Flexible Matrix Multiplication Accelerator with High-Level synthesis

Application Acceleration with High-Level Synthesis Final Project
github repo link : https://github.com/sssh311318/HLS_FinalProject
吳承哲 許睿哲 石思宇

Introduction

Matrix operations are extensively utilized in various fields such as Engineering, science, and mathematics, making them one of the most frequently used operation types. They play a significant role in process control, real-time image processing, real-time digital signal processing, and network control systems [1]. Similar to other algorithms, scientists and engineers aim to minimize data movement to enhance performance, especially considering the growing technology gap between computation and memory speeds [2]. Recently, there has been considerable research focused on improving the computation speed of the Matrix Multiplication operation, which involves multiplying two input matrices to generate a third matrix. Scholars have investigated different approaches, with some focusing on multiprocessor parallel computing to enhance operation speed, while others concentrate on achieving parallel computing to optimize overall computational performance. In this project, our objective is to replicate the design of a previous year's final project completed by three students [3]. Subsequently, we aim to identify a justifiable cause for poor performance when the size of the input matrix does not align as a multiple of the hardware compute tile size; also analyze any option of optimization. The report's different parts are as follows: 1. System Architecture 2. Block Diagram 3. Code Description 4. Experimental Results and Analysis 6. Conclusion.

1. System Architecture

In parallel computing, systolic array architecture is used to efficiently implement certain algorithms, particularly those involving regular, repetitive computations such as matrix multiplication and convolution. It consists of a grid of processing elements (PEs) connected in a regular pattern, often arranged in rows and columns. The PEs in a systolic array are tightly coupled and communicate with their neighboring PEs through a set of dedicated data paths. In the paper the project is based on, the architecture for a 2D grid is shown in Fig 1 was to be chosen; however, the authors of the project were opted to efficiently map their system onto various interconnect structures, including non-uniform and hierarchical setups, as well as multiple-chiplet FPGAs or multiple FPGAs, they have the option to collapse the 2D array of processing elements (PEs) into a 1D array[2], hence Fig 1.

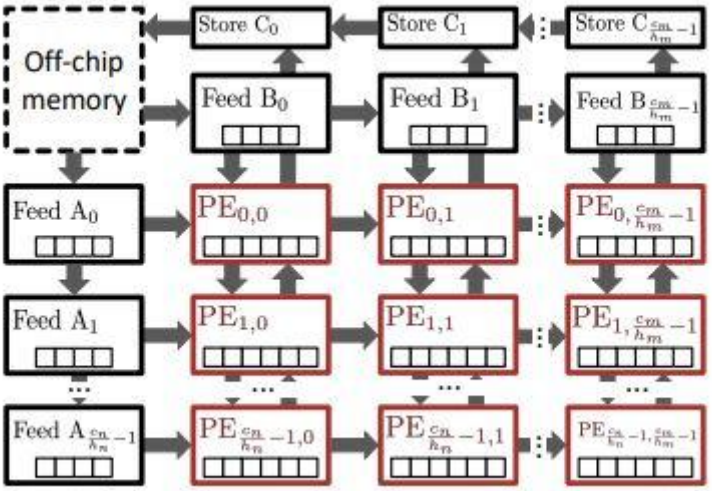


Fig 1. Computation in 2D grid(from [2])

Fig 1 shows the architecture of the system which consists of the PL side of the FPGA, where the kernel will be implemented, a High-bandwidth Memory as well as the host responsible to start the kernel computing and verify if the operation results are correct. Below show the steps taken when the system is running:

a. Initialization of OpenCL
b. Read matrices A & B from host to kernel device
c. Execute Kernel
d. Write product matrix C from kernel device to host
e. Computation of Golden matrix on host
f. Comparison of the results

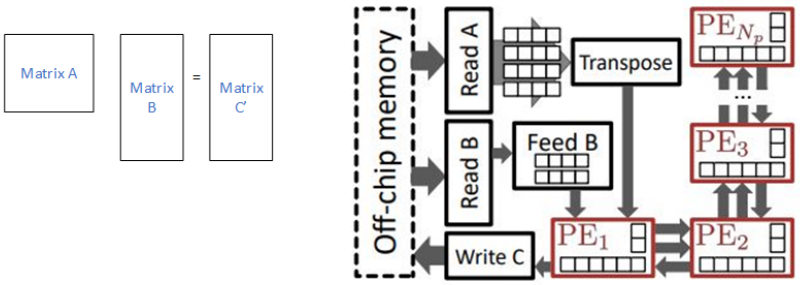


Fig 2. System Architecture(right side from [2])

2. Block Diagram Decomposition

Fig 2 presents a detailed depiction of the kernel architecture. The diagram showcases the arrangement of components within the kernel. On the left side, you can observe the input matrices denoted as A and B, while the output matrix is represented as C. The kernel is partitioned into multiple outer tiles, with their respective sizes indicated as kOuterTileSizeN and kOuterTileSizeM. These outer tiles are further subdivided into inner tiles, which possess sizes referenced as kInnerTileSizeN and kInnerTilesM.

Moreover, each inner tile is divided into multiple processing elements (PEs). The PEs are one-dimensional and possess a size denoted as kComputeTileSizeM, defined as compute units.

further explanation

The system aims to leverage parallelism during matrix multiplication. To determine the appropriate size of the outer tiles, the number of processing elements, and the number of tiles to execute in parallel, the cmake command is utilized, as demonstrated in the example:

```
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
```

Listing 1. Cmake command

Here is the breakdown of the cmake command:

DMM_DATA_TYPE: Specifies the data type of the input/output matrices, such as float, int, double, etc.

DMM_PARALLELISM_N: Represents the number of processing elements (PEs) within an inner tile that will run in parallel. This corresponds to kComputeTilesN in Figure 2.

DMM_PARALLELISM_M: Indicates the size of a processing element (PE), or the number of compute units it possesses. This aligns with kComputeTileSizeM in Figure 2.

DMM_MEMORY_TILE_SIZE_N: Refers to the size of an outer tile in the N dimension, denoted as kOuterTileSizeN.

DMM_MEMORY_TILE_SIZE_M: Denotes the size of an outer tile in the M dimension, represented as kOuterTileSizeM.

kInnerTilesN is obtained by dividing the kOuterTileSizeN by kComputeTileSizeM.
kInnerTilesM is obtained by dividing the kOuterTileSizeM by kComputeTilesN.

By adjusting these parameters in the cmake command, the system can effectively control the parallelism, tile sizes, and processing elements to optimize the matrix multiplication operation.

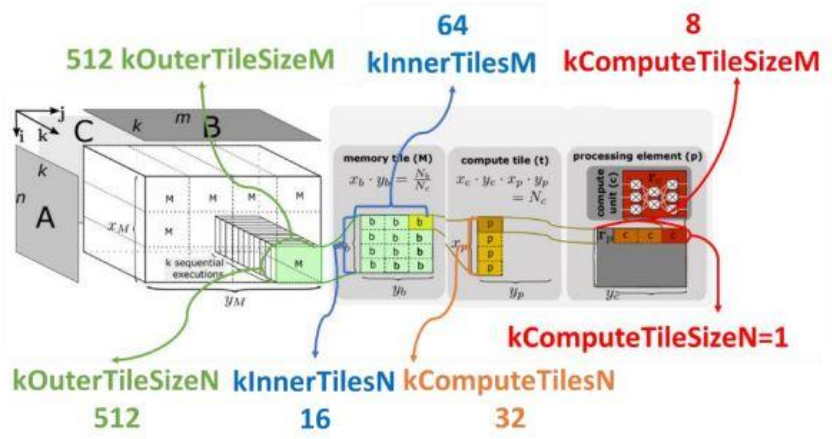


Fig 3. Block Diagram Decomposition with Parameters(from [2])

Fig 4 depicts the architecture of a single PE block. (I) Prepare matrix A; (II) Prepare B; (III) Multiply and Add; (IV) Write back.

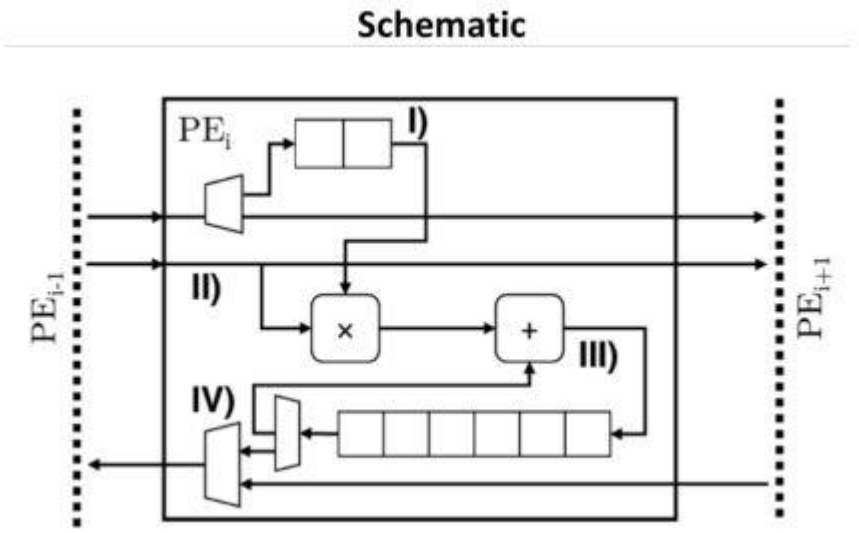
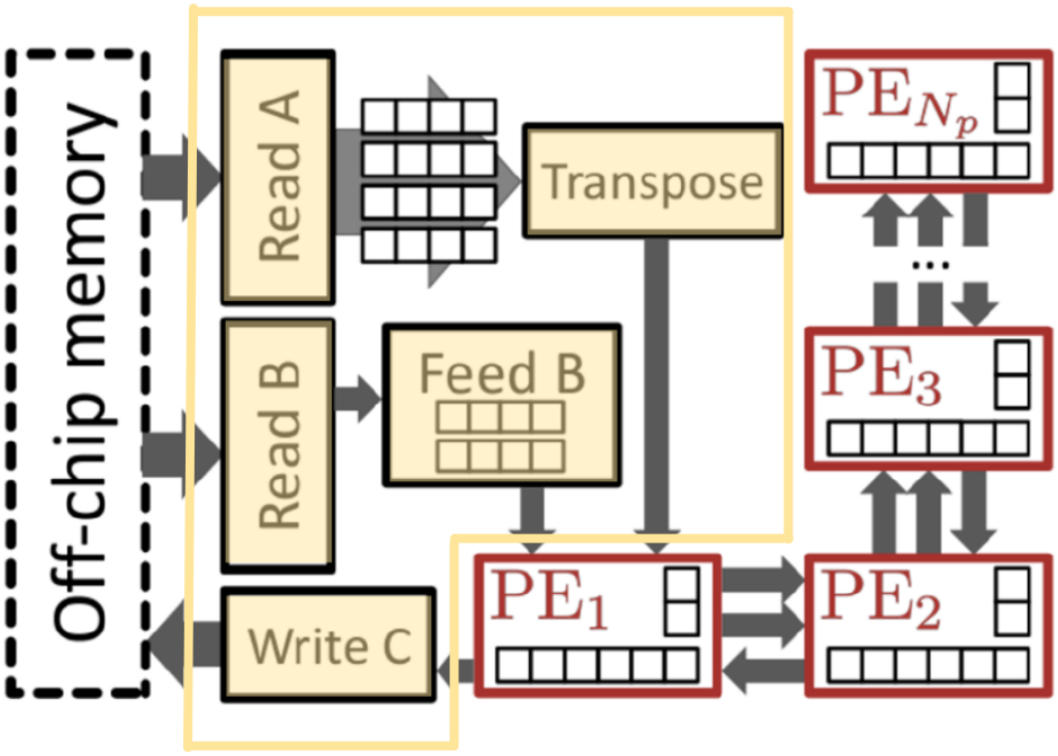


Fig 4. Architecture of a single PE

3. Code Description

這裡分成兩個Block Data Processing Block(黃色區域)跟Computation Block(紅色區域), Data Processing Block包含Read A、Transpose、Read B、Feed B、Write C這些sub-blocks；Computation Block則是包括全部的PE。



- **Data Processing Block**
  - **ReadA()**

    OuterTilesN 判斷Input matrix 為MM_Memory_Tile_SizeN的幾倍，size_k為Input matrix k方向的size，kTranspose則是可調的參數(MM_TRANSPOSER_WIDTHBytes/sizeof(Data_t))，kInnerTilesN則為Fig 5上的x_M。

    可以從Fig 5對應到Lisitng2 ，以Fig 5為例，這裡的OuterTilesN =3, OuterTilesM =4, size_k/kTransposeWidth = 11， kInnerTilesN為3。ReaA會先讀出藍色區域(n1)並送進PE中，依序讀出紅色區域，並送進PE中因為這裡的矩陣大小與，Outer loop與Second-level loop的trip count則分別為3跟4。

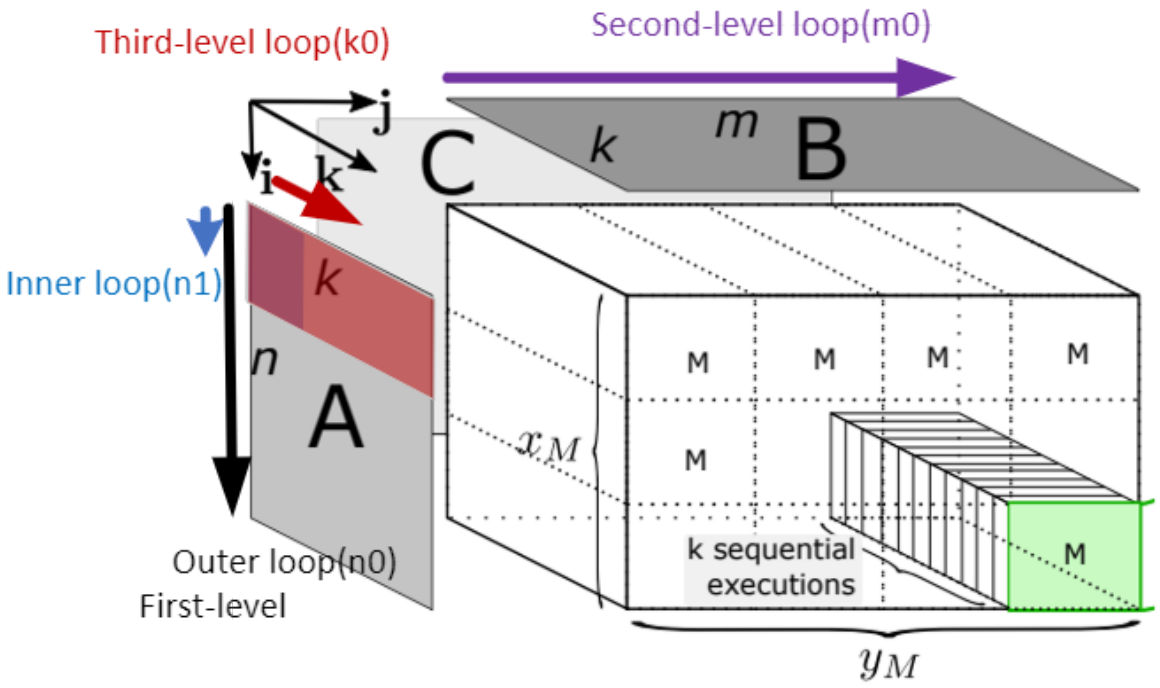


Fig 5. Read A示意圖

```
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);
        }
      }
    }
```

Listing 2. ReadA block

_ReadAInnerLoop()、_ReadAInner()、IndexA() 這裡使用#pragma HLS INLINE 可以減少function call 的overhead，也可以對loop做optimize。
這裡以Listing 2 與Listing 3對照Fig 6，由於KtransposeWidth可以自己設定，可由多個data組成的，有可能會遇到kTransposeWidth=kMemoryWidth，會造成trip count=1，無法進行pipeline，因此使用_ReadAInnerLoop<1>來解決。
這裡已Fig 6 為例，若KtransposeWidth/kMemoryWidth=1，KTransposeWidth=4，_ReadAInnerLoop的KInnerTilesN為PE的數量，它先前有在cmake就定義了，kMemoryWidthK則可看FPGA來做決定，這裡假設為1。
_ReadAInnerLoop block的KInnerTilesN所在的loop為Fig 6的step1。
_ReadAInnerLoop block則是因為unroll loop，asplit[0]、asplit[1]...asplit[3]可以同時個別讀取1.2.3.4。之後跟著step1.step2...來讀取，step4則是等matrixB讀取完，PE運算完，在繼續做step5。
而ReadAIndexA block則是計算出矩陣A對應的address來存取。

```
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);
  }
}
```

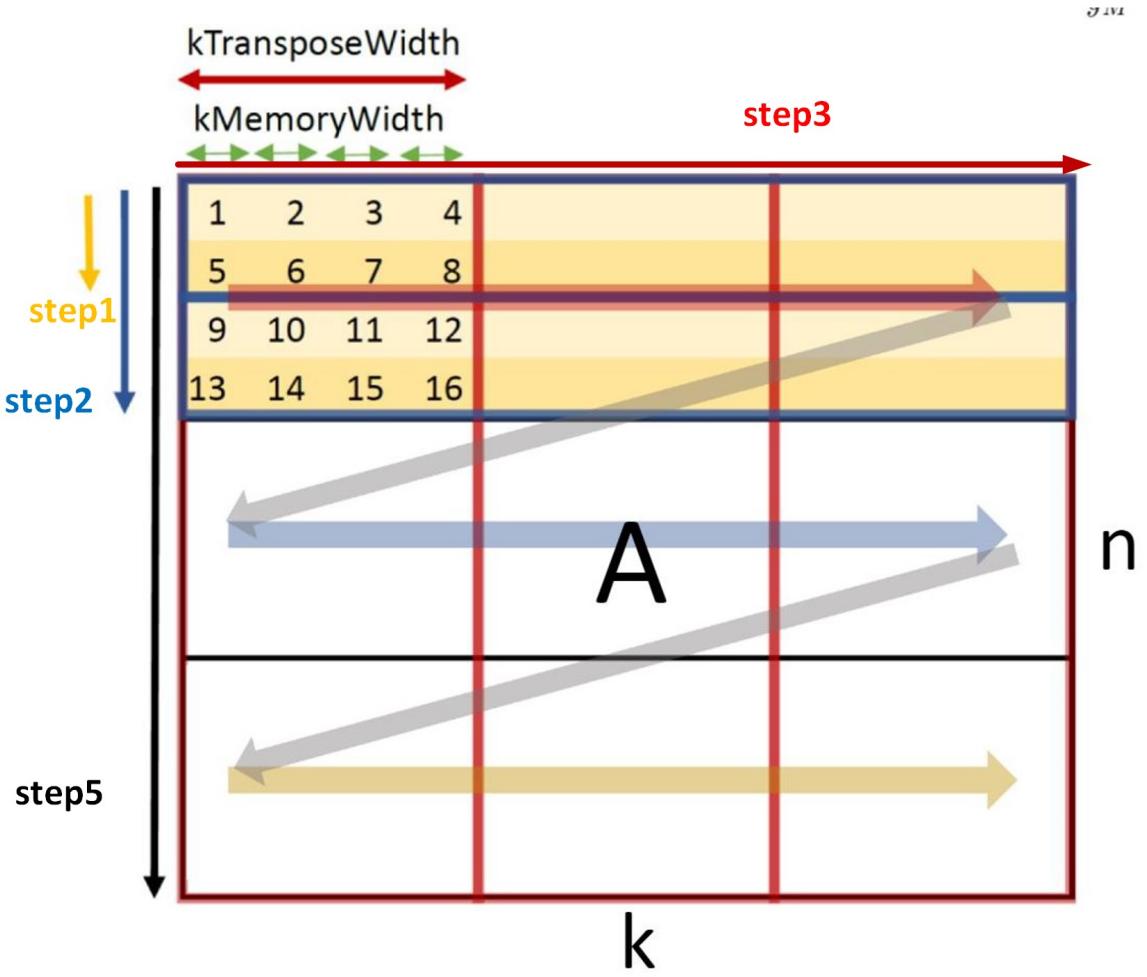
Listing 3. _ReadAInnerLoop block

```
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]);
  }
}
```

Listing 4. _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;
}
```

Listing5. IndexA block



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

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

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

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

Fig 6. ReadA asplit存取流程示意圖

- **TransposeA()**

_TransposeAInner block會根據K與來讀取各個PE中asplit中的值，這裡因為kComputeTileSizeN已經定義為1來做運算(coupute unit只有x方向 Fig 3的Processing element)，因此只需要放一個值到toKernel。

```
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);
      }
    }
  }
}
```

Listing6. 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);
  }
}
```

Listing 7. TransposeAInner block

- **ReadB()**

在ReadB中，會依照Index給的address來讀取，kOuterTileSizeMMemory上面有講過，可以自己設定及調整，ReadB則從memory讀取值進conver Bitsidth block中在進去FeedB()及PE中。

```
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)]);
        }
      }
    }
  }
}
```

Listing 8. 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;
}
```

Listing 9. IndexB block

- **FeedB()**
  讀完Matrix B的值之後，儲存在FeedB中的"buffer[]"大小為32(kInnerTiles64)，如Fig 7，

```
  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);
```

Listing 10. FeedB block

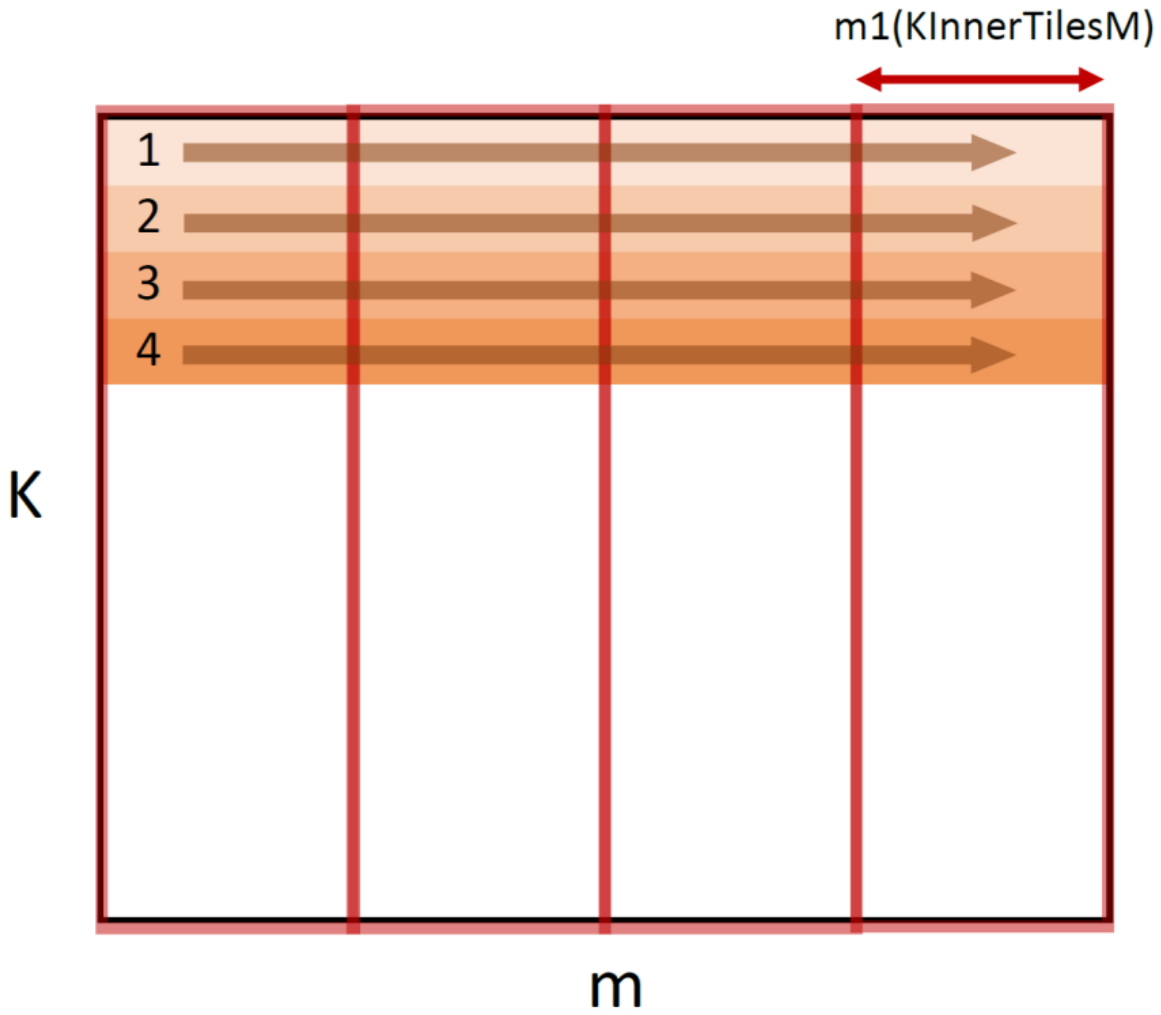


Fig 7. 矩陣B讀取示意圖

- **WriteC()**

  WriteC block則是利用Index計算出來的 address，將 output結果存回memory中對應的位址中，大小剛好是kouterTileSizeN x kouterTileSizeM。

```
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;
          }
        }
      }
```

Listing 11. 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;
}
```

Listing 12. IndexC blo

o Convert Bitwidth
從記憶體讀出的bitwidth與PE輸入的不一樣，需要矩陣A與矩陣B都需透過Convert Bitwidth block來讓bitwith對齊，在我們跑的例子中，kWidth為8。stream的bitwith變為原來的8倍。

```
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);
  }
}
```

ck

Listing 13. Convert Bitwidth block

- **computation**
  - **Dataflow**

下圖為整個架構的dataflow，outer Tile為亮綠色框框。橘色框框裡的紅色藍色則為inner tile，裡面放著PE及PE裡的CU，因為作者將此架構變為1-d，compute tile 裡的PE為y方向，CU為x方向，不同於systolic array，這裡的y方向的cu，輸入都為相同，而不是一個傳一個。

算完一塊Inner Tile後，便會 做step1，往 m1方向處理，依序做其他的 方向n1.k.，在算完一個outer tile後就寫回去，在往m0.n0方向做m運算

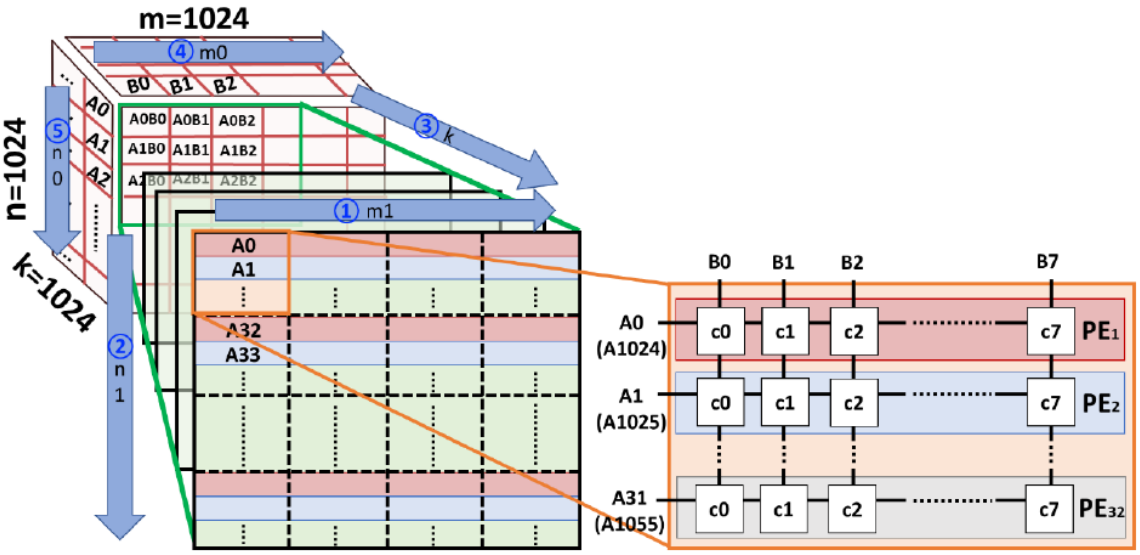


Fig 8. overall Dataflow

PE block

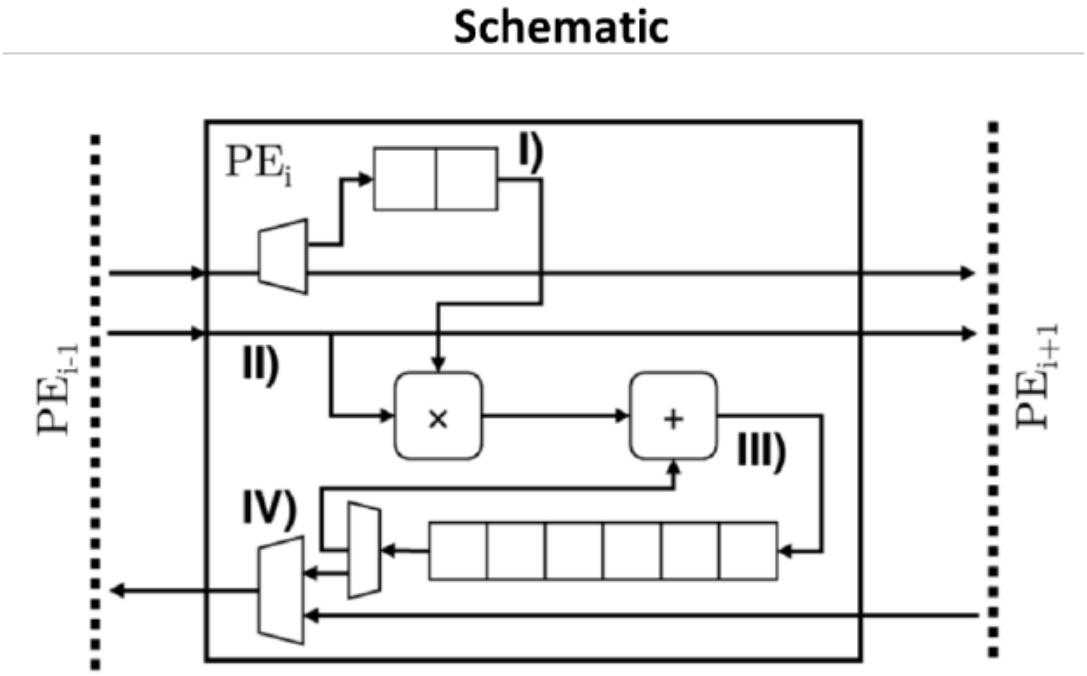


Fig 9. PE schematic

PE總共分成5個部分InitializeA. PrepareA. PrepareB. Multiply and ADD. WriteBack

(1)IntializeA:

這裡是為了進行A矩陣的double buffer，先從aIn讀取此PE所需的資料到aBuffer中，再將不需要的資料送回aOut。

```
  // 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();
    }
  }
```

Listing 14. Initilize A Buffer block

(2)PrepareA :
Listing15 這裡的for loop依照Fig 8來做，下半段則為double buffer的部分，因為這裡的PE為1D，若每個K都需要重傳值進PE，那會需要很久的時間(k x PE數量個cycle)，所以這裡的PE會先準備好下一個k值的資料，等到下一個K值時就可以使用如Fig10。

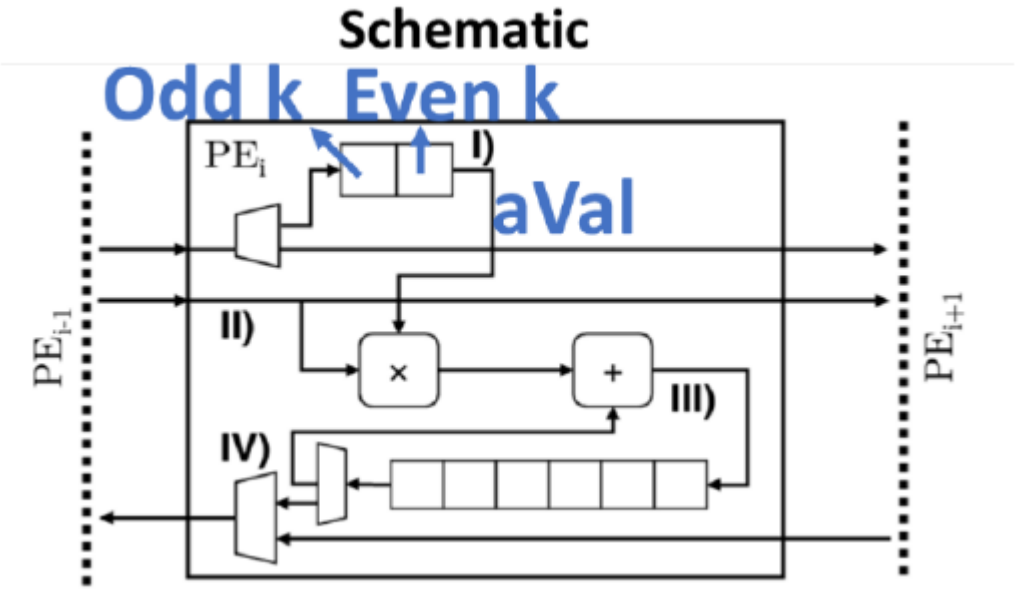


Fig 10. part I示意圖

```
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
```

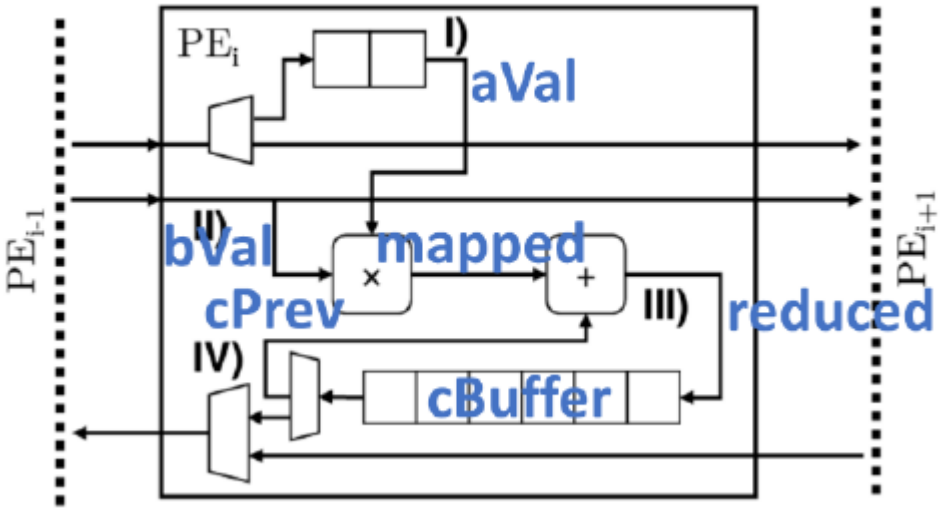
```
// Double-buffering scheme. This hijacks the m1-index to perform
// the buffering and forwarding of values for the following outer
// product, required to flatten the K-loop.
if ((n0 < OuterTilesN(size_n) - 1 || m0 < OuterTilesM(size_m) - 1 ||
     k < size_k - 1) &&
    m1 >= locationN            // 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);
    }
  }
}
```

(3)prepare B:
因為PE的輸入都是一樣的，只有全部PE算完後，才會讀取新的B進來，這裡的bIn大小為8個data，kComputeTileSize為8，這個PE做完之後再將bVal傳至下一個PE。

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

(4)Multiply and Add:
如下圖，這裡將aVal各乘上bVal的8個數字，得到一個1x8的矩陣，並存進cbuffer中，且unroll每個compute unit，

Schematic


PE_i
I)
aVal
PE_i-1
II)
bVal
mapped
cPrev
×
+
III)
reduced
PE_i+1
IV)
cBuffer

```
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));

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

(5) Write Back

下圖的code則為writeBack vlock 當一個outer tile計算完後，就會從最末端的PE傳至最前端的PE，且每筆都有8個data(dataCout.Push(cBuffer))。

```
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;
    }
  }
```

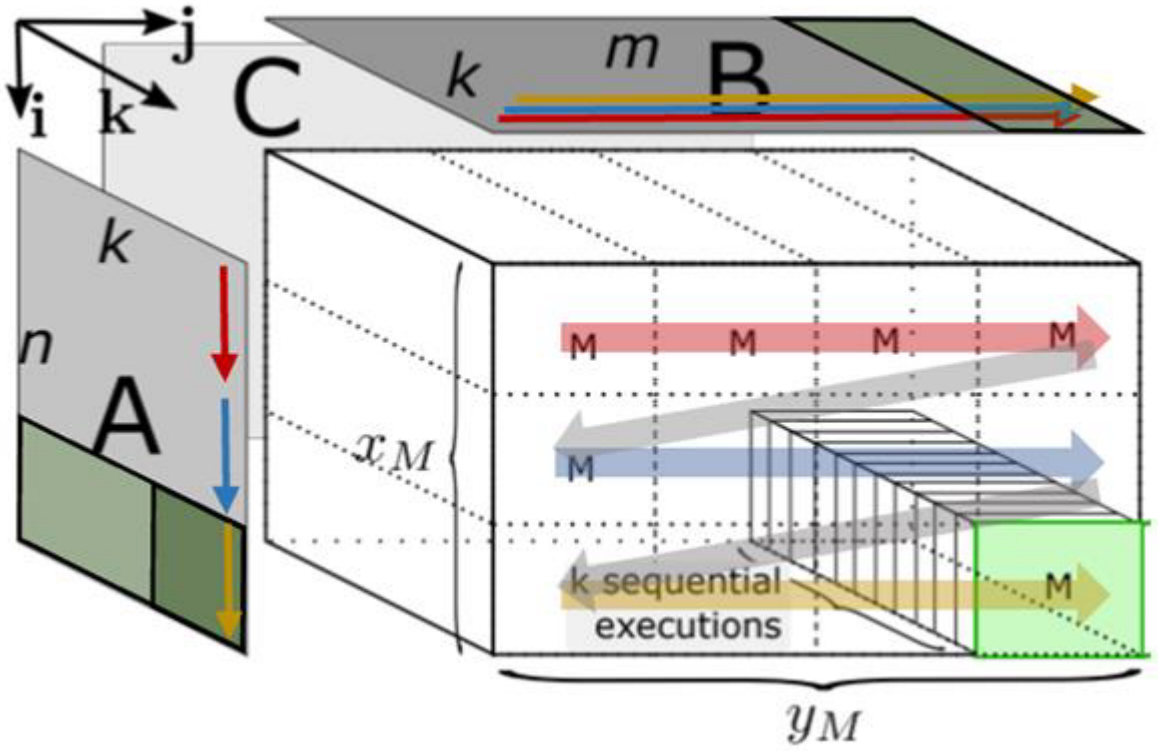


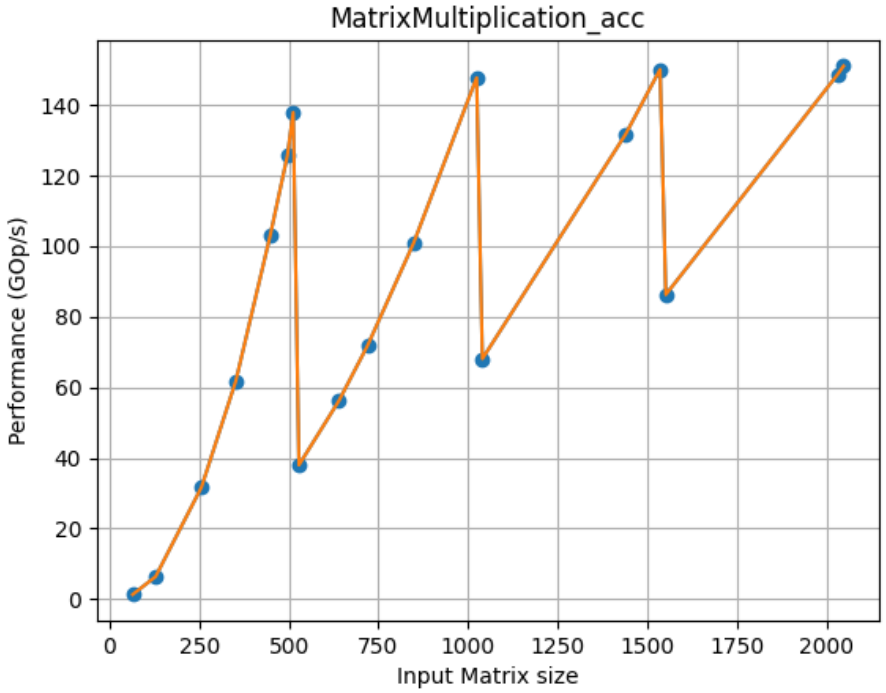
Fig 5. 矩陣 A 、B與output計算對應圖

4. Experimental Result and Analysis

一開始我們先按照github上的步驟執行，但是因為vitis的版本不同，我們要使用的是2021.1的版本，還有platform的問題，後來在討論區發問並解決上述問題，於是我們成功build hardware並得到下面測試結果：

```
02.CtINEm@HLS02:~/m110061608/final/gemm_hls/build$ ./RunHardware.exe 1024 1024 1024
Initializing host memory... Done.
Initializing OpenCL context...
Programming device...
Initializing device memory...
Copying memory to device...
Creating kernel...
Executing kernel...
Kernel executed in 0.0145308 seconds, corresponding to a performance of 147.789 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.
02.CtINEm@HLS02:~/m110061608/final/gemm_hls/build$
```

之後就著手進行當input matrix size接近kernel size(*512*512)時，perforamance會drop的情形，如下圖：



我們在測試的時候，發現當input size超出kernel size時就會drop，然後隨著input size慢慢增大performance就會變高，一直到超過kernel size的倍數就會drop下來，所以我們決定跑hardware emulation看waveform來了解到底是發生什麼事情。

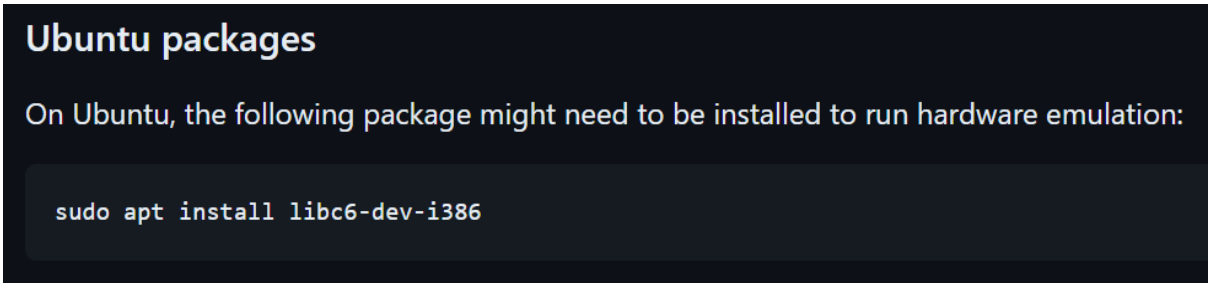
一開始跑hardware emulation的時候沒有意識到kernel size設定成512*512會太大，導致跑emulation的時間會太長，之後決定down size到64*64，以下是build kernel下的參數：

```
cmake ../
 -DMM_DATA_TYPE=float
 -DMM_PARALLELISM_N=4
-DMM_PARALLELISM_M=1
-DMM_MEMORY_TILE_SIZE_N=64
-DMM_MEMORY_TILE_SIZE_M=64
```

一開始的想法是將size和Parallelism同時除8跑跑看emulation，有順利跑出結果並且也是算對的，如下圖所示：

```
02.CtINEm@HLS02:~/m110061608/final/gemm_hls/build_div8$ ./RunHardware.exe 64 64 64 hw_emu
Initializing host memory... Done.
Initializing OpenCL context...
Programming device...
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/02.CtINEm/m110061608/final/gemm_hls/build_div8/.run/5184/hw_em/device0/binary_0/behav
_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommen
ded that a small dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and hence the p
erformance data generated is approximate.
configuring penguin scheduler mode
scheduler config ert(0), dataflow(1), slots(16), cudma(1), cuisr(0), cdma(0), cus(1)
Initializing device memory...
Copying memory to device...
Creating kernel...
Executing kernel...
Kernel executed in 84.0076 seconds, corresponding to a performance of 6.24096e-06 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.
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/02.CtINEm/m110061608/final/gemm_hls/build_div8/.run/5184/hw_em/device0/binary_0/behav
_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
02.CtINEm@HLS02:~/m110061608/final/gemm_hls/build_div8$
```

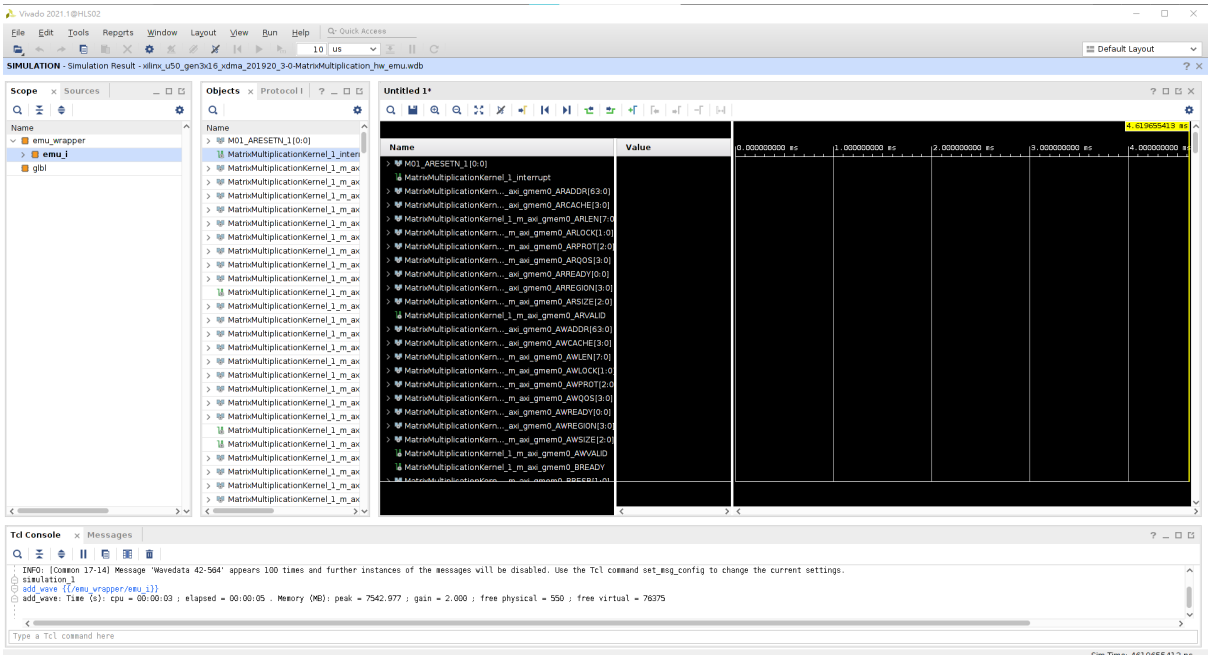
之後要找waveform看但是卻找不到.wdb之類的檔案，後來我們在paper作者的github上看到如果要跑hardware emulation需要在另外安裝package，如下圖所示：



安裝完之後再重新build hardware emulation 就有出現.wdb檔，而且在RunHardware時有出現以下情況：

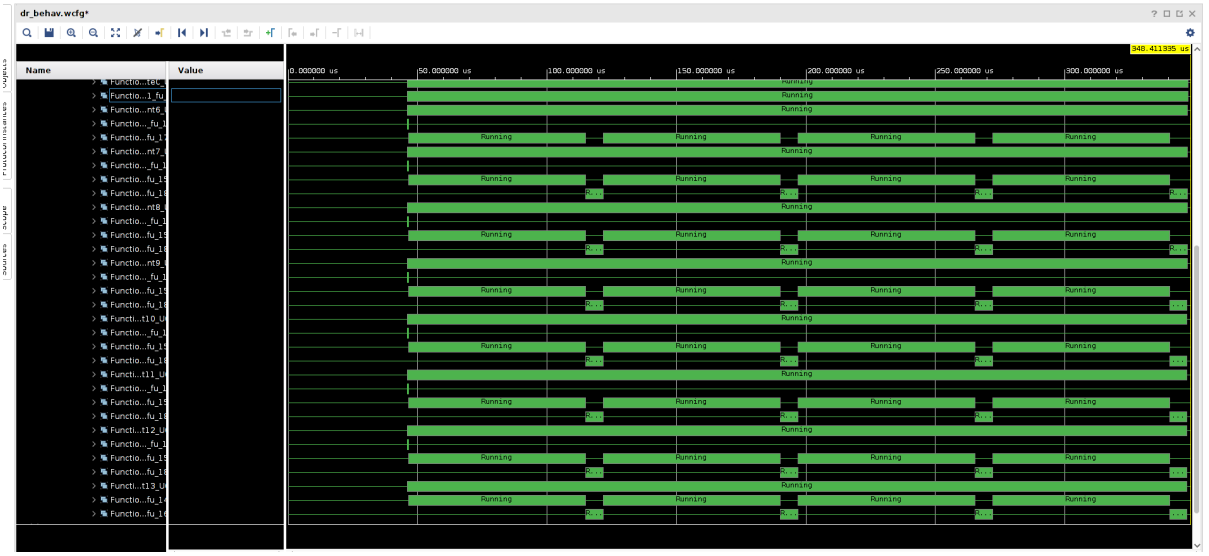
```
02.CtINEm@HLS02:~/m110061608/final/gemm_hls/hw64N8M2emu$ ./RunHardware.exe 64 64 64 hw_emu
Initializing host memory... Done.
Initializing OpenCL context...
Programming device...
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/02.CtINEm/m110061608/final/gemm_hls/hw64N8M2emu/.run/144962/hw_em/device0/binary_0/be
hav_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommen
ded that a small dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and hence the p
erformance data generated is approximate.
configuring penguin scheduler mode
scheduler config ert(0), dataflow(1), slots(16), cudma(1), cuisr(0), cdma(0), cus(1)
Initializing device memory...
Copying memory to device...
Creating kernel...
Executing kernel...
Kernel executed in 87.0084 seconds, corresponding to a performance of 6.02572e-06 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.
Data transfer between kernel(s) and global memory(s)
MatrixMultiplicationKernel_1:m_axi_gmem0-HBM[0]          RD = 16.000 KB              WR = 0.000 KB
MatrixMultiplicationKernel_1:m_axi_gmem1-HBM[1]          RD = 16.000 KB              WR = 0.000 KB
MatrixMultiplicationKernel_1:m_axi_gmem2-HBM[1]          RD = 0.000 KB               WR = 16.000 KB
```

之後用vivado開啟這個檔案裡面只有一些signal的名字沒有waveform如下圖：



後來發現一個dr_behv.wcfg的檔案開啟之後就看到behavior的waveform，如下圖：

memory的read/write也都差不多，但是只能看到behavior不能看內部實際的value是多少，所以我們嘗試不同的input matrix size看behavior有沒有差別，但是PE的部分都差不多如下圖所示：



之後在跟.wdb同directory下有看到一個profile_kernel.csv檔，裡面記錄著各個function的runtime，所以我們決定對同一個kernel size用不同的input matrix size跑hardware emulation觀察有什麼差別。

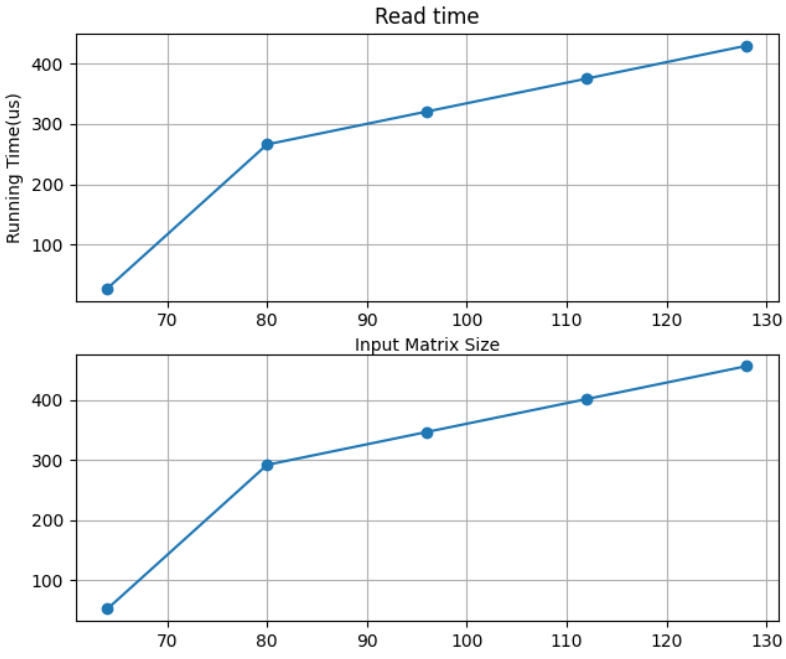
kernel size都是固定64，平行度的部分
-DMM_PARALLELISM_N=16 / 8
-DMM_PARALLELISM_M=4 / 2
我們有嘗試過N/M ratio 不是 4，但是跑emulation的時候會產生deadlock的問題，因為時間有限所以都用 N : M = 4 : 1做參數設定。

input matrix size是{64,80,96,112,128},
N K M 的部分都是相同的size(=input matrix size),
總共有以下幾個function要觀察:

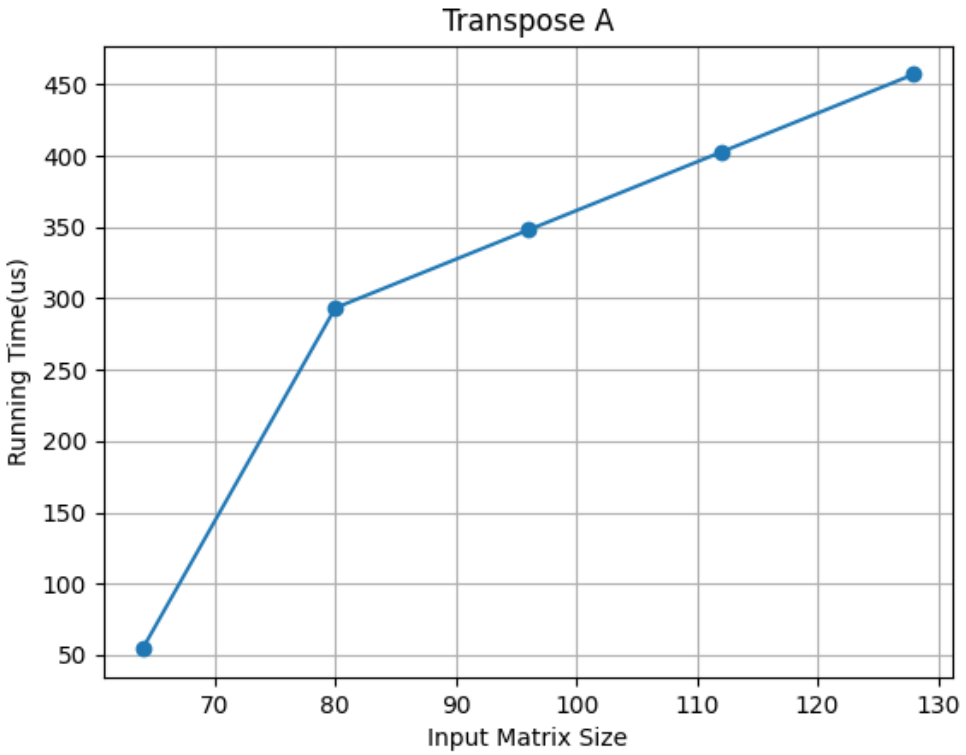
- ReadA,B
- TransposeA
- ConvertWidthB,C
- FeedB
- WriteC
- PE

❖ PARALLELISM_N = 8 , PARALLELISM_M = 2

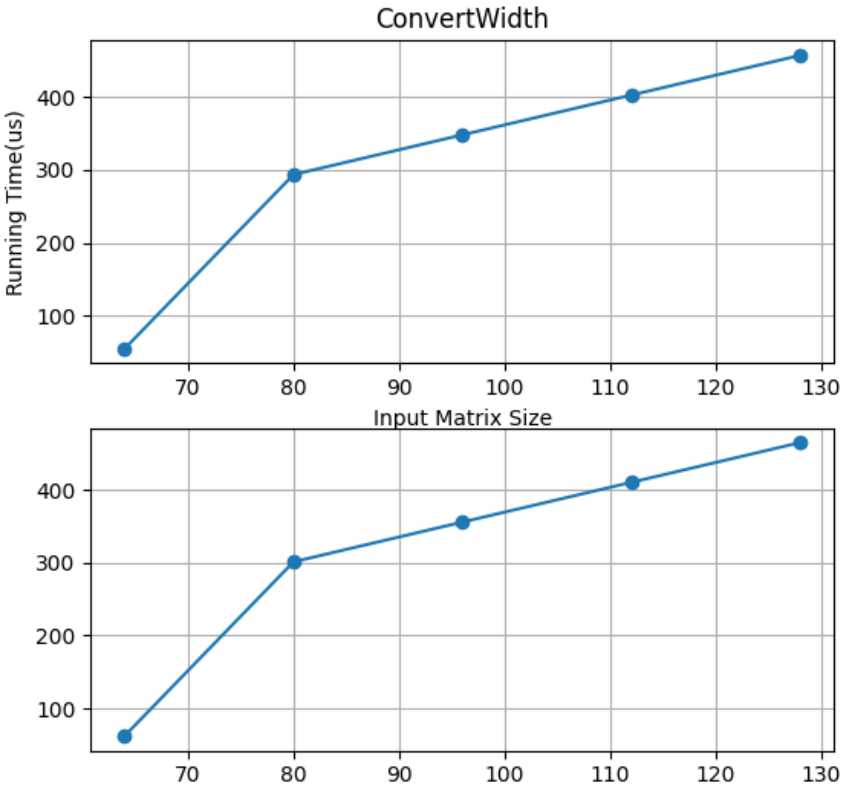
- ReadA,B



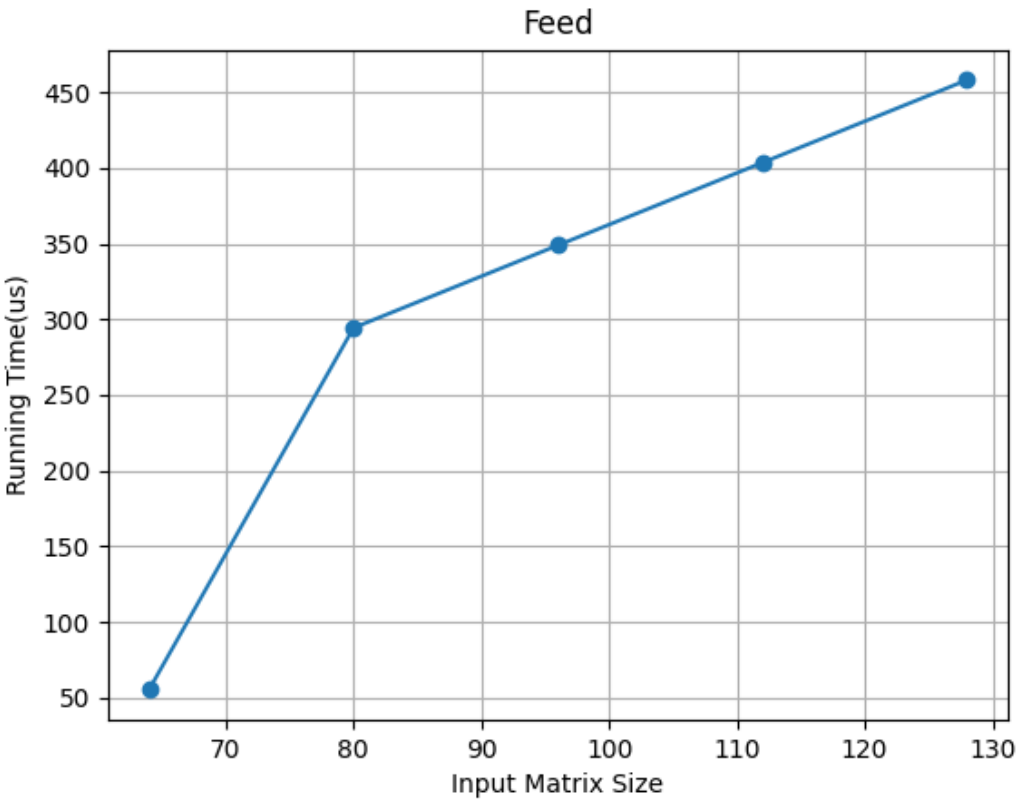
- TransposeA



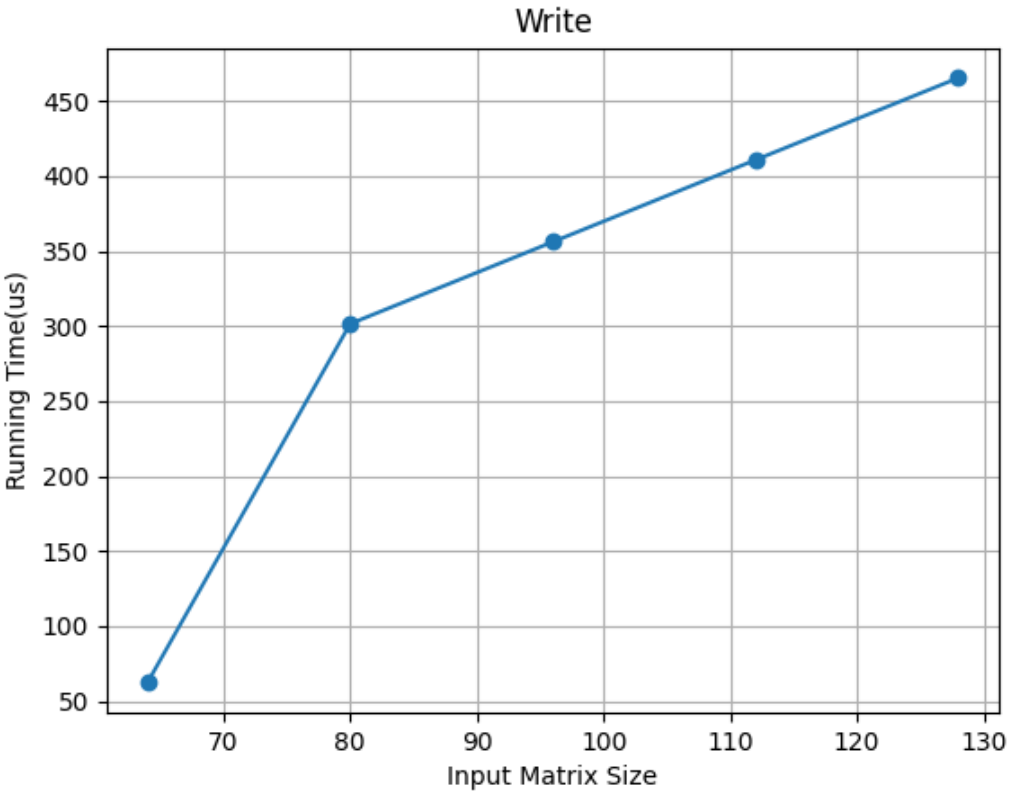
- ConvertWidthB,C



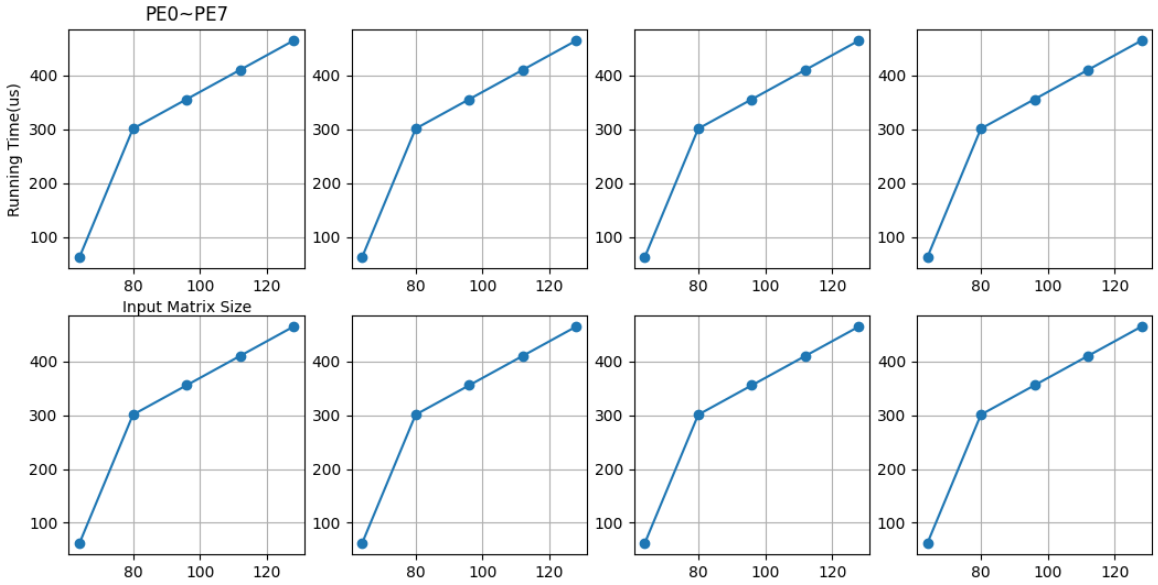
- FeedB



- WriteC

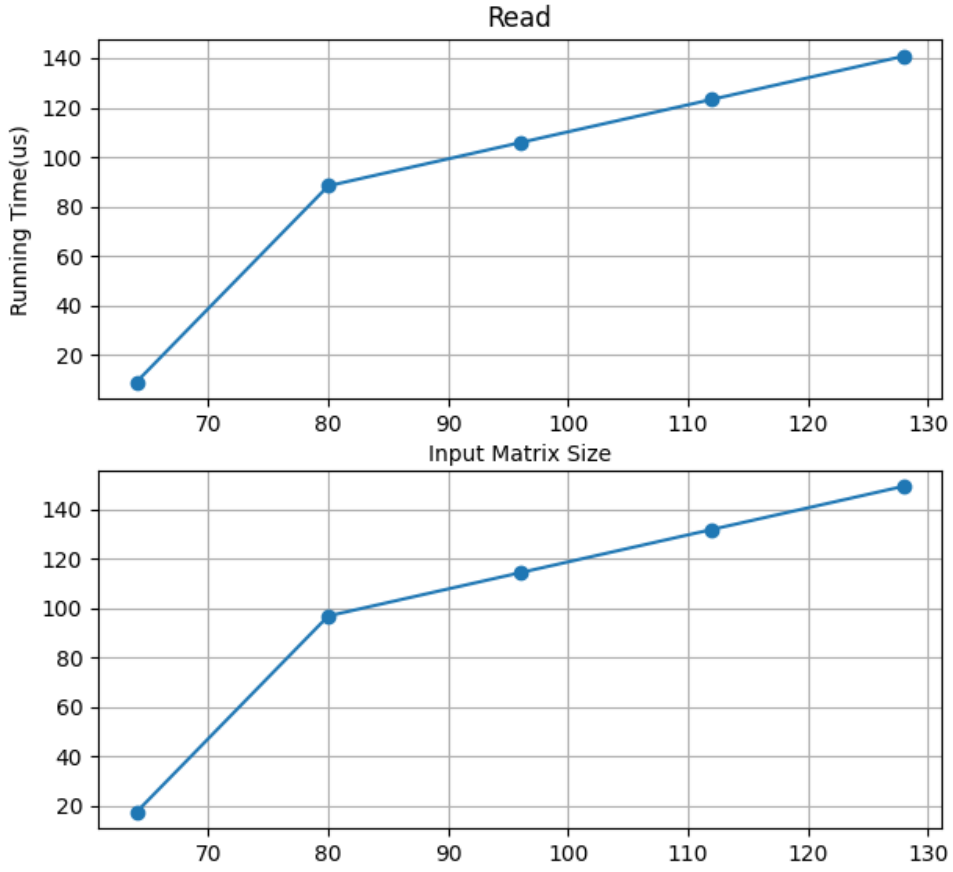


- PE0~PE7

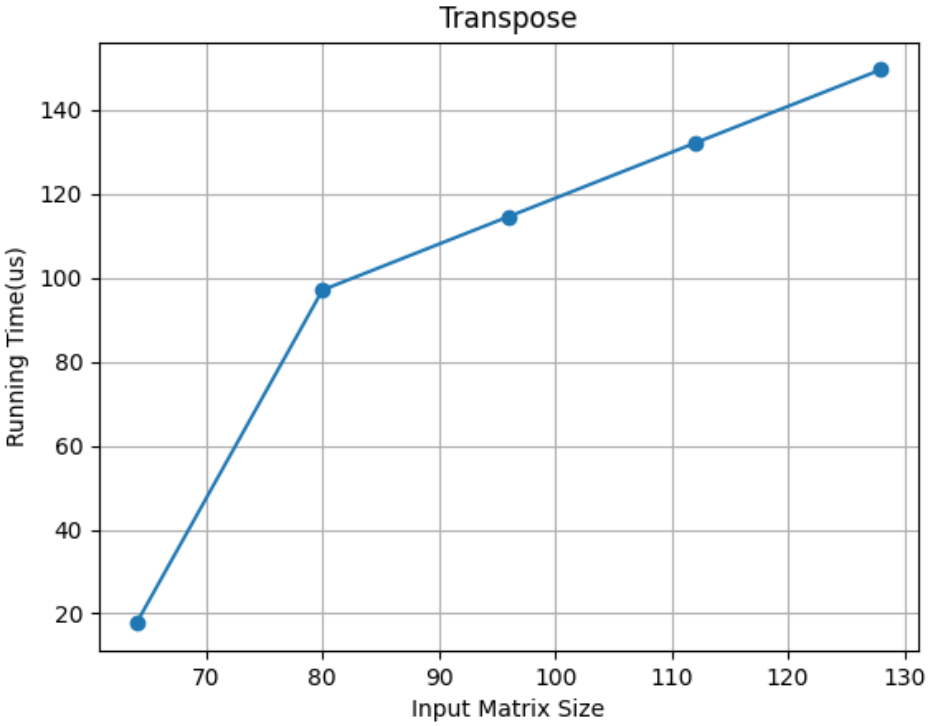


❖ PARALLELISM_N = 16 , PARALLELISM_M = 4

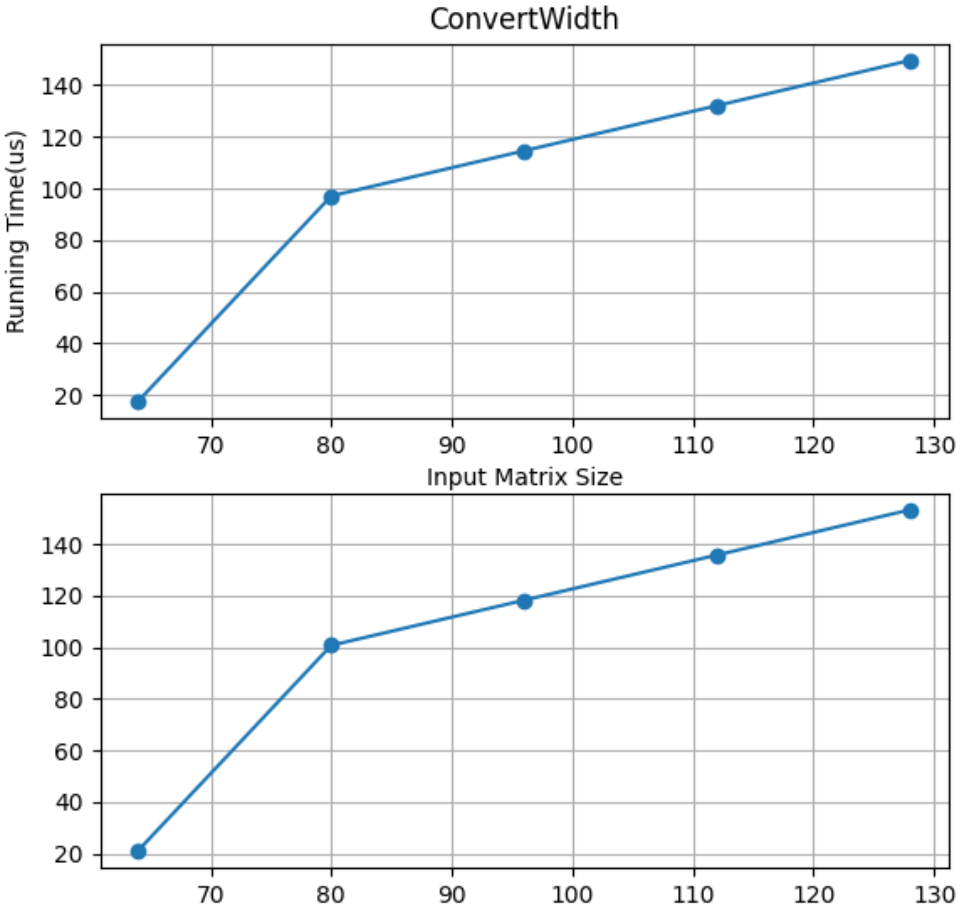
- ReadA,B



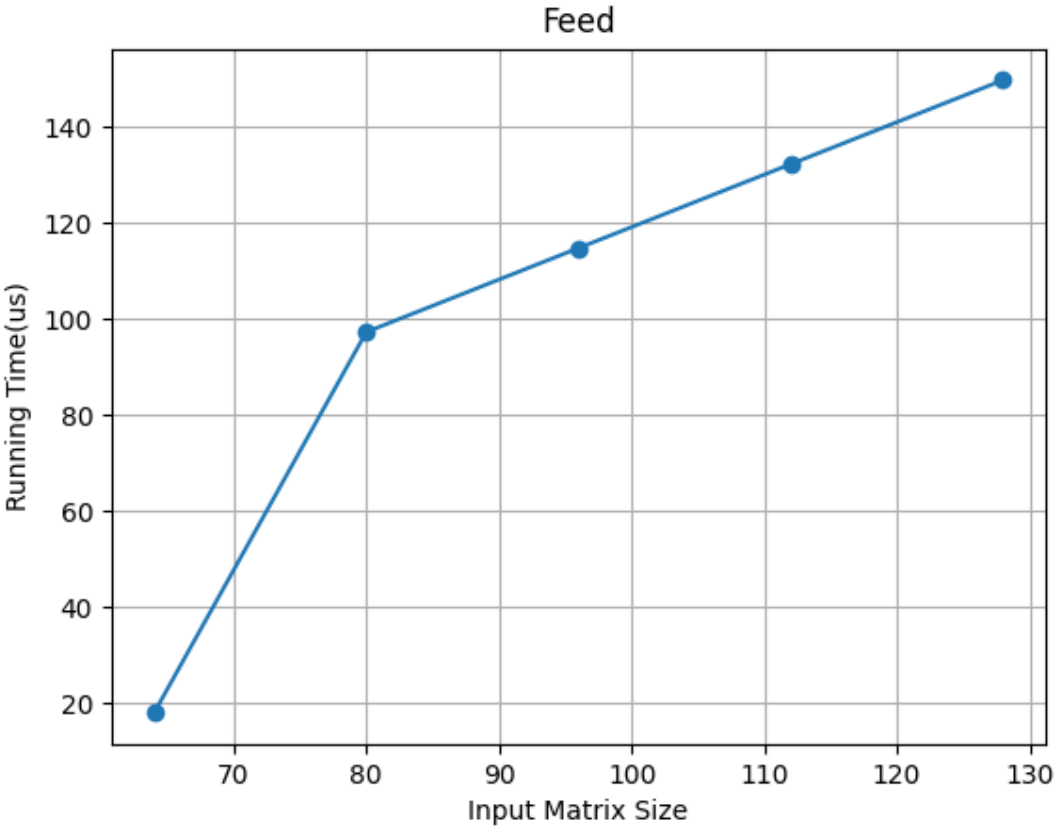
- TransposeA



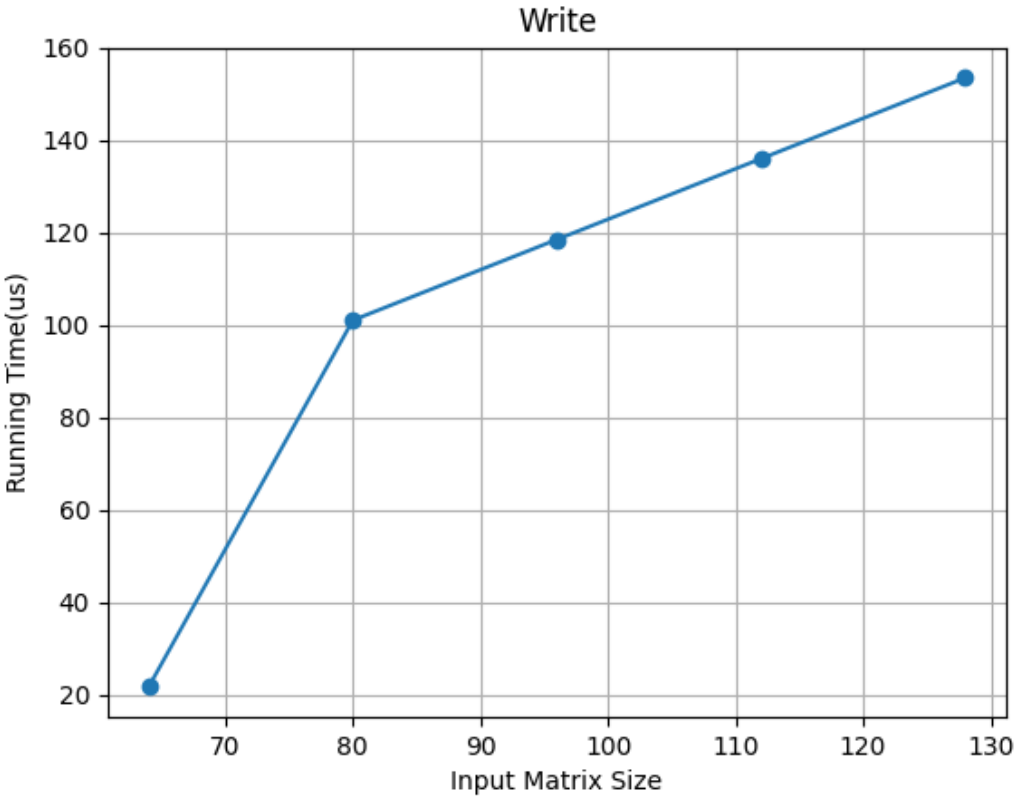
- ConvertWidthB,C



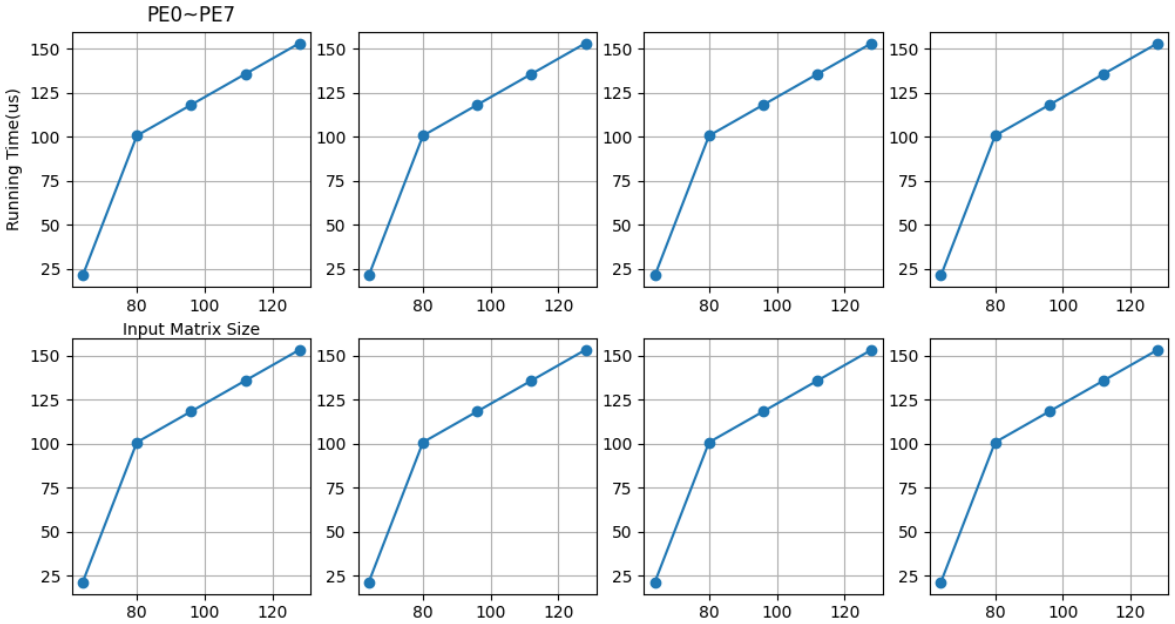
- FeedB



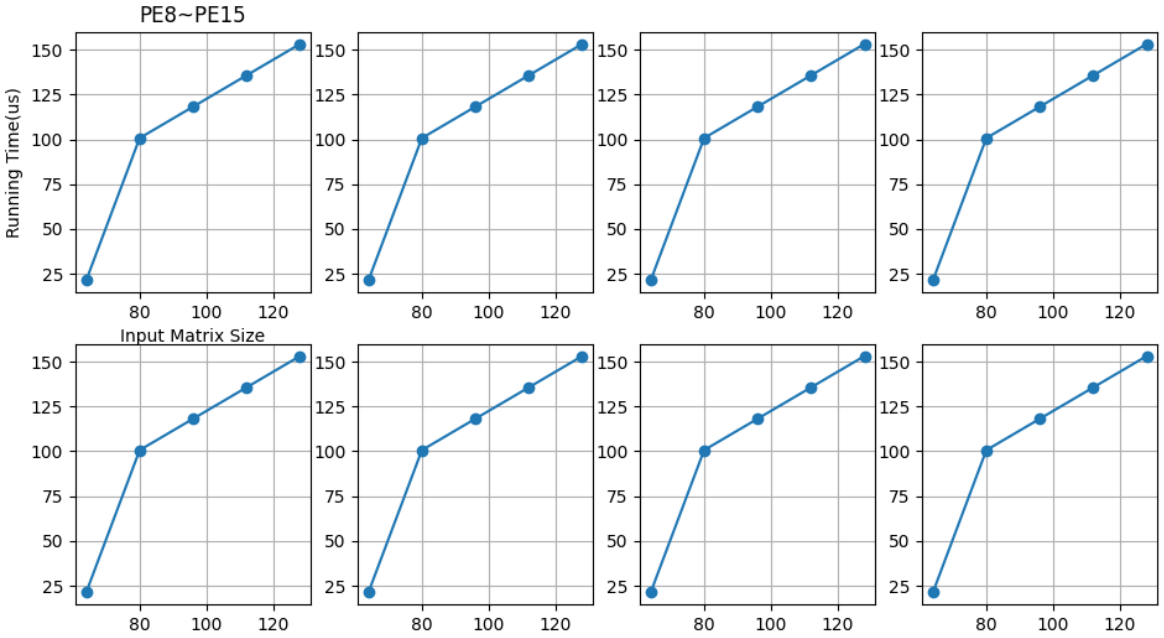
- WriteC



- PE0~PE7

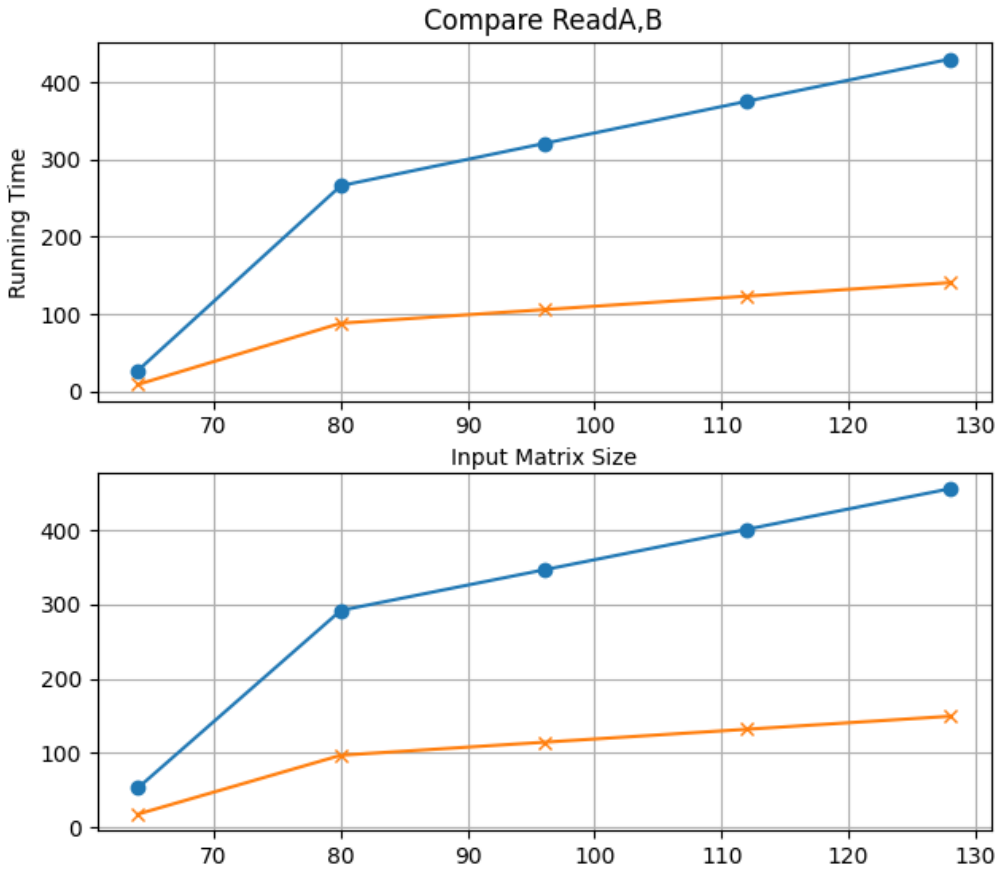


- PE8~PE15

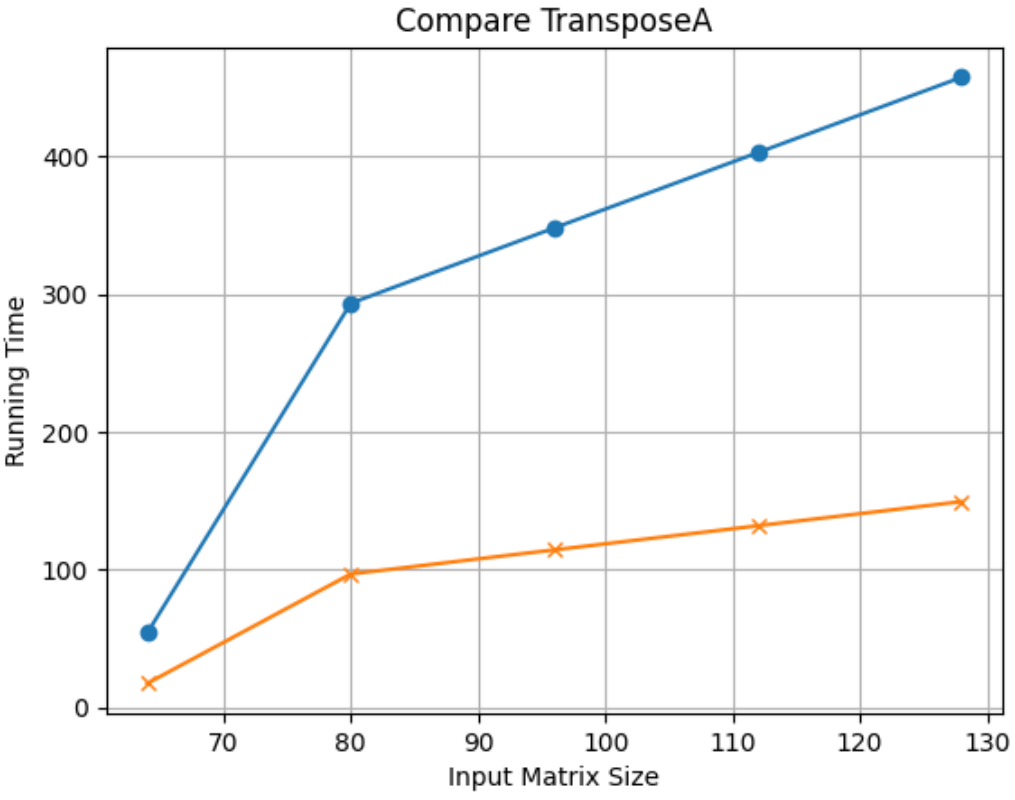


透過上面的結果可以發現當input matrix size 從64到80的時候Runtime會大幅度增加，而之後是線性的增長，所以我們想進一步分析，(N,M) = (8,2) 和 (N,M) = (16,4)的差異，也是對各個function分析，藍線是(N,M) = (8,2)，橘線是(N,M) = (16,4)。

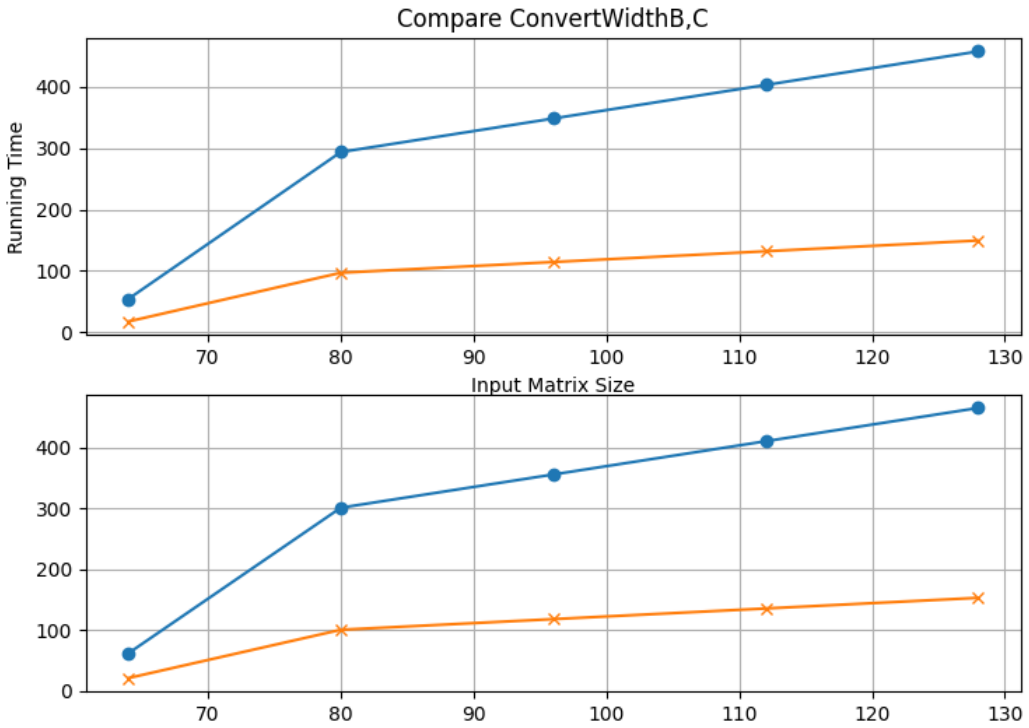
- ReadA,B



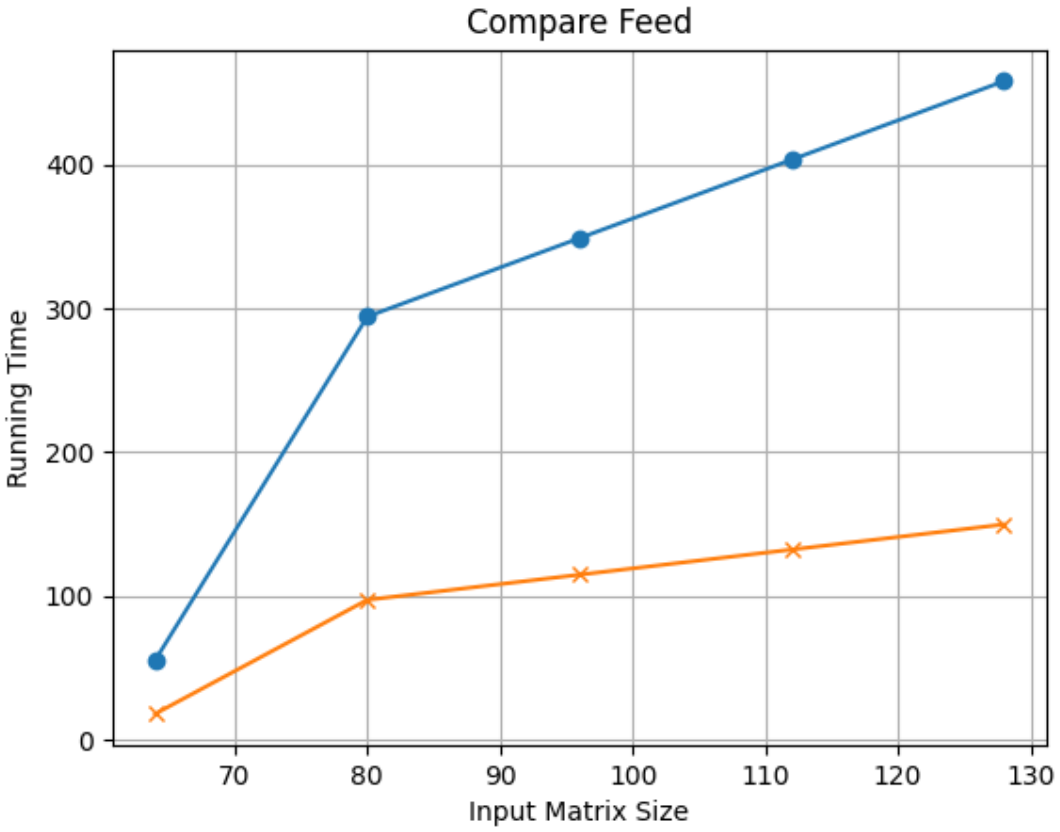
- TransposeA



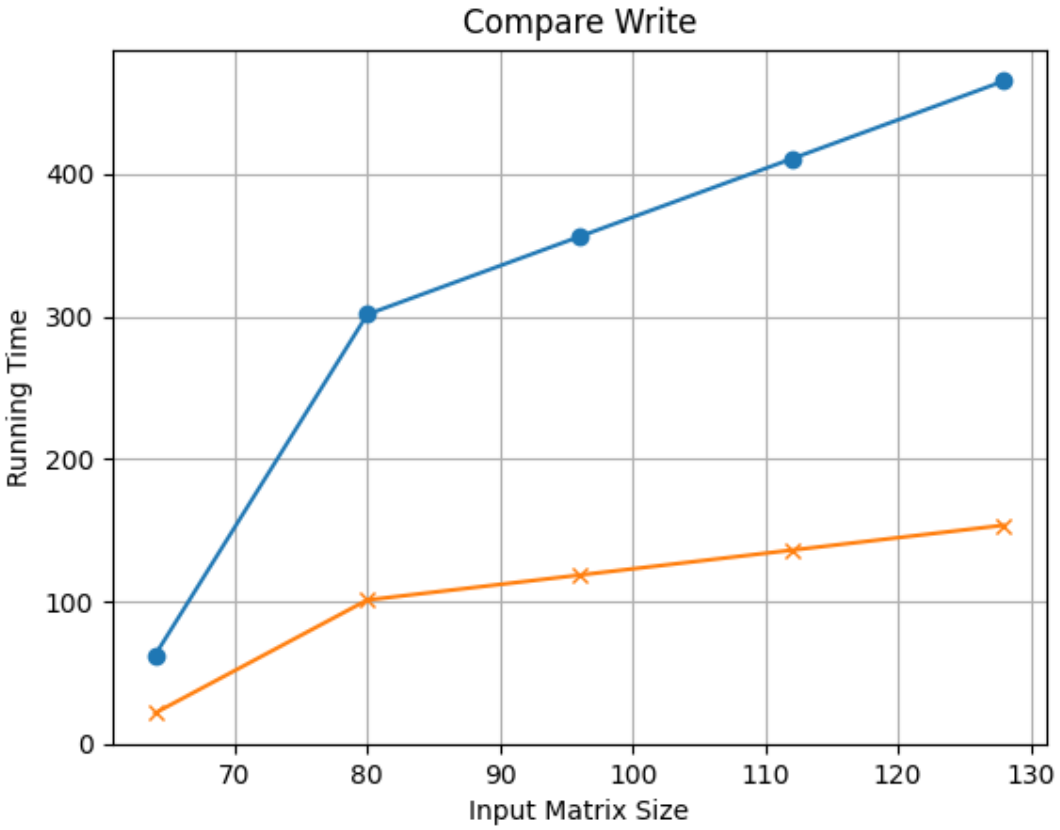
- ConvertWidthB,C



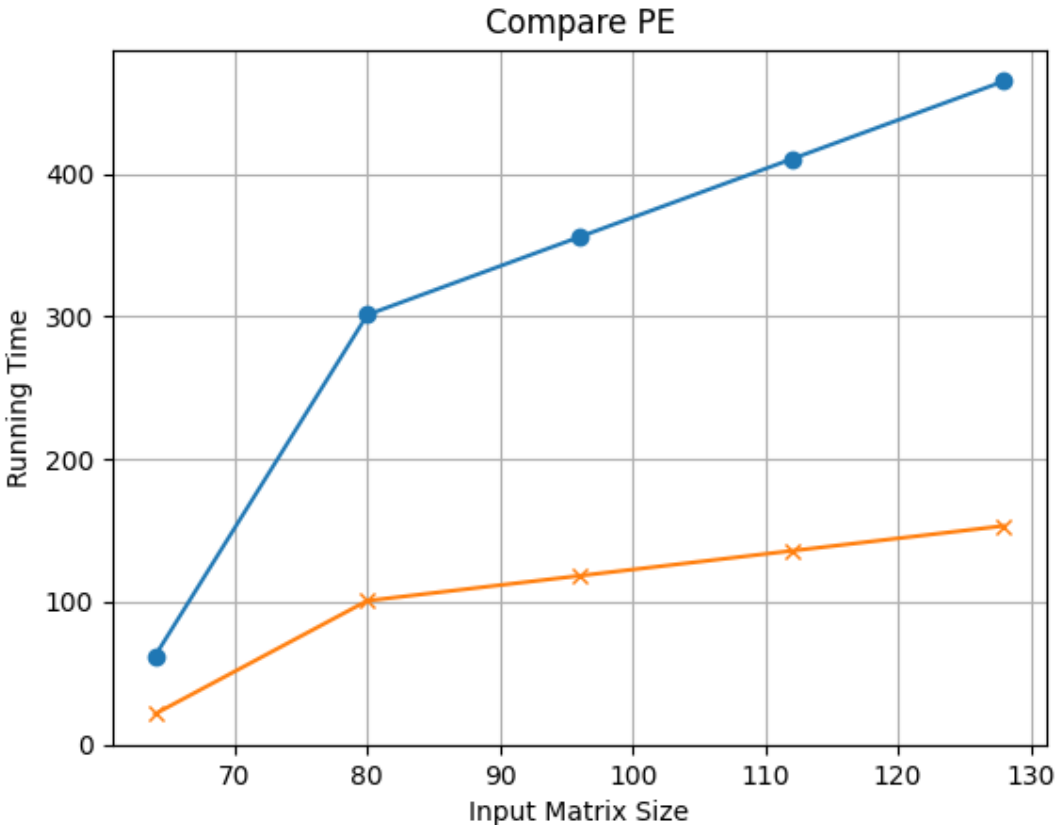
- FeedB



- WriteC

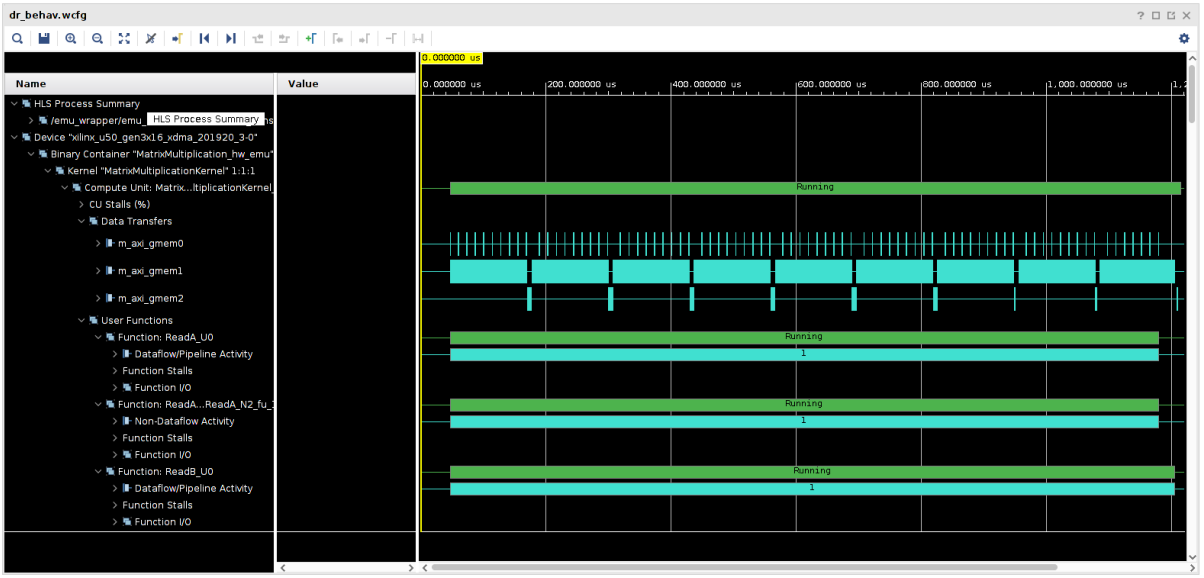

Compare Write
Running Time
Input Matrix Size
0
100
200
300
400
70
80
90
100
110
120
130

- PE

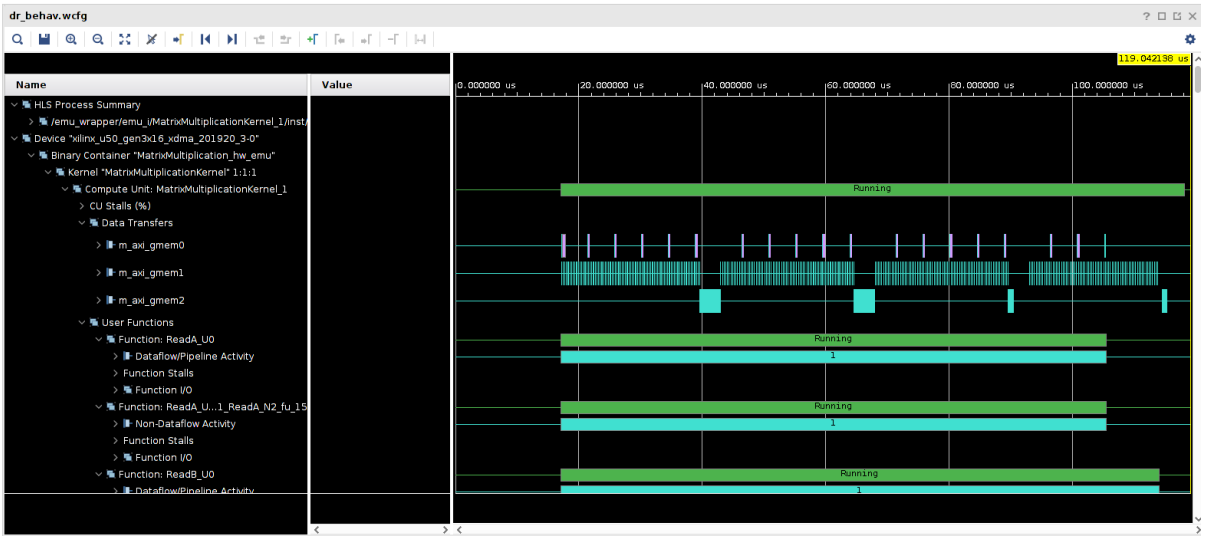

Compare PE
Running Time
Input Matrix Size
0
100
200
300
400
70
80
90
100
110
120
130

從上述結果可以得出橘線的上升幅度比藍線小，因為N代表有幾個PE(Processing element)，M代表一個PE裡有幾個CU(Compute unit)，所以我們認為當PE和CU變多的時候，在各個function裡會讓Runtime上升的幅度減小。

但還是沒有找到Drop的原因，所以我們從另一個角度去分析，我們發現當PE和CU數量不同，會對Runtime有影響，所以我們再去看waveform進而發現他read/write的次數不同，下圖是(N,M) = (8,2)，input matrix size為80：



下圖是(N,M) = (16,4)，input matrix size為80：



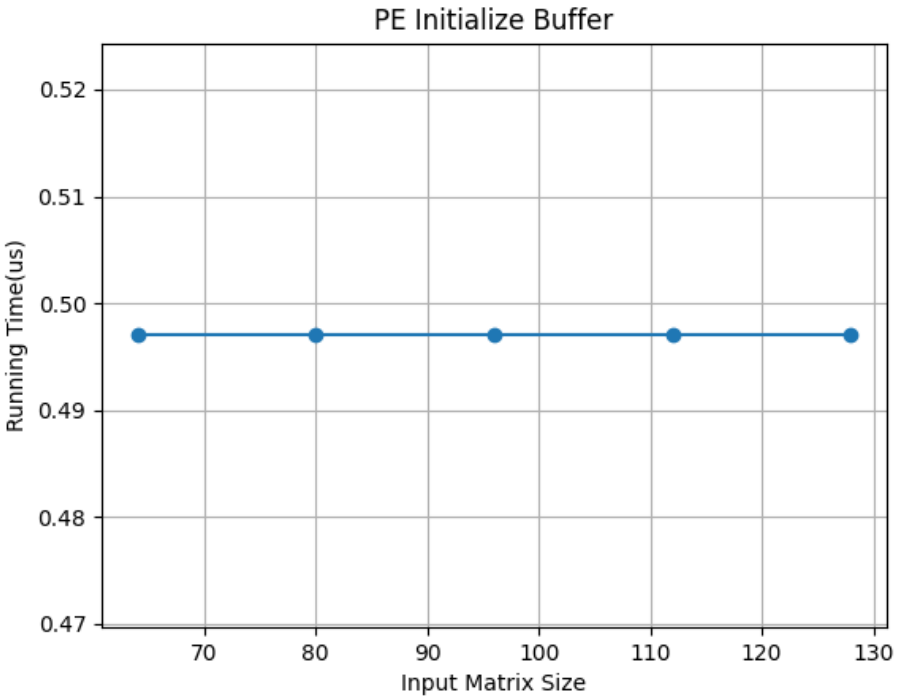
由上述可得，PE和CU數量越多他讀寫的次數會比較少。

再者我們分析PE裡有再細分成三個function:

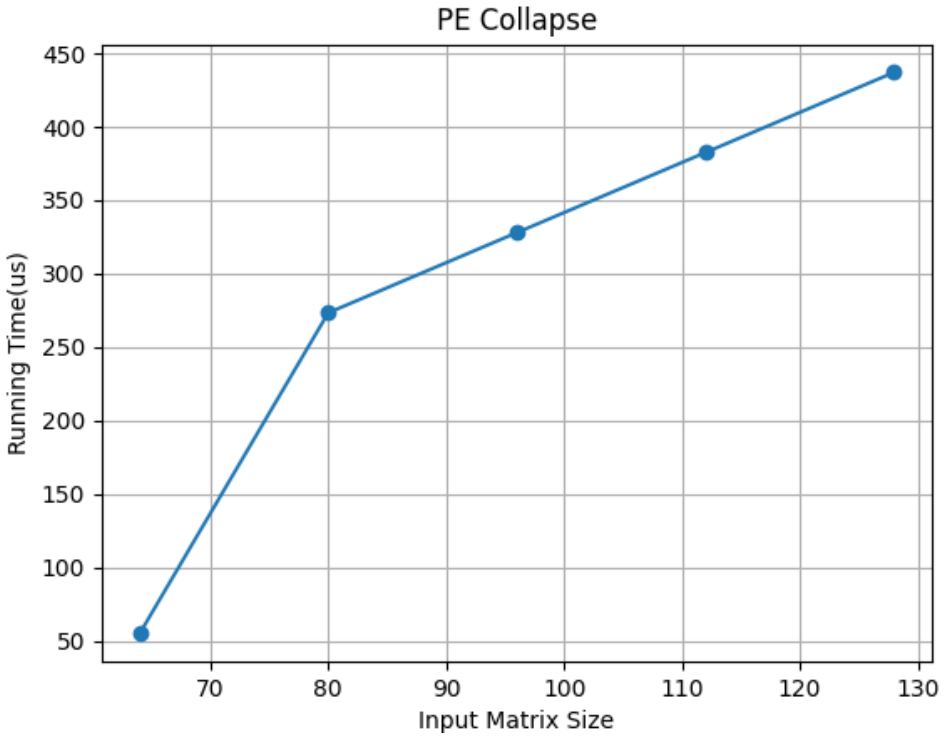
- Initialize ABuffer
- Compute
- WriteC

針對這三個function作圖分析，分析結果如下:

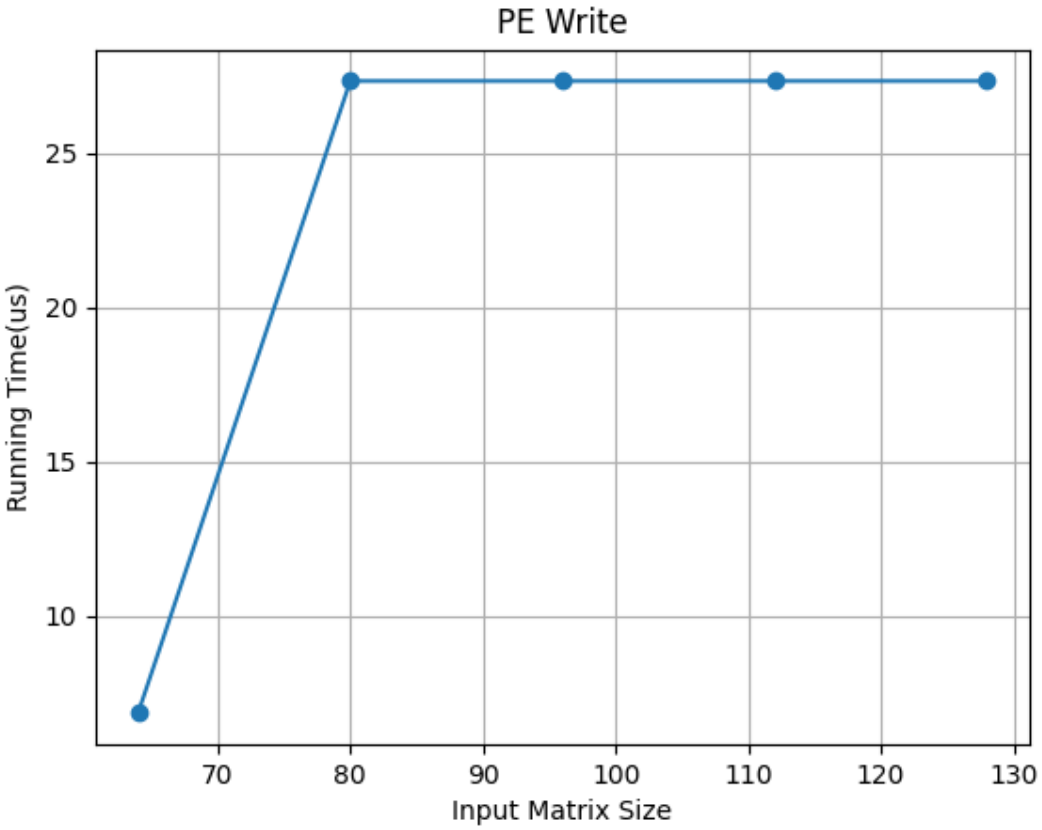
- (N,M) = (8,2) ，Input matrix size = 80
    - Initialize Abuffer



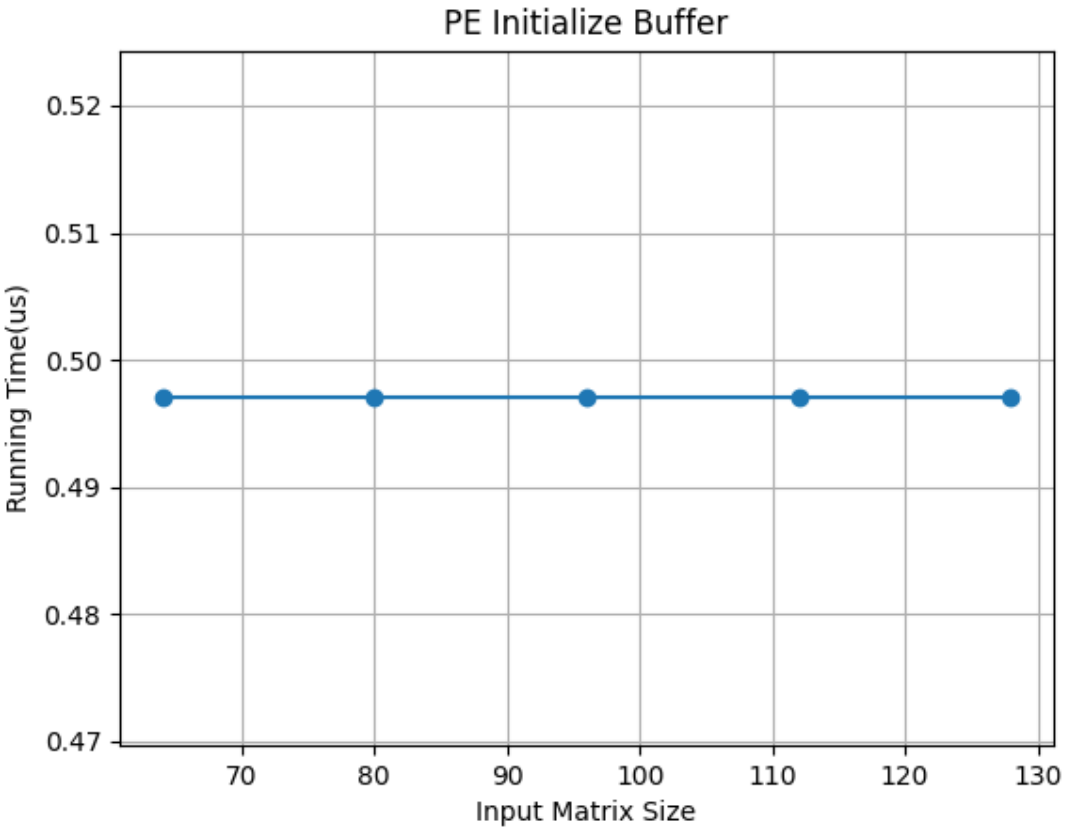
    - Compute



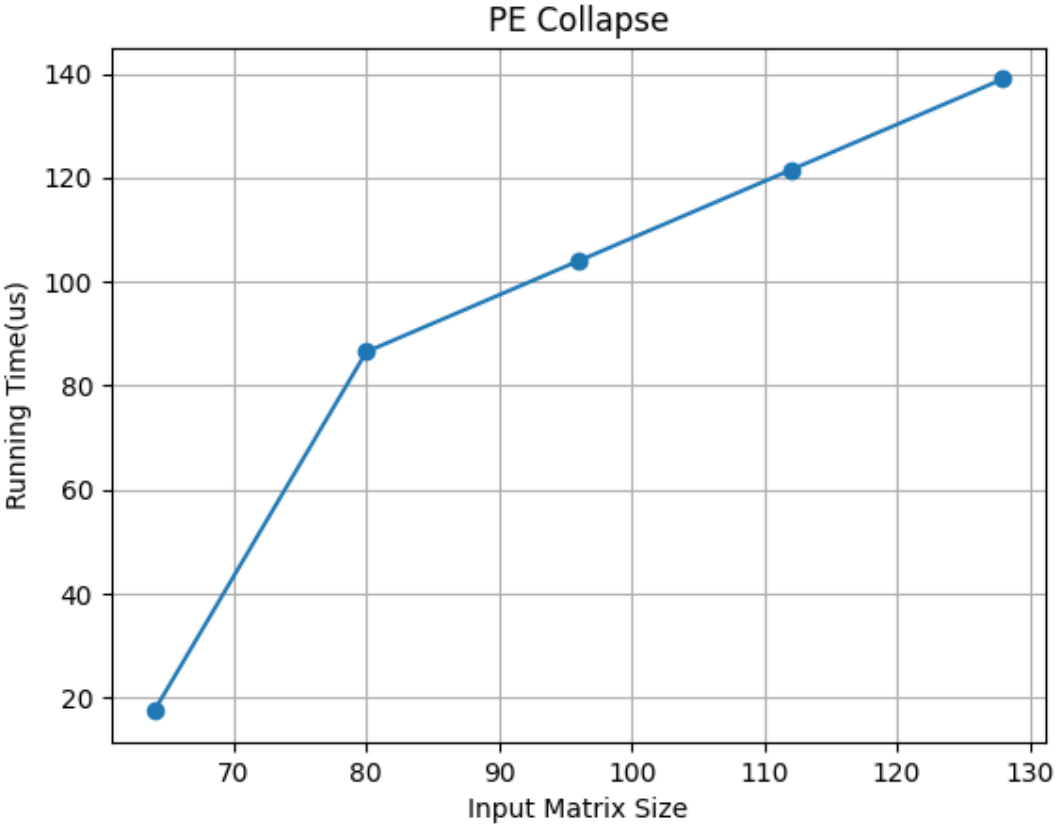
- WriteC


PE Write
Running Time(us)
25
20
15
10
70
80
90
100
110
120
130
Input Matrix Size

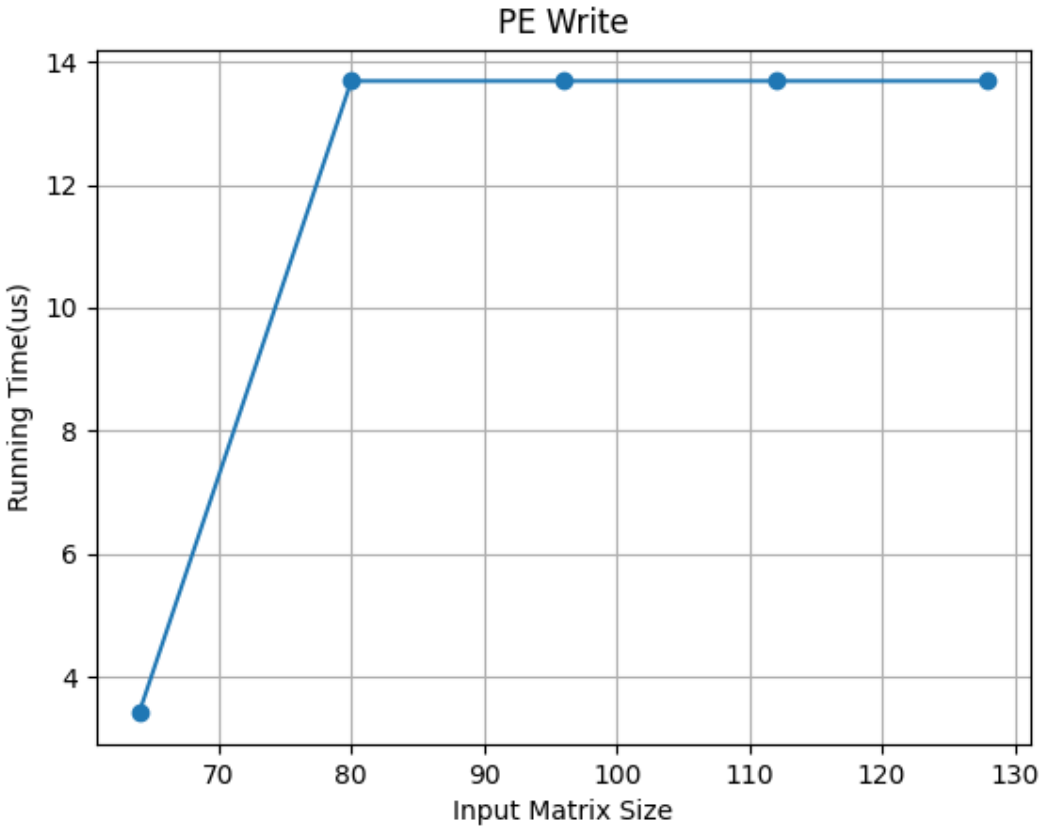
- (N,M) = (8,2) , Input matrix size = 80
  - Initialize Abuffer



  - Compute

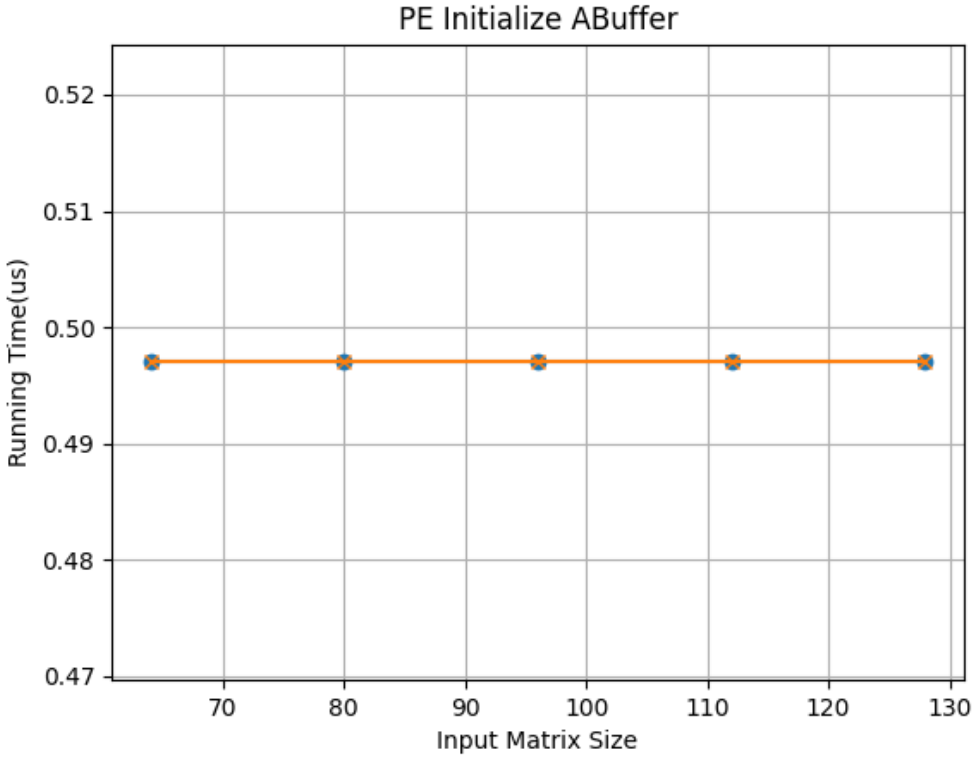


- WriteC

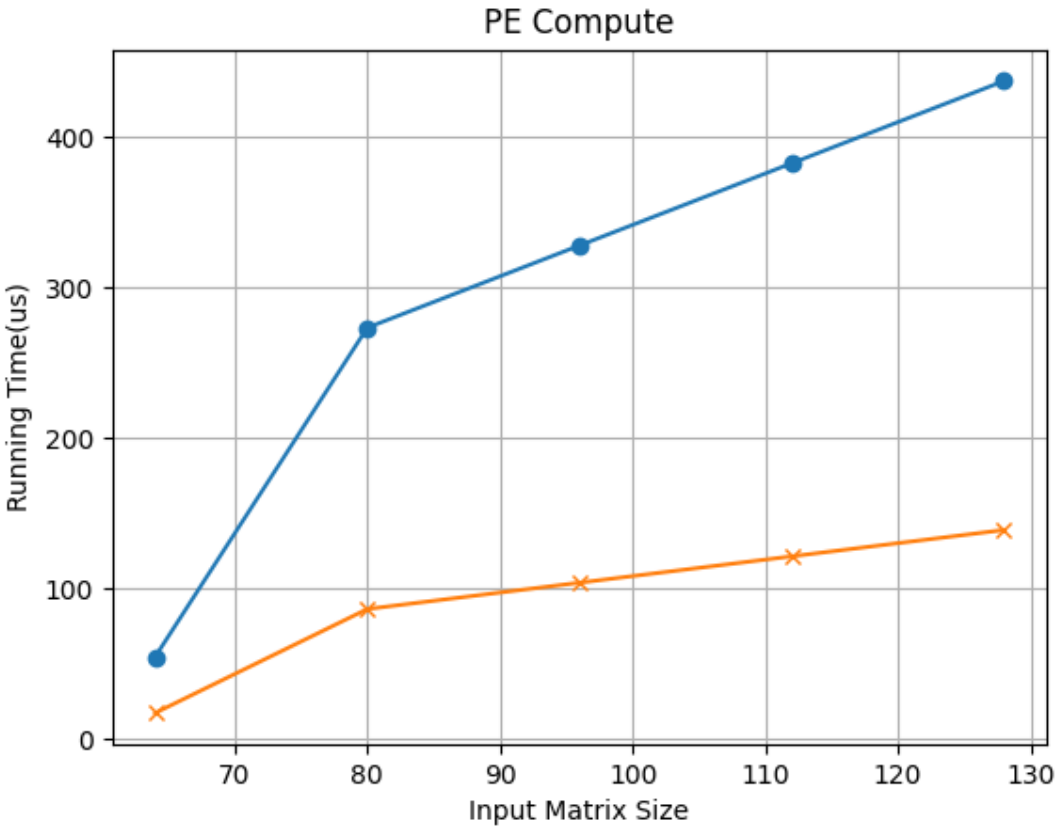


再來將兩者比較，藍線是(N,M) = (8,2)，橘線是(N,M) = (16,4)，如下圖所示：

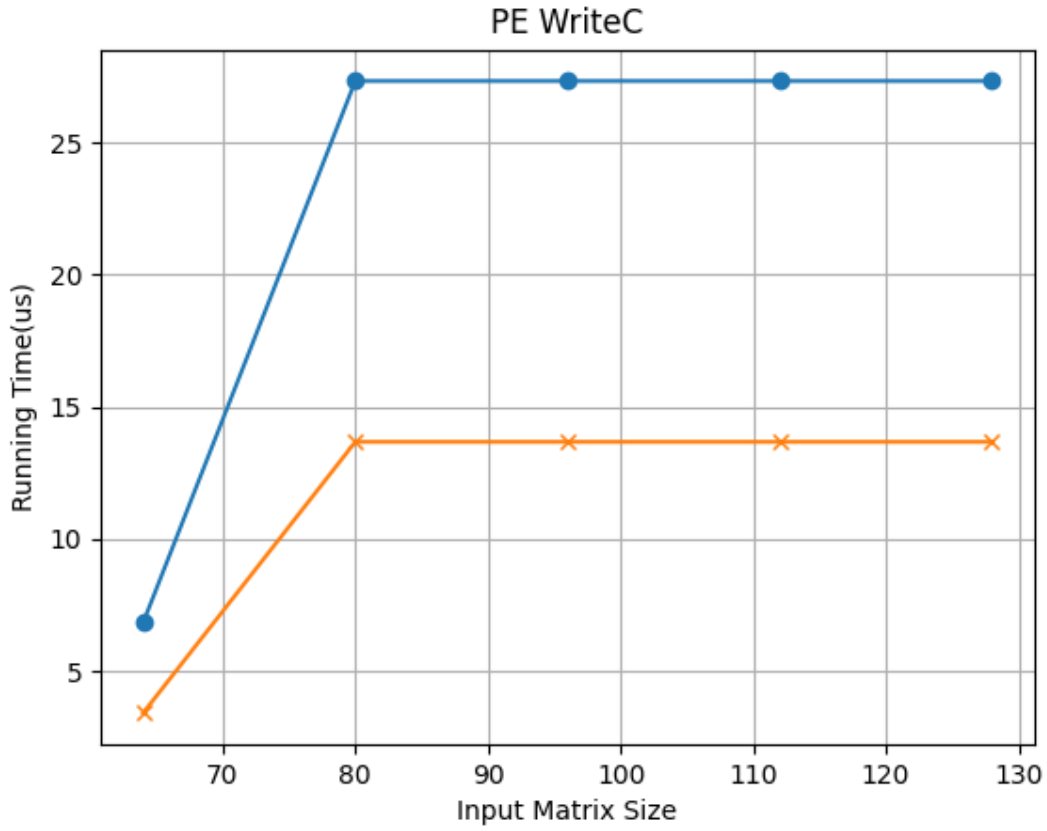
- Initialize ABuffer



- Compute



- WriteC



從上述結果可得知，在Initialize ABuffer不管N,M是多少都是花一樣的時間，在Compute的部分藍線的斜率變化比橘線高，所以PE和CU的數量也是會有影響，數量越多所花的時間就越少，在WriteC的部分，只要超過kernel size(64*64)，input size增加但Write的時間還是一樣，當input從64變成80時，兩者的Runtime都有增加，只是藍線的幅度比橘線大，這個跟上面分析function runtime的結果是相同的。

5. Reflection

這次的final.project我們學到更多關於Xilinx.tool的用法，用cmake和makefile的用法去build.design，透過這些tool先重現出別人的design，再透過tool去分析waveform,runtime，雖然還是沒有找到performance drop的確切原因，但我們推測應該是因為kernel是計算512*512的大小，當input matrix size超過kernel size或是kernel size的倍數，在後面沒有inpur的時候要等到正在算的data算完才能去verify，所以整個時間才會拉長，如果是符合kernel.size的話，等待的時間就會比較短，但以上都只是推測，因為只能看behavior的waveform沒有辦法看實際上發生的情況，因為時間有限所以沒辦法分析完整，但還是有找出一些端倪可以當作出發點去研究。

reference

[1] Z. Chen, Hardware Accelerator of Matrix Multiplication on FPGAs (Uppsala Universitet) p. 1
[2] J. d. F. Licht, G. Kwasniewski, T. Hoefler, Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis https://spcl.inf.ethz.ch/Publications/.pdf/gemm-fpga.pdf
[3] 2022's Final Project
https://github.com/bol-edu/2022-spring-nthu/blob/main/Final/gemm_hls/Final/Group8_Final_Report.pdf