高階合成技術於應用加速 Final Project Report

109061806 羅允辰 110061555 李冠霈 110061575 王睿瑄

Github 連結： https://github.com/jasonlo0509/gemm_hls

I. Introduction

矩陣乘法是重要的運算，特別是現在許多人工智慧、圖的相關應用都會使用到大量矩陣乘法。許多研究都嘗試探討如何設計優良的矩陣乘法加速器。

我們選定了目前公開在網路上最好的 HLS 矩陣乘法加速器，並且分析他如何用 HLS code 寫出需要的硬體結構、並對應將加速器移植到 U50 FPGA 上做分析，接下來的段落我們會依序介紹：

1. 系統架構
2. 各個硬體模塊的 HLS 語法與硬體之間的對應關係
3. 部署至 U50 的實際測試結果
4. 實驗分析
5. 結論

II. System Diagram

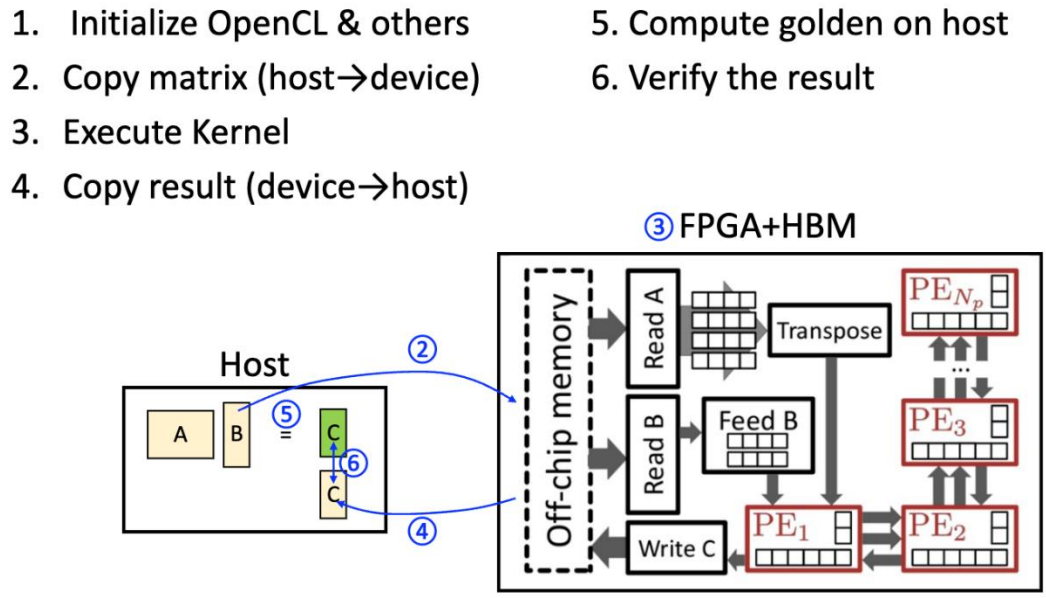


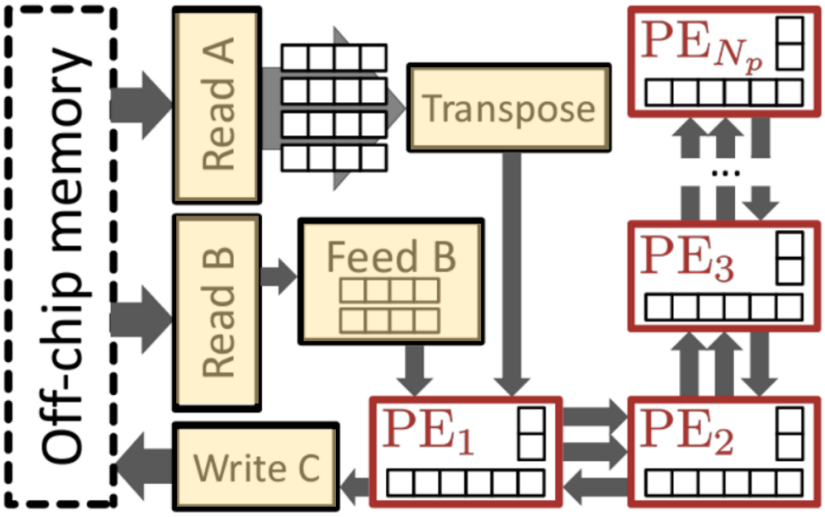
圖 1、系統方塊圖

整個系統可以分成 Host 和 FPGA 兩大部分，在驗證矩陣乘法加速器上可以分成幾個主要步驟：首先，我們會初始化 OpenCL 的 device 並創建好 buffer，接著，會將 host 端需要運算的矩陣搬到 FPGA 上並進行實際運算，最後的結果會被放回 Host，並與 Host 使用 CPU 算出的結果進行比對。

III. Block-wise Code Analysis

- **Memory-related**

在 Memory-related 的部份，會依序介紹 Read A、Transpose、Read B、 Feed B、Write C，以及 Convert Bitwidth，各個 block 的運作方式。



- **Read A**

在進行矩陣乘法運算時，會優先計算 row 方向的 output，因此矩陣 A 和矩陣 B 對應的讀取狀況如圖 2 所示，不論是矩陣 A 或是矩陣 B 都是以 row major 的形式儲存在 off chip memory，但是由圖 2 可以發現，要計算任何一塊 output M 的區域時，矩陣 A 的讀取方式和存儲方向是不同的，因此會先讀出圖 2 中矩陣 A 深綠的部份，再經過 Transpose 轉換成以 column(黃色箭頭)的次序送入 PE 中運算，因此在圖三的 Read A block 中的 size_k/KTransposeWidth 會選出要讀取的深綠色範圍，而 KInnerTilesN 則是會把多個列分為一組，切割矩陣 A 的深綠範圍。

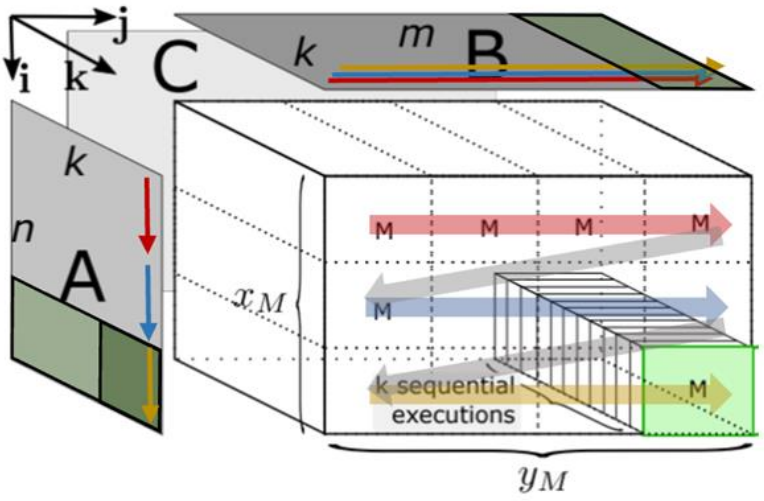


圖 2、矩陣 A、B 與 output 計算對應圖

```
void ReadA(MemoryPackK_t const a[], Stream<Data_t> aSplit[kTransposeWidth],
           const unsigned size_n, const unsigned size_k,
           const unsigned size_m) {
  assert((static_cast<unsigned long>(OuterTilesN(size_n)) *
          OuterTilesM(size_m) * (size_k / kTransposeWidth) * kInnerTilesN *
          kInnerTileSizeN * (kTransposeWidth / kMemoryWidthK) *
          MemoryPackK_t::kWidth) == TotalReadsFromA(size_n, size_k, size_m));

ReadA_N0:
  for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
  ReadA_M0:
    for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
    ReadA_K0:
      for (unsigned k0 = 0; k0 < size_k / kTransposeWidth; ++k0) {
      ReadA_N1:
        for (unsigned n1 = 0; n1 < kInnerTilesN; ++n1) {
          _ReadAInnerLoop<kTransposeWidth / kMemoryWidthK>(
              a, aSplit, n0, n1, k0, size_n, size_k, size_m);
        }
      }
    }
```

圖 3、ReadA block

在_ReadAInnerLoop 會更詳細的切割矩陣 A 的讀取順序，各參數的對應如圖 4，而由於一組 kTransposeWidth 的範圍，可能是由多個 memory address 儲存，因此_ReadAInnerLoop 和_ReadAInnerLoop<1>分別處理 kTransposeWidth=x*kMemoryWidth 以及 kTransposeWidth=kMemoryWidth 兩種狀況，以防止在 kTransposeWidth=kMemoryWidth 時，會造成 inner trip count=1，而無法進行 pipeline 的情況發生。

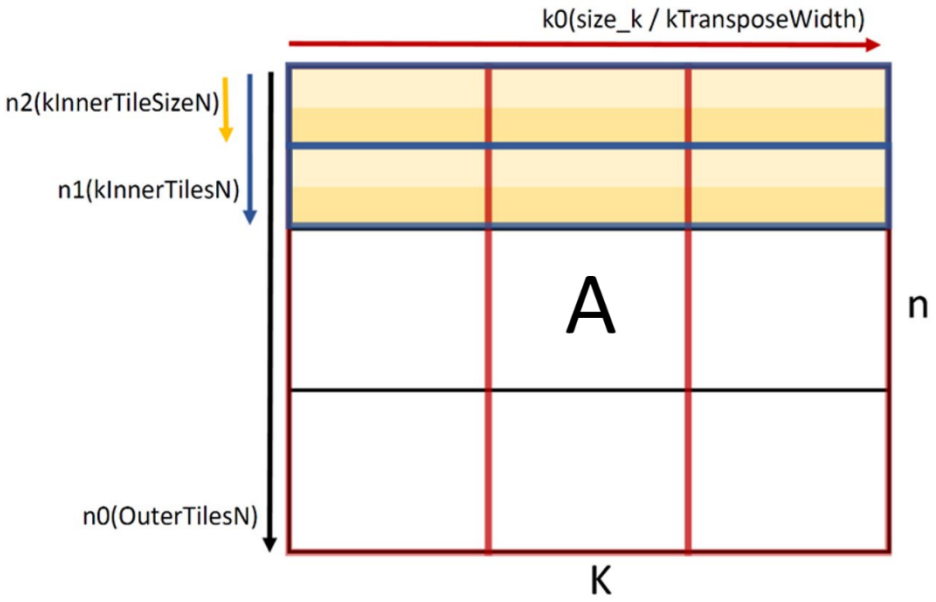


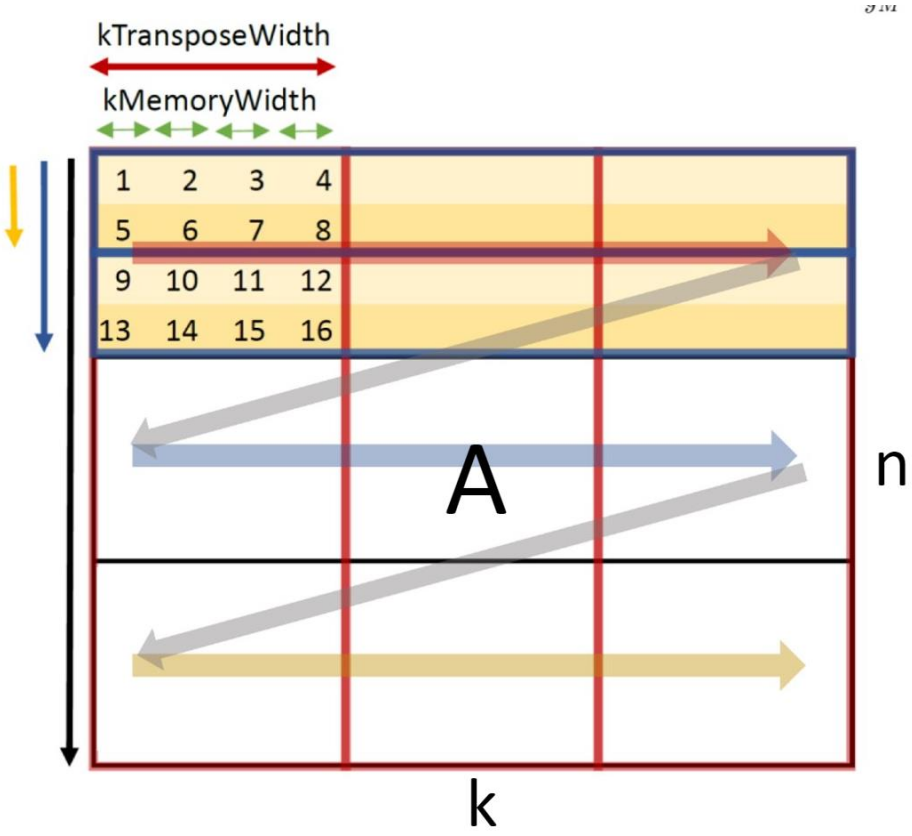
圖 4、矩陣 A 讀取順序示意圖

```
void _ReadAInnerLoop(MemoryPackK_t const a[],
                     Stream<Data_t> aSplit[kTransposeWidth], unsigned n0,
                     unsigned n1, unsigned k0, const unsigned size_n,
                     const unsigned size_k, const unsigned size_m) {
  #pragma HLS INLINE
ReadA_N2:
  for (unsigned n2 = 0; n2 < kInnerTileSizeN; ++n2) {
  ReadA_TransposeWidth:
    for (unsigned k1 = 0; k1 < (kTransposeWidth / kMemoryWidthK); ++k1) {
      #pragma HLS PIPELINE II=1
      #pragma HLS LOOP_FLATTEN
      _ReadAInner(a, aSplit, n0, n1, n2, k0, k1, size_n, size_k, size_m);
    }
  }
}

// Need a special case for kMemoryWidthK == kTransposeWidth, as Vivado HLS
// otherwise doesn't pipeline the loops (because the inner trip count is 1).
template <>
void _ReadAInnerLoop<1>(MemoryPackK_t const a[],
                        Stream<Data_t> aSplit[kTransposeWidth],
                        const unsigned n0, const unsigned n1, const unsigned k0,
                        const unsigned size_n, const unsigned size_k,
                        const unsigned size_m) {
  #pragma HLS INLINE
ReadA_N2:
  for (unsigned n2 = 0; n2 < kInnerTileSizeN; ++n2) {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_FLATTEN
    _ReadAInner(a, aSplit, n0, n1, n2, k0, 0, size_n, size_k, size_m);
  }
}
```

圖 5、_ReadAInnerLoop block

_ReadAInner block 中的 asplit 會有 kTransposeWidth 個位址，儲存由 memory 讀出來的矩陣 A，儲存的方式如圖 6 示意圖，會將同個 column 的數值儲存在同個 address 中，方便後續的讀取運算。而 IndexA block 則是會計算目前要讀出的矩陣 A 位址。



asplit[0]=[1,5,9,13]

asplit[1]=[2,6,10,14]

asplit[2]=[3,7,11,15]

asplit[3]=[4,8,12,16]

圖 6、 _ReadAInner block asplit 存取示意圖

```
void _ReadAInner(MemoryPackK_t const a[],
                 Stream<Data_t> aSplit[kTransposeWidth], const unsigned n0,
                 const unsigned n1, const unsigned n2, const unsigned k0,
                 const unsigned k1, const unsigned size_n,
                 const unsigned size_k, const unsigned size_m) {
  #pragma HLS INLINE
  auto pack = a[IndexA(n0, n1, n2, k0, k1, size_n, size_k, size_m)];
ReadA_Unroll:
  for (unsigned w = 0; w < kMemoryWidthK; ++w) {
    #pragma HLS UNROLL
    aSplit[k1 * kMemoryWidthK + w].Push(pack[w]);
  }
}
```

圖 7、_ReadAInner block

```
unsigned IndexA(const unsigned n0, const unsigned n1, const unsigned n2,
                const unsigned k0, const unsigned k1, const unsigned size_n,
                const unsigned size_k, const unsigned size_m) {
  #pragma HLS INLINE
  const auto index =
      (n0 * kOuterTileSizeN + n1 * kInnerTileSizeN + n2) * SizeKMemory(size_k) +
      (k0 * (kTransposeWidth / kMemoryWidthK) + k1);
  // assert(index < size_n * SizeKMemory(size_k));
  return index;
}
```

圖 8、IndexA block

- **Transpose A**

  _TransposeAInner 會根據目前要計算的 K 位置判斷所需要的數值被存在 asplit 的哪個位址，再根據 kComputeTileSizeN 的大小，也就是 PE 內部 compute unit 在 N 方向的個數，來決定要從 asplit[k% kTransposeWidth]中讀取幾個數值放進 toKernel，送入 PE 中進行運算。並會依照 kComputeTileSizeN 是否等於 1，決定要以 _TransposeAInner 或是_TransposeAInner<1>計算，避免 inner trip count=1，而無法進行 pipeline 的狀況。

```
void TransposeA(Stream<Data_t> aSplit[kTransposeWidth],
                Stream<ComputePackN_t> &toKernel, const unsigned size_n,
                const unsigned size_k, const unsigned size_m) {
  assert((static_cast<unsigned long>(OuterTilesN(size_n)) *
          OuterTilesM(size_m) * size_k * kOuterTileSizeN) ==
         TotalReadsFromA(size_n, size_k, size_m));

TransposeA_N0:
  for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
  TransposeA_M0:
    for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
    TransposeA_K:
      for (unsigned k = 0; k < size_k; ++k) {
        _TransposeAInner<kComputeTileSizeN>(aSplit, toKernel, k);
      }
    }
  }
}
```

圖 9、TransposeA block

```
template <unsigned inner_tiles>
void _TransposeAInner(Stream<Data_t> aSplit[kTransposeWidth],
                      Stream<ComputePackN_t> &toKernel, const unsigned k) {
  #pragma HLS INLINE
  for (unsigned n1 = 0; n1 < kOuterTileSizeN / kComputeTileSizeN; ++n1) {
    ComputePackN_t pack;
  TransposeA_N2:
    for (unsigned n2 = 0; n2 < kComputeTileSizeN; ++n2) {
      #pragma HLS PIPELINE II=1
      #pragma HLS LOOP_FLATTEN
      pack[n2] = aSplit[k % kTransposeWidth].Pop();
      // Pop from each stream kOuterTileSizeN times in a row
      if (n2 == kComputeTileSizeN - 1) {
        toKernel.Push(pack);
      }
    }
  }
}

template <>
void _TransposeAInner<1>(Stream<Data_t> aSplit[kTransposeWidth],
                         Stream<ComputePackN_t> &toKernel, const unsigned k) {
  #pragma HLS INLINE
  for (unsigned n1 = 0; n1 < kOuterTileSizeN; ++n1) {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_FLATTEN
    ComputePackN_t pack;
    pack[0] = aSplit[k % kTransposeWidth].Pop();
    toKernel.Push(pack);
  }
}
```

圖 10、TransposeAInner block

- **Read B**

在進行矩陣 B 讀取的部份，會根據 IndexB 計算出的數值，決定要從 memory 中讀取的位址，並將數值放入 pipe，之後會以 stream 的形式送入各個 PE 中運算。

```
void ReadB(MemoryPackM_t const memory[], Stream<MemoryPackM_t> &pipe,
           const unsigned size_n, const unsigned size_k,
           const unsigned size_m) {
  assert((static_cast<unsigned long>(OuterTilesN(size_n)) *
          OuterTilesM(size_m) * size_k * kOuterTileSizeMMemory *
          MemoryPackM_t::kWidth) == TotalReadsFromB(size_n, size_k, size_m));

ReadB_OuterTile_N:
  for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
  ReadB_OuterTile_M:
    for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
    ReadB_K:
      for (unsigned k = 0; k < size_k; ++k) {
      ReadB_BufferB_M1:
        for (unsigned m1m = 0; m1m < kOuterTileSizeMMemory; ++m1m) {
          #pragma HLS PIPELINE II=1
          #pragma HLS LOOP_FLATTEN
          pipe.Push(memory[IndexB(k, m0, m1m, size_n, size_k, size_m)]);
        }
      }
    }
  }
}
```

圖 11、ReadB block

```
unsigned IndexB(const unsigned k, const unsigned m0, const unsigned m1m,
                const unsigned size_n, const unsigned size_k,
                const unsigned size_m) {
  #pragma HLS INLINE
  const auto index =
      k * SizeMMemory(size_m) + (m0 * kOuterTileSizeMMemory + m1m);
  // assert(index < size_k * SizeMMemory(size_m));
  return index;
}
```

圖 12、IndexB block

- **Feed B**

在 proposal 有提到從 memory 中讀出的 Matrix B 會儲存在 Feed B 中，儲存的位置為圖 14 的”buffer”中，在 n1(KInnerTilesN)為 0，也就是第一次使用到矩陣 B 的該數值時，會將數值從 memory 中讀取，並存入 buffer 中，後續運算需要用到同樣的數值時，就會直接從 buffer 中拿取，而不會再由 off chip memory 中重新讀取，當 KInnerTilesM 的數值都存入 val 後，便會將 val 放入 toKernel，送入 PE 中進行運算，以圖 2 的 output tile 切割方式為例，圖 13 矩陣 B 的第一列數值會被重複使用 3 次(KInnerTilesN=3)，才會讀取矩陣 B 的第二列數值進行運算。

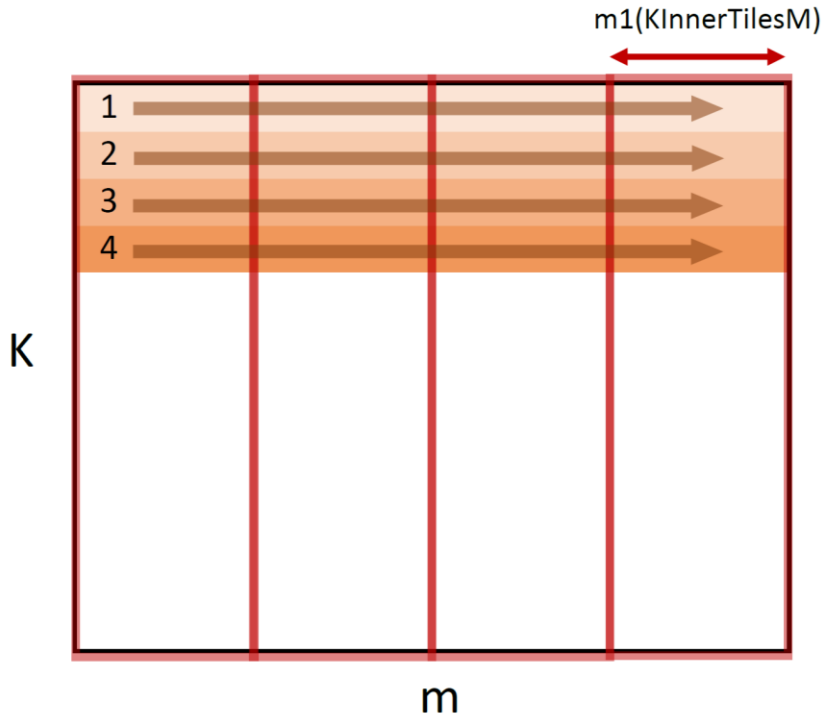


圖 13、矩陣 B 讀取順序示意圖

```
  const unsigned bound_n = OuterTilesN(size_n);
  const unsigned bound_m = OuterTilesM(size_m);

FeedB_OuterTile_N:
  for (unsigned n0 = 0; n0 < bound_n; ++n0) {
  FeedB_OuterTile_M:
    for (unsigned m0 = 0; m0 < bound_m; ++m0) {
    FeedB_K:
      for (unsigned k = 0; k < size_k; ++k) {
        ComputePackM_t buffer[kInnerTilesM];

      FeedB_Pipeline_N:
        for (unsigned n1 = 0; n1 < kInnerTilesN; ++n1) {
        FeedB_Pipeline_M:
          for (unsigned m1 = 0; m1 < kInnerTilesM; ++m1) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
            ComputePackM_t val;
            if (n1 == 0) {
              val = fromMemory.Pop();
              buffer[m1] = val;
            } else {
              val = buffer[m1];
            }
            toKernel.Push(val);
```

圖 14、FeedB block

- **Write C**

最後則是利用 IndexC 計算出來的 address，將 output 結果存回 memory 中對應的位址中。

```
void WriteC(Stream<MemoryPackM_t> &pipe, MemoryPackM_t memory[],
            const unsigned size_n, const unsigned size_k,
            const unsigned size_m) {
  // assert((OuterTilesN(size_n) * OuterTilesM(size_m) * kOuterTileSizeN *
  //         kOuterTileSizeMMemory * MemoryPackM_t::kWidth) == size_n * size_m);

WriteC_OuterTile_N:
  for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
  WriteC_OuterTile_M:
    for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
    WriteC_N1:
      for (unsigned n1 = 0; n1 < kOuterTileSizeN; ++n1) {
      WriteC_M1:
        for (unsigned m1m = 0; m1m < kOuterTileSizeMMemory; ++m1m) {
          #pragma HLS PIPELINE II=1
          #pragma HLS LOOP_FLATTEN
          const auto val = pipe.Pop();
          if ((n0 * kOuterTileSizeN + n1 < size_n) &&
              (m0 * kOuterTileSizeMMemory + m1m < SizeMMemory(size_m))) {
            memory[IndexC(n0, n1, m0, m1m, size_n, size_k, size_m)] = val;
          }
        }
      }
```

圖 15、WriteC block

```
unsigned IndexC(const unsigned n0, const unsigned n1, const unsigned m0,
                const unsigned m1m, const unsigned size_n,
                const unsigned size_k, const unsigned size_m) {
  #pragma HLS INLINE
  const auto index = (n0 * kOuterTileSizeN + n1) * SizeMMemory(size_m) +
                     (m0 * kOuterTileSizeMMemory + m1m);
  // assert(index < size_n * SizeMMemory(size_m));
  return index;
}
```

圖 16、IndexC block

- **Convert Bitwidth**

因為記憶體的 bitwidth 和 PE 需要的輸入 bitwidth 不一樣，因此需要進行位寬轉換 (如圖 17 所示，實際 B 矩陣需要 1->8 的轉換)，具體來說就是會有一個模組專門花八個 cycle 整理出運算需要的資料。圖 18 則展示了如何用 HLS 語法做到這件事情，最內圈的 for 迴圈是為了將資料從 narrow stream 讀出、排成需要的數筆一組的格式，並排到 wide stream 中。

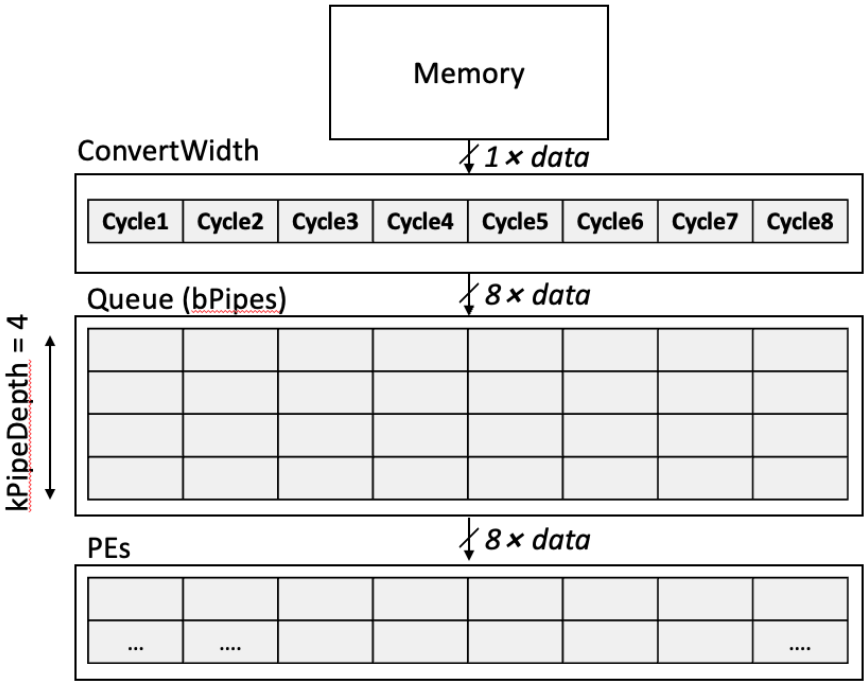


圖 17、Convert Bitwidth 示意圖

```
void ConvertWidthA(Stream<Data_t> &narrow, Stream<ComputePackN_t> &wide,
                   const unsigned size_n, const unsigned size_k,
                   const unsigned size_m) {
ConvertWidthA_Outer:
  for (unsigned i = 0;
       i < TotalReadsFromA(size_n, size_k, size_m) / ComputePackN_t::kWidth;
       ++i) {
    ComputePackN_t pack;
  ConvertWidthA_Compute:
    for (unsigned w = 0; w < ComputePackN_t::kWidth; ++w) {
      #pragma HLS PIPELINE II=1
      #pragma HLS LOOP_FLATTEN
      pack[w] = narrow.Pop();
    }
    wide.Push(pack);
  }
}
```

圖 18、Convert Bitwidth HLS code

- **Computation-related**

  在 Compute-related 的部分，會依序介紹關於 PE 的 architecture、data flow 及 compute flow。

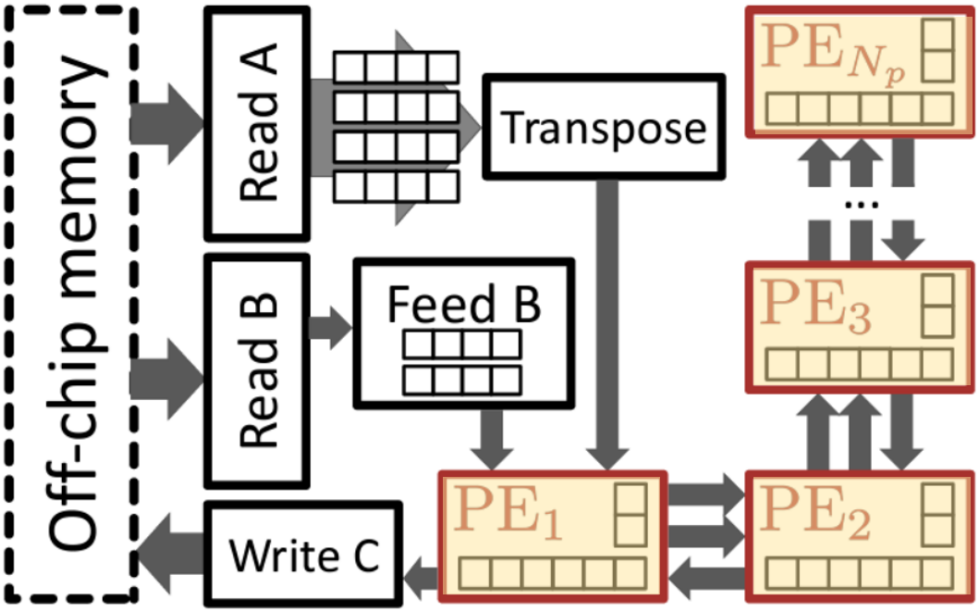


圖 19、Module Layout of final kernel architecture

  - **Architecture**

    對應 Block Decomposition 的參數如圖 21 所示。A、B 分別為 input 矩陣，C 為 output 矩陣，n、m、k 以 1024 為例。

    在實作矩陣乘法時，我們將矩陣拆分成好幾個 Outer Tile (如圖 21 中淺綠色的 M 矩陣)，此小矩陣的 size 稱為 kOuterTileSizeN 及 kOuterTileSizeM，在創立 kernel 時進行選擇，而我們在實作時先實驗了 512 作為 Outer Tile 的大小(見圖 20 的 DMM_MEMORY_TILE_SIZE_N、DMM_MEMORY_TILE_SIZE_M)。

    接著，將 Outer Tile 進一步拆分為 Inner Tile，而按照 cmake 指令，我們創立的 Inner Tile 為 kInnerTileSizeN (DMM_PARALLELISM_N) =32，kComputeTileSizeM (DMM_PARALLELISM_M) =8，而藉由 Outer Tile size 除 Inner Tile Size，可以得到 kInnerTilesM=512/8=64，kInnerTilesN=512/32=16。

    一小塊的 Inner Tile 由很多的 Compute Tile (PE)組成。其中 kComputeTileSizeN 為了建立 1D Array，因此被固定為 1。藉由 kInnerTileSizeN/kComputeTileSizeN=kComputeTilesN=32。

```
cmake ../ -DMM_DATA_TYPE=float -DMM_PARALLELISM_N=32 -DMM_PARALLELISM_M=8 -DMM_MEMORY_TILE_SIZE_N=512 -DMM_MEMORY_TILE_SIZE_M=512
```

圖 20、cmake 指令

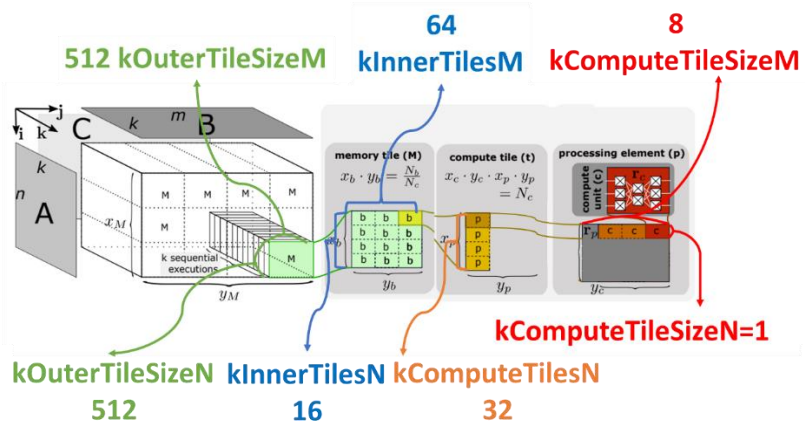


圖 21、Block Decomposition Parameters

圖 22 的 code 利用 pragma UNROLL 來達到 PE 平行處理的目標。搭配圖 22 將圖 19 的 PE 展開畫成圖 23，可以更好的理解 PE 之間 data 傳遞的關係，詳細的資料內容會在 PE Block 解釋。所有 PE 會平行處理，並按照順序計算每一塊 Inner Tile，當一塊 Outer Tile(圖 20 的淺綠色 M 矩陣)結束運算後，最後的 cPipes[31]會開始回傳資料給 cPipes[30]，並一直向前傳遞給 cPipes[0]，再移動至下一塊 Outer Tile，直到整個矩陣運算完畢。

```
for (unsigned pe = 0; pe < kComputeTilesN; ++pe) {
  #pragma HLS UNROLL
  HLSLIB_DATAFLOW_FUNCTION(ProcessingElement,
                           aPipes[pe],
                           aPipes[pe + 1],
                           bPipes[pe],
                           bPipes[pe + 1],
                           cPipes[pe],
                           cPipes[pe + 1],
                           pe, size_n, size_k, size_m);
}
```

圖 22、Processing Element Function

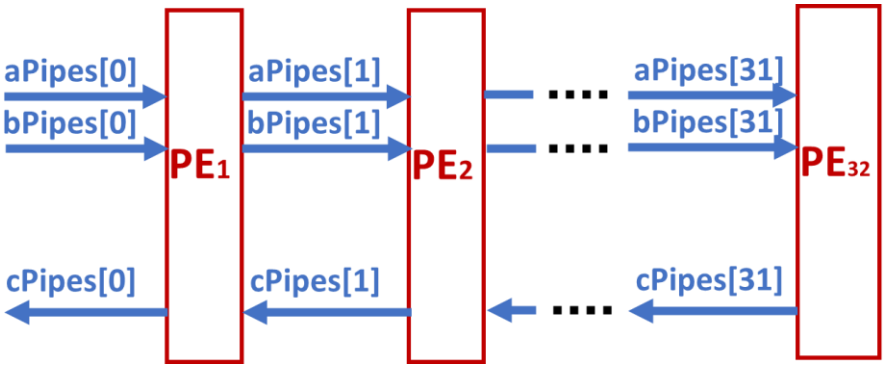


圖 23、Detail of data transfer between PEs

- **PE Dataflow**

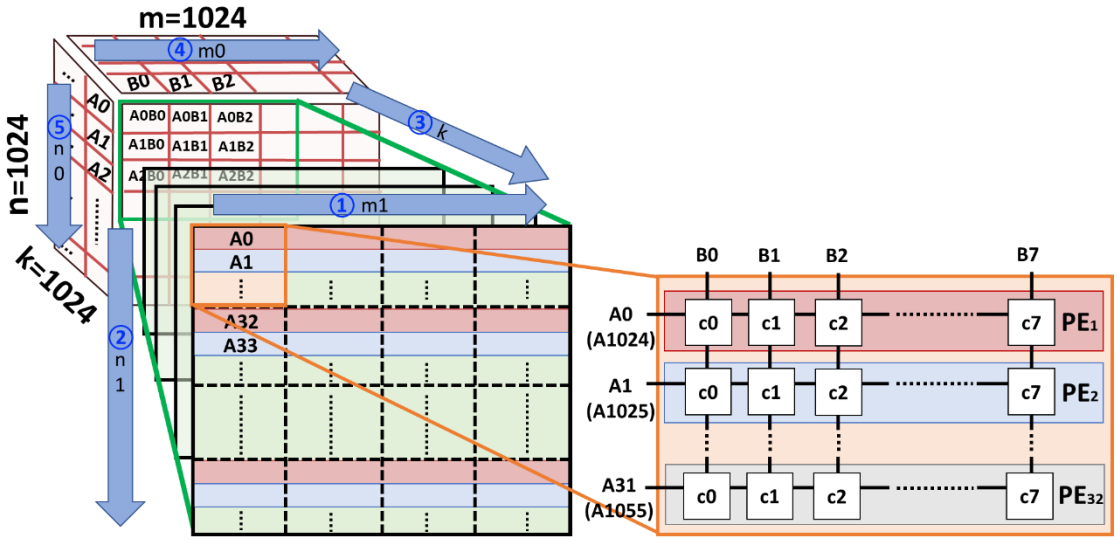


圖 24、Dataflow of MMM

圖 24 為矩陣乘法的 Dataflow。首先，Outer Tile 為綠色區塊，Inner Tile 為橘色區塊，紅色、藍色區塊則為 PE，一塊 Inner Tile 有 32 個 PE。以紅色為例，一個 PE 會同時計算 A0 與 B0、B1、B2…B7 的乘法，產生 8 個結果，並分別以 c0、c1、c2…c7 表示。在 PE1 計算的同時，由於圖 22 中 pragma UNROLL 的關係，PE2 也會一起計算 A1 與 B0、B1、B2…B7 的乘法，並產生 8 個結果，以此達到平行運算的效果。由此我們可以發現，由於 1D array 中的 PE 串現在需要計算多個 row 的 MAC value，因此 A 矩陣會經由圖 23 的 aPipes 將每一個 row 的 A data 一層一層傳遞至下一個 PE。由於 PE 數量龐大，傳遞很容易耗時，因此為了能夠良好的進行 Pipeline，A 會進行 double buffer，意即存取當下這個 k 與下一個 k 的 data (A0 在運算時會存 A1024)，就可以在算完一個 Outer Tile 時，立即運算下一個 k 的 MAC，而不用等待下一串 A data 的輸入，達到完美的 Pipeline；相反的，PE 串的 B data 在同一個 Inner Tile 內都是使用 B0~B7，因此我們僅需使用 Stream 的方式傳遞即可。

算完一塊 Inner Tile 後，便會①往 m1 方向處理，②往 n1 方向處理，③往 k 方向累加，直到 Outer Tile 處理完畢。由於 FPGA 的大小最多就是 Outer Tile 這麼大，無法再存取更多資料，因此，算完一個 Outer Tile 就必須先將結果寫回。接著便會往下一個 Outer Tile，也就是④往 m0 方向處理，⑤接續 n0 方向，直到整張矩陣結束。

- **PE Block**

一塊完整的 PE 如圖 25 所示。

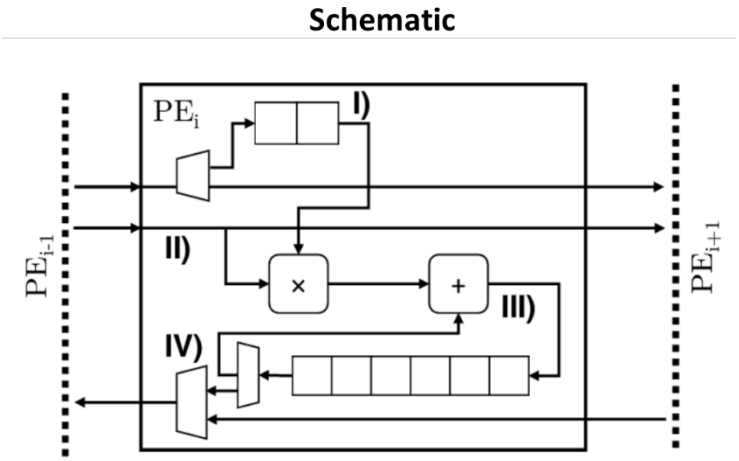


圖 25、 Architecture of a single PE

從圖 25 可以發現，總共標示了 I、II、III、IV 四個 Part，以下針對初始化及這四個部分進行說明。

- **Part 0: Initialize**

在初始化的部分，為了進行 A 矩陣的 double buffer，需要先存取 A 矩陣 k=0 的資料。從圖 23 可以看到 aIn 為 aPipes[pe]，aOut 為 aPipes[pe+1]。在 Initialize 的地方會從 aIn 中抓取此 PE 需要的資料到 aBuffer 中，再將不需要的資料存回 aOut。以 PE0 為例，因為 kOuterTileSizeN (設 512)的關係，PE0 會接收到 index 0~511 的 data。從圖 24 可以看出 PE0 在每一個 Inner Tile 負責計算 A0，故 aBuffer 會存取 index mod 32 為 0 的 index 作為每一個 Inner Tile 要送入 PE0 的對象。至於剩下與 PE0 無關的 data 則會存入 aPipes[1]，並傳送給 PE1，再交由 PE1 挑選需要的 data。圖 26 為 Initialize 的 code，圖 27 為 A 矩陣 index 示意圖，Table 1 列出每種參數所有的 A 矩陣 index。

圖 26 的 code 在 InitializeABuffer_Outer 那一層級加了 pragma LOOP_FLATTEN，原因是在 RTL Implementation 中，進出 for loop 皆需要一個 cycle，因此藉由此 pragma 可以將 nested loop 攤平成一個 single loop，並進一步降低 latency。

```
  // Populate the buffer for the first outer product
InitializeABuffer_Inner:
  for (unsigned n2 = 0; n2 < kInnerTilesN; ++n2) {
    if (locationN < kComputeTilesN - 1) {
      // All but the last processing element
    InitializeABuffer_Outer:
      for (unsigned n1 = 0; n1 < kComputeTilesN - locationN; ++n1) {
        #pragma HLS PIPELINE II=1
        #pragma HLS LOOP_FLATTEN
        const auto read = aIn.Pop();
        if (n1 == 0) {
          aBuffer[n2] = read;
        } else {
          aOut.Push(read);
        }
      }
    } else {
      // Last processing element gets a special case, because Vivado HLS
      // refuses to flatten and pipeline loops with trip count 1
      #pragma HLS PIPELINE II=1
      aBuffer[n2] = aIn.Pop();
    }
  }
```

圖 26、Initialize A Buffer

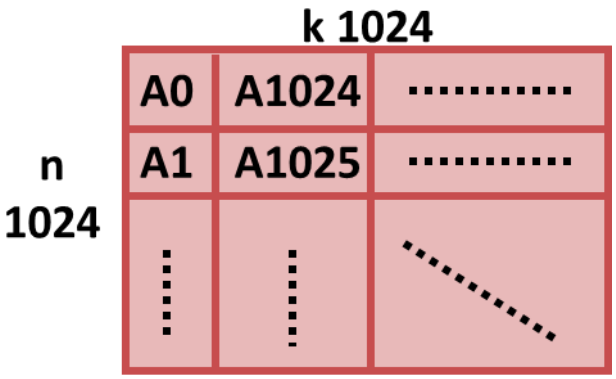


圖 27、A 矩陣 index 示意圖

Table 1

aPipes[0]	0, 1, 2, 3, 4, …, 511
PE = 0	
aBuffer[32]	0, 32, 64, 96, …, 480, 0, 0, …, 0
aPipes[1]	1, 2, 3, …, 31, 33, 34, …, 481, 482, …, 511
PE = 1	
aBuffer[32]	1, 33, 65, 97, …, 481, 0, 0, …, 0
aPipes[2]	2, 3, 4, …, 31, 34, 35, …, 482, 483, …, 511
PE = 2	
aBuffer[32]	2, 34, 66, 98, …, 482, 0, 0, …, 0
aPipes[3]	3, 4, 5, …, 31, 35, 36, …, 483, 484, …, 511
PE = 3	
aBuffer[32]	3, 35, 67, 99, …, 483, 0, 0, …, 0
aPipes[4]	4, 5, 6, …, 31, 36, 37, …, 484, 485, …, 511
PE = 4	
aBuffer[32]	4, 36, 68, 100, …, 484, 0, 0, …, 0
aPipes[5]	5, 6, 7, …, 31, 37, 38, …, 485, 486, …, 511
PE = 5	
aBuffer[32]	5, 37, 69, 101, …, 485, 0, 0, …, 0
aPipes[6]	6, 7, 8, …, 31, 38, 39, …, 486, 487, …, 511
…	
PE = 30	
aBuffer[32]	30, 62, 94, 126, …, 510, 0, 0, …, 0
aPipes[31]	31, 63, 95, 127, …, 511
PE = 31	
aBuffer[32]	31, 63, 95, 127, …, 511

- **Part I: Prepare A**

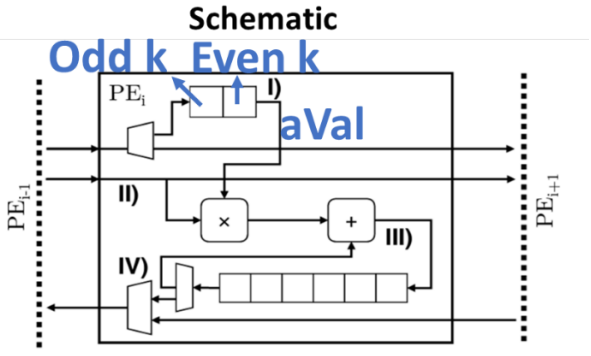


圖 28、Part I 示意圖

圖 29 為 for loop 的層級，次序與圖 24 所示之步驟①~⑤相同。

圖 30 為 A 矩陣 double buffer 的 scheme。藉由此 scheme，PE 會在運算的前先將下一個 k 的資料準備好，就能夠在切換到下一個 k 時，不需要花費大量時間再傳送一次 A data 至所有 PE 串。以 k=0 為例，PE0 的 aBuffer[16:31]會儲存 index 1024+0、1024+32、1024+64…等等共 16 筆資料。從圖 30 及圖 31 可以發現有 aBuffer 寫入又讀取的情況，由於 HLS 不知道 n1 及 k 的值，並且保守的認為 aBuffer[n1+(k%2)?kInnerTilesN:0]跟 aBuffer[n1+(k%2)?0: kInnerTilesN] 有關係。因此，藉由 pragma DEPENDENCE false 聲明兩者不依賴。

```
OuterTile_N:
  for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
  OuterTile_M:
    for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {

      // We do not tile K further, but loop over the entire outer tile here
    Collapse_K:
      for (unsigned k = 0; k < size_k; ++k) {
        // Begin outer tile ----------------------------------------------------

      Pipeline_N:
        for (unsigned n1 = 0; n1 < kInnerTilesN; ++n1) {

        Pipeline_M:
          for (unsigned m1 = 0; m1 < kInnerTilesM; ++m1) {

            // Begin compute tile -----------------------------------------------
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
```

圖 29、for loop hierarchy

```
// Double-buffering scheme. This hijacks the m1-index to perform
// the buffering and forwarding of values for the following outer
// product, required to flatten the K-loop.
if ((n0 < OuterTilesN(size_n) - 1 || m0 < OuterTilesM(size_m) - 1 ||
     k < size_k - 1) &&
    m1 >= locationN           // Start at own index.
    && m1 < kComputeTilesN) {  // Number of PEs in front.
  const auto read = aIn.Pop();
  if (m1 == locationN) {
    // Double buffering
    aBuffer[n1 + (k % 2 == 0 ? kInnerTilesN : 0)] = read;
    #pragma HLS DEPENDENCE variable=aBuffer false
  } else {
    // Without this check, Vivado HLS thinks aOut can be written
    // from the last processing element and fails dataflow
    // checking.
    if (locationN < kComputeTilesN - 1) {
      aOut.Push(read);
    }
  }
}
```

圖 30、double buffer scheme

```
// Double buffering, read from the opposite end of where the buffer
// is being written
const auto aVal = aBuffer[n1 + (k % 2 == 0 ? 0 : kInnerTilesN)];
#pragma HLS DEPENDENCE variable=aBuffer false
```

圖 31、choose aVal

圖 31 為根據 k 為偶數或奇數選擇 aVal 的 code。根據 double buffer 的 scheme，我們可以將 aBuffer 的儲存狀況畫成圖 32。當 k 為偶數時，就會根據現階段的 n1 拿取 aBuffer[n1]的 data；若 k 為奇數，則會拿取 aBuffer[16+n1]的 data。以 k=0 為例，aBuffer[0:15]會儲存 A 矩陣 index 為 0、32、64…等等 16 筆資料，並且由於 double buffer 的關係，aBuffer[16:31]會儲存 A 矩陣 index 為 1024、1056、1088…等 16 筆資料，也就是在 k=1 的資料。其餘以此類推。

PE0	Even k					Odd k				
n1	0	1	2	.....	15	16	17	18	.....	31
aBuffer	0	32	64	.....	480	1024	1056 (1024+32)	1088 (1024+64)	.....	1504 (1024+480)
aPipes[1]	1 ≀ 31	33 ≀ 63	65 ≀ 95	.....	481 ≀ 511	1025 ≀ 1055	1057 ≀ 1087	1089 ≀ 1119	.....	1505 ≀ 1535
PE1										
aBuffer	1	33	65	.....	481	1025	1057 (1024+32)	1089 (1024+64)	.....	1505 (1024+480)
aPipes[2]	2 ≀ 31	34 ≀ 63	66 ≀ 95	.....	482 ≀ 511	1026 ≀ 1055	1058 ≀ 1087	1090 ≀ 1119	.....	1506 ≀ 1535

圖 32、aBuffer after double buffer scheme

- **Part II: Prepare B**

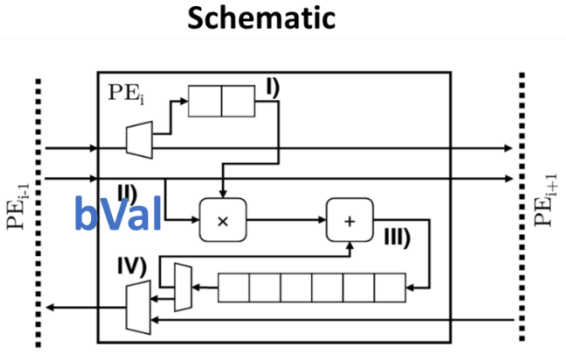


圖 33、Part II 示意圖

由圖 24 我們可以發現，PE 串的 B 都是一樣的。僅有在 PE 計算完一個 Inner Tile 需要移動至下一個 Inner Tile 時，B 才會有所改變。

因此，我們在傳遞 B 的時候不需要像 A 一樣 buffer 住很多東西，相反的，僅須利用 Stream 的方式就可以傳遞 B 的資料。

```
const auto bVal = bIn.Pop();
if (locationN < kComputeTilesN - 1) {
  bOut.Push(bVal);
}
```

圖 34、choose bVal

bIn 的 size 為 kComputeTileSizeM (設 8)，因此 bIn Pop 一次，bVal 就會得到 8 筆 data。從圖 23 可以看到，由於下一個 PE 的 bPipes 來自於前一個 PE 的 bPipes，但又不需要進行改變，因此可以直接將使用到的 bVal 存回 bOut，就能直接傳至下一個 PE 的 input bPipes。

- **Part III: Multiply and Add**

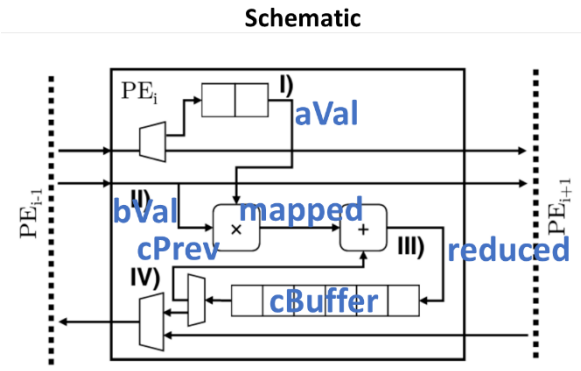


圖 35、Part III

在 Part III 主要會實現將 Part I 及 Part II 得到的 aVal、bVal 進行矩陣乘法，並且如果 k 不為 0 時，要與前面的 k 計算完的 cPrev 進行累加。在這部分主要會分為兩個 for loop，並且由於 PE 內部的 compute unit 皆為平行運算的，因此兩個 for loop 皆會進行 UNROLL。

圖 36、圖 37 為 PE 的 code，即對應到圖 24 右側的一塊 PE。如果 k>0，則 cPrev 就是存在 cBuffer 內的資料；如果 k 為 0，則 cPrev 初始化為 0。圖 37 的 OperatorMap 即為 hlslib 的 Multiply，OperatorReduced 為 hlslib 的 Add，故 mapped 即為 A0 與 B0、B1、B2…B7 進行乘法後的 8 個結果，reduced 則為進行累加後的結果。如果這 8 個結果的所在位置沒有超過矩陣的大小的話，就會將累加過後的 8 筆 data 存入 cBuffer 中；如果有超過的話，就不會進行累加，並把之前的資料存回 cBuffer 中。

```
Unroll_N:
  for (unsigned n2 = 0; n2 < kComputeTileSizeN; ++n2) {
    #pragma HLS UNROLL

    const bool inBoundsN = ((n0 * kInnerTilesN * kComputeTileSizeN +
                             n1 * kComputeTileSizeN + n2) < size_n);

    ComputePackM_t cStore;
    const auto cPrev = (k > 0)
                       ? cBuffer[n1 * kInnerTilesM + m1][n2]
                       : ComputePackM_t(static_cast<Data_t>(0));
```

圖 36、PE N-wise

```
Unroll_M:
  for (unsigned m2 = 0; m2 < kComputeTileSizeM; ++m2) {
    #pragma HLS UNROLL

    const bool inBoundsM = ((m0 * kInnerTilesM * kComputeTileSizeM +
                             m1 * kComputeTileSizeM + m2) < size_m);

    const bool inBounds = inBoundsN && inBoundsM;

    const auto mapped = OperatorMap::Apply(aVal[n2], bVal[m2]);
    MM_MULT_RESOURCE_PRAGMA(mapped);
    const auto prev = cPrev[m2];

    const auto reduced = OperatorReduce::Apply(prev, mapped);
    MM_ADD_RESOURCE_PRAGMA(reduced);
    // If out of bounds, propagate the existing value instead of
    // storing the newly computed value
    cStore[m2] = inBounds ? reduced : prev;
    #pragma HLS DEPENDENCE variable=cBuffer false
  }

  cBuffer[n1 * kInnerTilesM + m1][n2] = cStore;
}
```

圖 37、PE M-wise

- **Part IV: Write Back**

圖 38 為寫回 C 的 dataflow，當一個 Outer Tile 計算完後，就會從最尾端的 PE 將結果回傳到最前端。圖 39 為 cBuffer 的示意圖，cBuffer 內含 kInnerTilesN x kInnerTileM (設 16*64)筆資料，而每一筆資料內皆有 8 筆 data，分別為 c0`~c7。故從圖 41 的 code 可以看到，cOut.Push(cBuffer)一次就是 Push 一個 C0，也就是 c0～c7。

圖 41 是一個被 flatten 過的 for loop，且有 PIPELINE II=1 的 pragma，故可以將 cOut 的儲存情況畫成圖 40。首先，1.) PE1 會儲存 C0 至 C63 的 cBuffer 至 cOut 中，2.) PE2 儲存 C64 至 C127 的 cBuffer，3.) …，一直到 32.) PE32 儲存 C1984 至 C2047 的 cBuffer，這 32 件事會進行 Pipeline。接著，當 PE1 完成 C0 至 C63 的儲存後，

便會拿取在 cPipes[1]的資料，也就是 PE2 的 cOut，也就是 C64，並接續拿取來自 PE2 的結果。同樣的，PE2 也會拿取 PE3 的結果，一直到 PE31 拿取 PE32 的結果。故在一個 Inner Tile 內，PE32 共儲存 64*1 次，PE31 共儲存 64*2 次，…，PE1 共儲存 64*32 次。當平行的 Inner Tile 都結束存取後，就會往下一排 Inner Tile 移動，並重複以上的 Pipeline。直到整個 Outer Tile 被存完，就會移動至下一個 Outer Tile，並重複 Part I 至 Part IV。

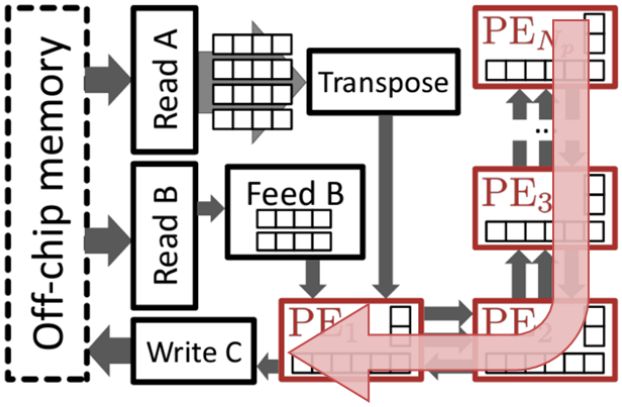


圖 38、Part IV flow

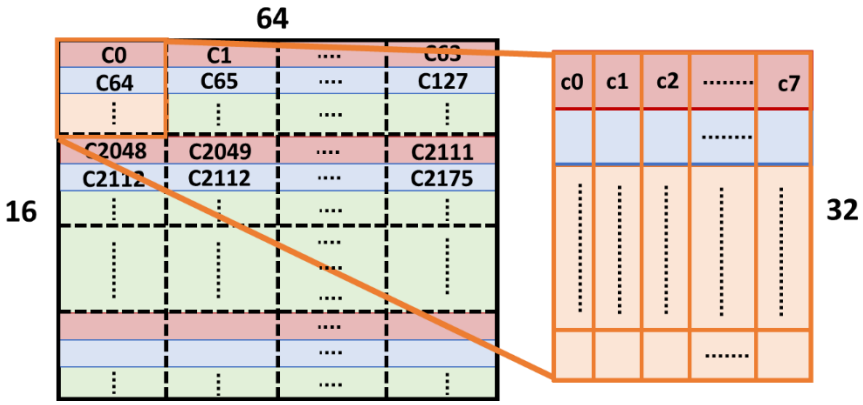


圖 39、cBuffer 示意圖

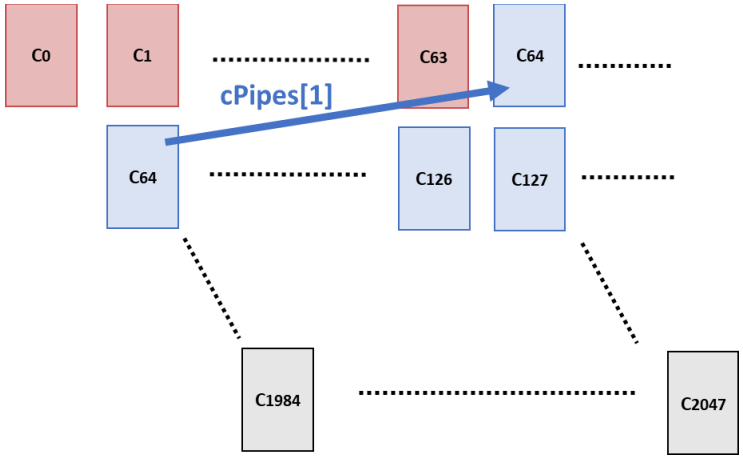


圖 40、cOut.Pop() Pipeline 示意圖

```
WriteC_Flattened:
  for (unsigned i = 0; i < writeFlattened; ++i) {
    #pragma HLS PIPELINE II=1
    if (inner < kComputeTileSizeN * kInnerTilesM) {
      cOut.Push(cBuffer[n1 * kInnerTilesM + m1][n2]);
      if (m1 == kInnerTilesM - 1) {
        m1 = 0;
        if (n2 == kComputeTileSizeN - 1) {
          n2 = 0;
        } else {
          ++n2;
        }
      } else {
        ++m1;
      }
    } else {
      if (locationN < kComputeTilesN - 1) {
        cOut.Push(cIn.Pop());
      }
    }
    if (inner == writeFlattenedInner - 1) {
      inner = 0;
      ++n1;
    } else {
      ++inner;
    }
  }
```

圖 41、Write back code

IV. 部署到 U50 的實際測試結果

```
[g109061806@ic21 build]$ ./RunHardware.exe 1024 1024 1024 hw
Initializing host memory... Done.
Initializing OpenCL context...
Programming device...
Initializing device memory...
Copying memory to device...
Creating kernel...
Executing kernel...
Kernel executed in 0.0147058 seconds, corresponding to a performance of 146.029 GOp/s.
Copying back result...
Running reference implementation...
WARNING: BLAS not available, so I'm falling back on a naive implementation. This will take a long time for large matrix sizes.
Verifying result...
Successfully verified.
```

圖 42、Deployment Result on U50 board

圖 42 則顯示了我們部署到 U50 板子實際測試的結果，部屬的資料型態是 float，並且每次可以做 512*512 大小的矩陣 tile，實際測試兩個 1024*1024 的矩陣在上面運作算出來的結果是完全一樣的，這可以證明這個矩陣乘法加速器的正確性。

V. 實驗分析

在 Memory-related 的部份說明了 Read A、Transpose、Read B、Feed B、Write C 等 block 的運作方式，Read A、Transpose、Read B、Feed B 四個 block 會同時運作，同時讀取矩陣 A 和矩陣 B 的數值，Read A 會以 512 個 cycle 讀取圖 43 中一個藍色框框的矩陣 A，也就是一個 cycle 可以讀取一列中的 8 個數值，但是 Transpose 在經過 512 個 cycle 後，只能處理紅色框框的數值，因此在送入 PE 前，矩陣 A 讀取的 bottleneck 為 Transpose cycle 數，總 cycle 數為 512×128×8×4=2097152。Read B、Feed B 兩者會同時運作，將 Read B 讀出的矩陣 B 數值送入 Feed B block 中儲存，因此兩者所需的 cycle 數相同，為 2×2×1024×64=262144。Write C 則會把算完的結果存回 off chip memory 中，所需的 cycle 數為 2×2×512×64=131072。

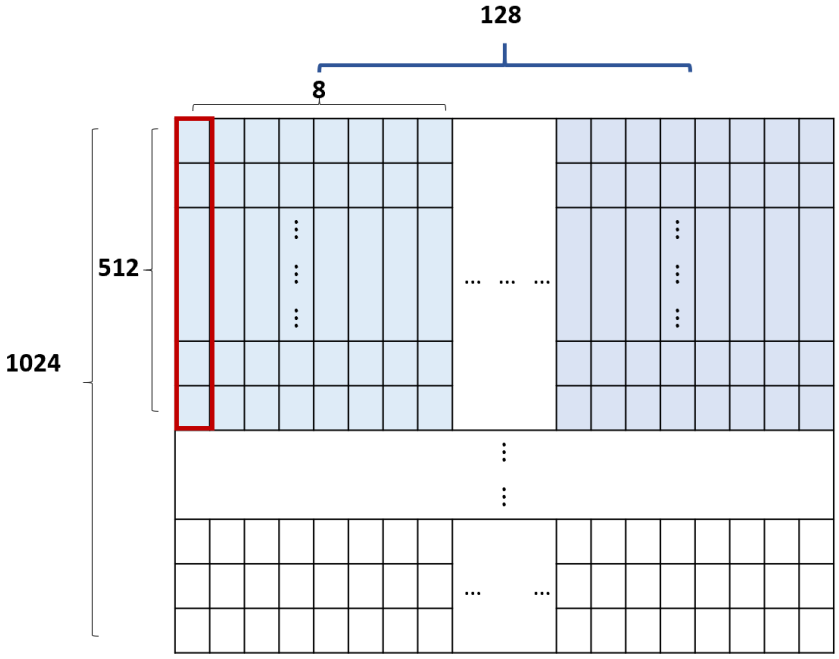


圖 43、Read A、Transpose 讀取 cycle 示意圖

由 III 可以得知此設計會將矩陣切成多個 Outer tile，每一個 Outer tile 的大小為 512*512，也就是 C 矩陣的 buffer 大小。以計算一個 Outer tile 為單位，得 A 矩陣需要存取 512*1024 筆資料；由於不同的 Outer tile 會重複使用到 B 矩陣，因此 B 矩陣需要存取 1024*1024 筆資料。最後，一筆資料為 8 Byte，Total Internal Buffer Size 列於 Table 2：

Table 2

Matrix	Buffer (#)	Size (MB)
A	512*1024	4.194304
B	1024*1024	8.388608
C	512*512	2.097152
Total		14.680064

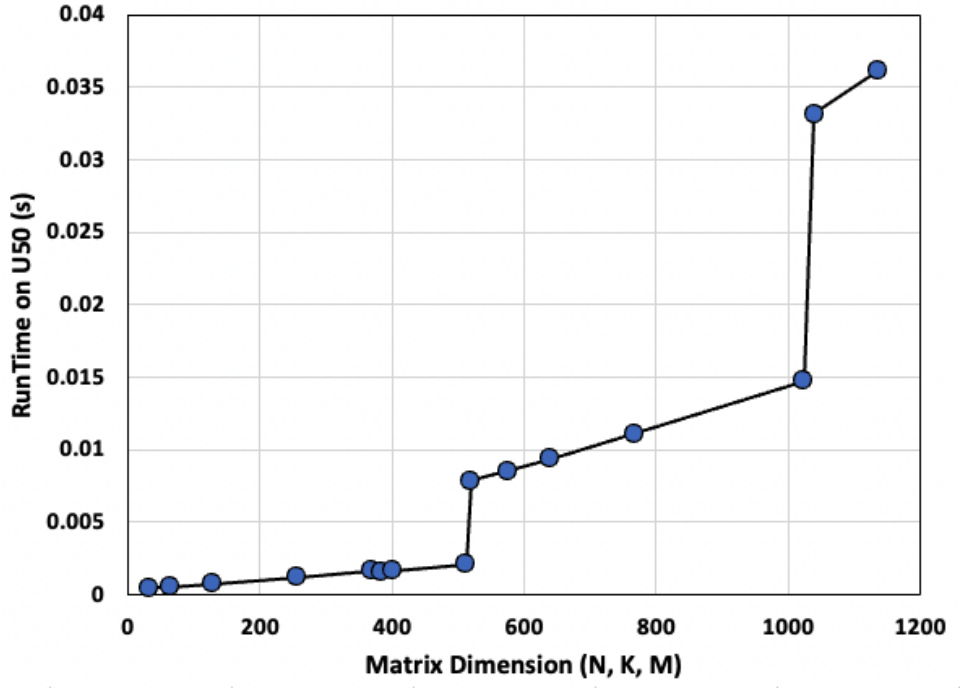


圖 44、實際測試 runtime 結果

我們也實際跑了許多分析來確認對架構的理解正確，圖 44 展示了時間對應矩陣大小的分析，有幾件事情值得注意：

1. 這個矩陣乘法加速器可以支援許多不同的矩陣大小，從 100 以下到 1024 以上都可以支援，且基本上運算時間和矩陣大小成正比。
2. 在矩陣大小超過 512 和 1024 分別有一個很大的斷層，我們認為這是加速器將矩陣切成 512*512 的 tile 造成之影響，當超過 512 時 pipeline 會需要把舊資料先寫出到 off-chip DRAM，因此造成時間上的 overheads，這也是一個可以優化這個加速器的未來方向。
3. 矩陣乘法的複雜度是 O(N^3)，因此矩陣越大、和時間的關係會斜率變大。

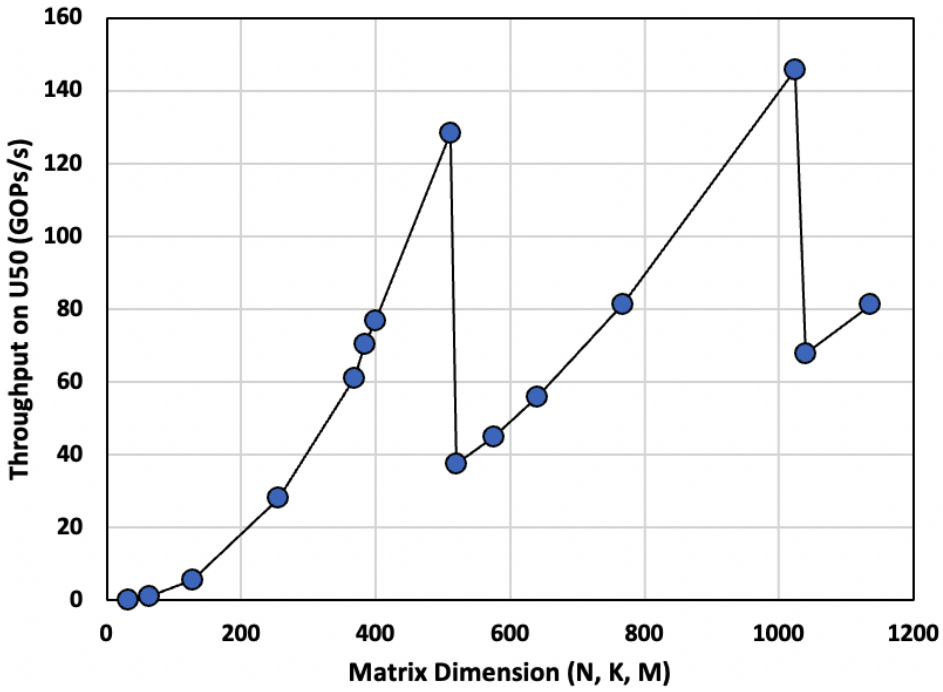


圖 45、實際測試 throughput 結果

圖 45 則展示了 throughput 對應矩陣大小的分析，有幾件事情值得注意：

1. 在矩陣較小時，加速器的 throughput 很低，這是因為加速器的乘法平行度是 8*32，沒有完全用滿這個值就會導致 underutilized。
2. 另外，當矩陣的 K dimension 增加的時候，因為 partial sum 資料不用輸出，所以運算會成為時間瓶頸，不斷增加會接近他運算上的理論 peak performance。
3. 當矩陣大小跨過 512，便會需要把某個 outer tile 寫回 off-chip memory，造成時間上的 overheads。

VI. 結論

我們期末專題分析了如何使用 HLS 來設計一個有競爭力的矩陣乘法加速器，並對他的架構進行分析、移植到實際板子上執行、並分析數據來確認對架構理解的正確性、並找到可以優化的地方。

從這個專題中我們學會如何將一篇論文對應的 HLS 加速器架構進行剖析、理解如何做出需要的架構，以及將它實際執行在 U50 板子上來進一步找到 insight。這個專題幫助我們理解如何用 HLS 設計具有競爭力的加速器。