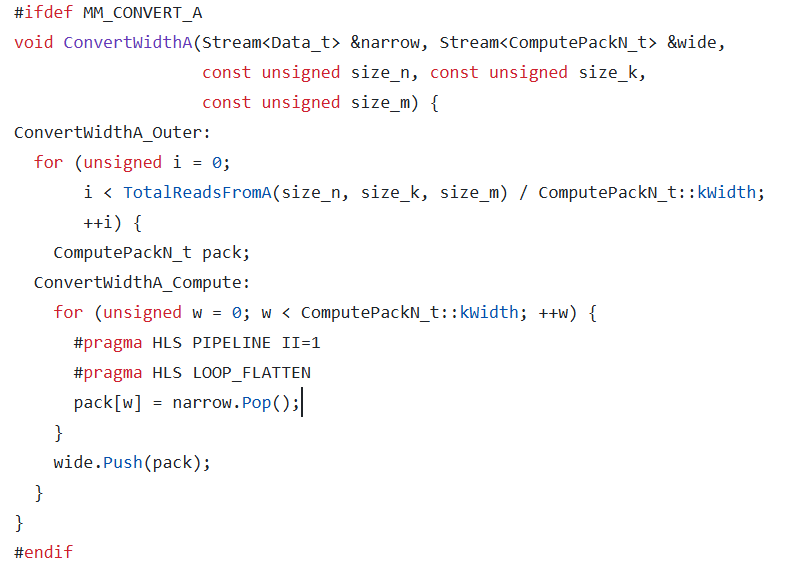


每次從某條 stream 中連續 Pop 出 kComputeTileSizeN 筆資料

組成一個 ComputePackN\_t (kernel所需要的形式）。



將stream的資料轉成row-major並送給kernel。這裡的 k是column index。每次從對應的 aSplit[k % kTransposeWidth] 取出資料，因為A的不同 column 原本被分到不同的 stream，現在要依 column 來收集。



由於A矩陣需要進行轉置，這會造成memory 資料的寬度與 kernel 所需的 compute 寬度不同，因此將單筆stream narrow 組成 ComputePackN\_t 的寬度(pack)，使資料寬度改變。



矩陣 A 已經以轉置形式儲存在 memory 中。所以 memory 的資料 layout 是按 column-major 排列（也就是按原始 A 的 K 為主）。這樣可以直接讀取每一 column 的資料，不需在硬體中額外轉置。



從 寬資料格式（memory 一次存 16 筆）轉換成較窄格式（kernel 一次只吃 4 筆），以對應 kernel 運算需求。

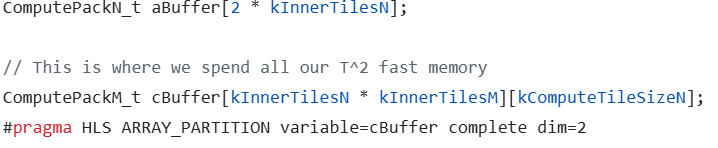


和ReadATransposed的功能相同。

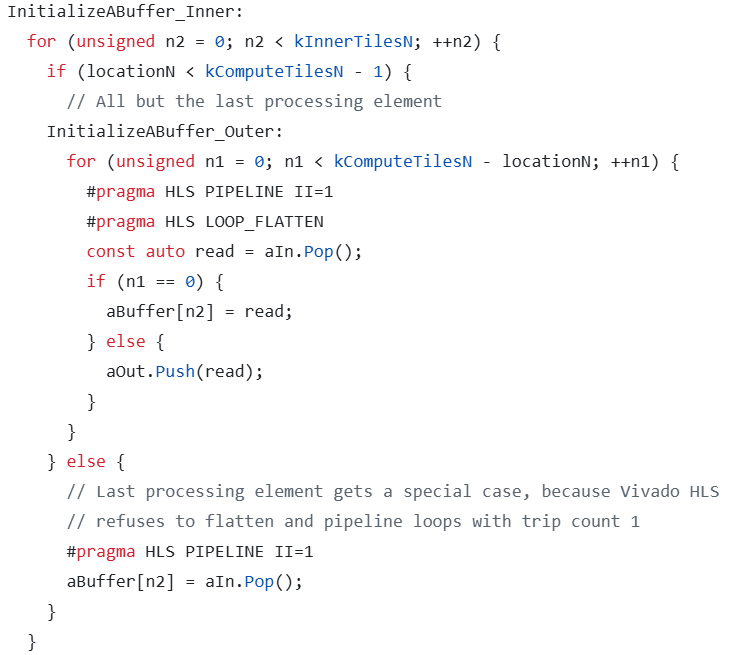


n0, m0 是outer tile索引，n1, m1m 是inner tile的位置，使用 IndexC 計算此 tile中第幾筆對應到的 global memory address，將 pipe 中的結果 pop 出來後寫回對應記憶體位置。

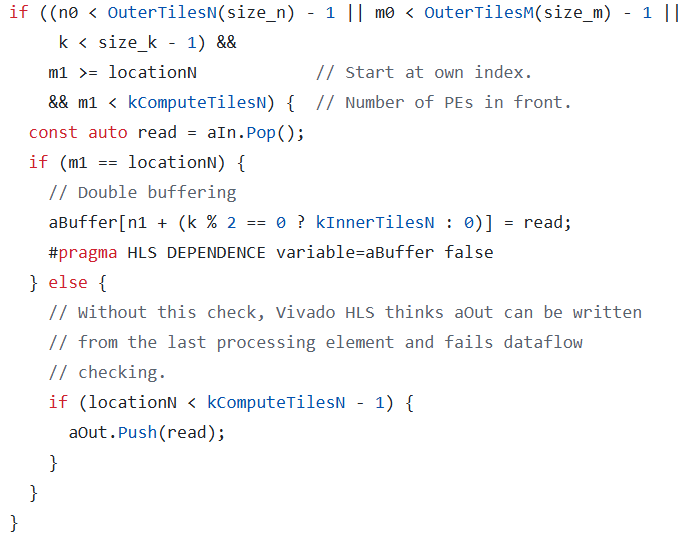
1. **Compute.cpp**

****

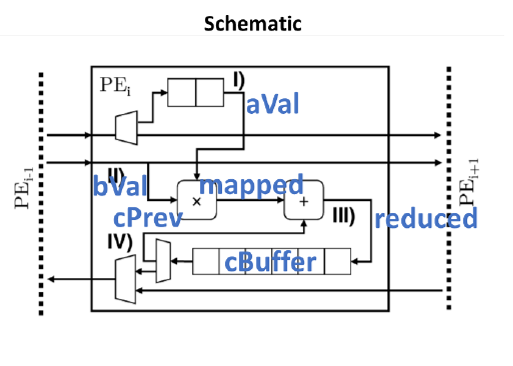
A 是雙緩衝設計，可以邊讀資料邊進行外積，cBuffer用來暫存 C 的 tile 結果，dim=2完全切開（partition），讓每個 column 都能被同時存取，加速累加計算。

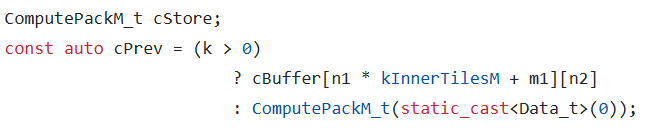


前面第 n1==0 筆是要自己用的，後面的資料會被往右傳給下個 PE。

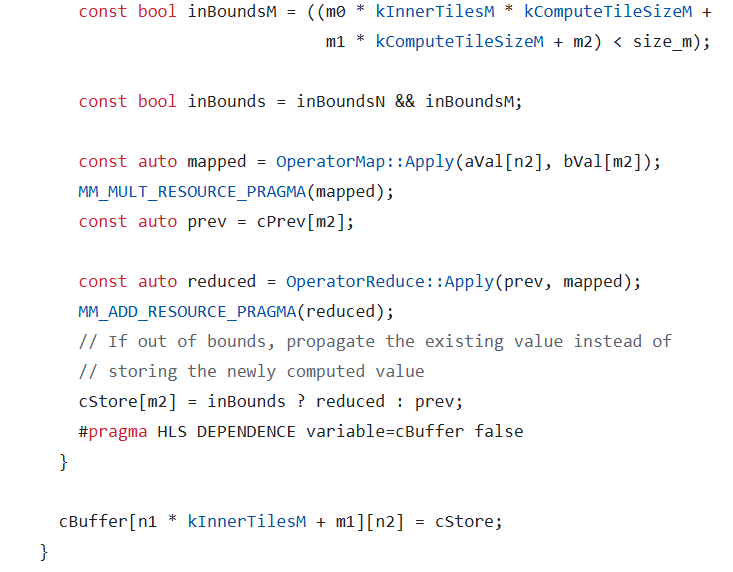


先以條件判斷在正確範圍之內，把自己的資料抓下來，同時把剩餘的資料下個PE傳遞。





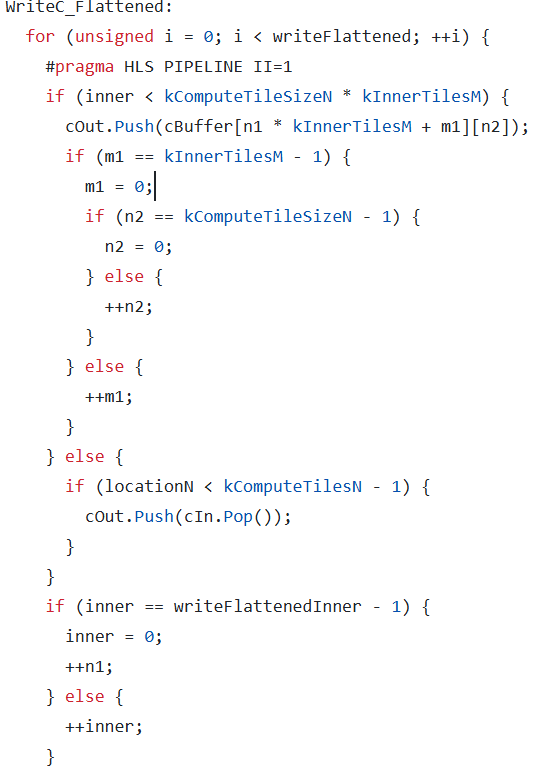
選取C矩陣的累加數值，如果是第一次(k=0)，就從0開始累加。



計算reduced = cPrev + (a × b)，使用兩個 Operator :

OperatorMap::Apply(a, b)：定義 a × b 的乘法邏輯

* OperatorReduce::Apply(prev, mapped)：定義如何累加，例如 +  
  用inBounds來判斷計算是否超過原矩陣大小（邊界 tile 的 padding），如果超過的話，就不要寫入新的值，保留原值。最後將這次累加的結果存到cBuffer。



由最末端的PE將資料一筆一筆送回，如果是再自己的範圍內就傳送本地資料，如果不是，則將上一個PE的資料傳遞下去。

* **HW Emulation Results**

一開始我們依照github 上面的指令執行，但發現些許問題。

1. Source code的Vitis版本與目前U50工作站的Vitis版本不同，需要先執行source /opt/Xilinx/Vitis/2021.1/settings64.sh，將版本切換為2021.1。
2. 原本我們打算在U50硬體上執行程式，但發現除了Vitis版本不同之外，xrt版本也不同。而xrt版本在工作站上只有最新版，無法切換成舊版，因此我們最後改成跑Hardware Emulation來模擬。
3. 由於工作站有租借時間限制，因此Memory Tile Size不能太大，否則會來不及跑完。

解決上述問題後，我們按照以下指令模擬，並有成功得到結果 :

mkdir build

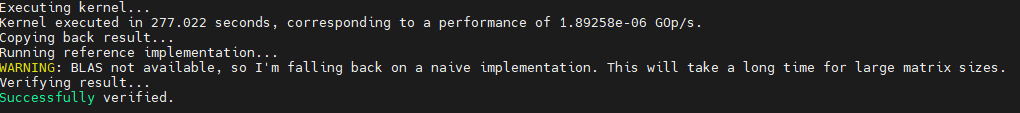
cd build

cmake ../ -DMM\_DATA\_TYPE=float -DMM\_PARALLELISM\_N=16 -DMM\_PARALLELISM\_M=4 -DMM\_MEMORY\_TILE\_SIZE\_N=64 -DMM\_MEMORY\_TILE\_SIZE\_M=64

make

make hw\_emu

./RunHardware.exe 64 64 64 hw



其中DMM\_PARALLELISM\_N代表在inner tile的PE數量，而DMM\_PARALLELISM\_M代表一個PE擁有的compute unit。我們將這兩個參數設定為16和4是因為在這個設定下的performance最好。

DMM\_MEMORY\_TILE\_SIZE\_N與DMM\_MEMORY\_TILE\_SIZE\_M代表Outer Tile 的大小，設定為64是因為能在工作站租借的時間限制內跑完，如果再往上調大，則無法進行模擬。

最後，./RunHardware.exe 64 64 64 hw這條指令後面的三個數字代表，A和B矩陣的長寬，也就是input size，因此我們模擬了以下幾種input size的設定(16,32,64,80,96,128)，並比較RunTime和Performace的差異。

從上圖可發現，當input size大於Outer Tile Size時會出現performance下降和runtime大幅度上升。我們猜測是因為當input size大於Outer Tile Size時，kernel需要多花4倍的時間來跑，也就是原本只需要1個Outer Tile的時間變成4倍，但除了第一段時間之外，其他的時間幾乎都在空轉，因此造成performance下降。

由於我們並不是在實際的硬體上面執行，因此有很多測試無法實作，目前只有先發現這個問題，尚未有實際的解決方式。

* Computation utilization



一張含有 文字, 字型, 白色, 書法 的圖片

AI 產生的內容可能不正確。

Assume hardware resource is fully utilized.

1024 \* 1024 \* 1024 / (32 \* 8) / (fs = 300e6) = 0.01398sec

0.01398/0.014706 = 0.951

* Reference

[1] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. Numer. Math. 13, 4 (Aug. 1969), 354–356.

[2] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In Proceedings of the International Confer ence for High Performance Computing, Networking, Storage, and Analysis (Denver, Colorado) (SC ’19).