

AAHLS Final Project

Scalable matrix matrix multiplication

方爾秋 謝睿宇

● Computation utilization

IV. 部署到 U50 的實際測試結果

```
[g1999061806@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。

參考 2022 清大同學結果做計算

$$(\text{DMM_PARALLELISM_N}) = 32$$

$$(\text{DMM PARALLELISM M}) = 8$$

Assume hardware resource is fully utilized.

$$1024 * 1024 * 1024 / (32 * 8) / (\text{fs} = 300\text{e}6) = 0.01398\text{sec}$$

$$0.01398 / 0.014706 = 0.951 \leq 0.05 \text{ idle}$$

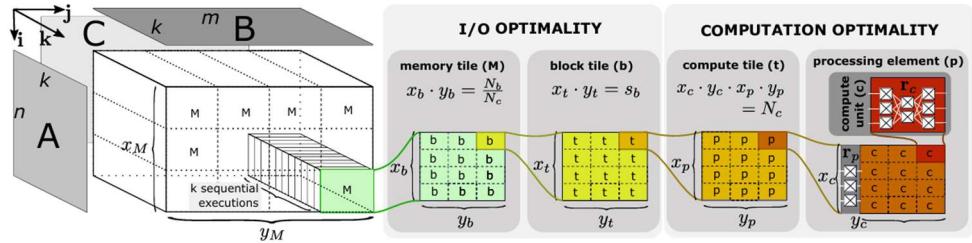
All most fully utilized hardware resource for memory tile 512*512.

● 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



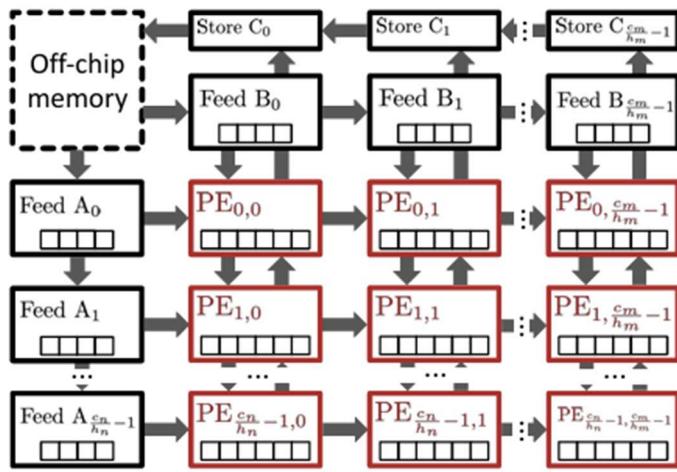
Configurable MMM 的 hierarchy 可以分為兩大部分，往下又能再細分成四個區塊，分別是 memory tile、block tile、compute tile 以及 processing element。兩大部分則是 memory 以及 computation。首先我們先看 computation 部分，在 computation 中最小的單位是 compute unit，一個 compute unit 支援 one word 的 MAC(乘加器)，PE 包含 $x_c y_c$ 個 compute unit，再往上是 compute tile，它包含 $x_p y_p$ 個 PE。這邊就是所有 parallel processing compute unit 數量($x_c y_c x_p y_p$)。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 N_b / N_c 個 block tile，其中 N_b 代表目標平台的 block ram 數量，而 N_c 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 r_c ，而目標平台最大的 hardware resource 是 r_{max} ，再考慮單一個 PE 所需要 Control overhead 是 r_p ，我們就能得出 $N_p r_p + N_c r_c \leq r_{max}$ 。如果再考慮 routing congestion 問題，給定不同 PE 之間一條 Bus width 是 w_p ，我們就會有額外兩條不等式。

$$x_c w_c \leq w_p, \quad y_c w_c \leq w_p$$

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



Memory Resource

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

$$N_{b,min} = x_p y_p \left\lceil \frac{w_c \cdot x_c y_c}{w_b} \right\rceil$$

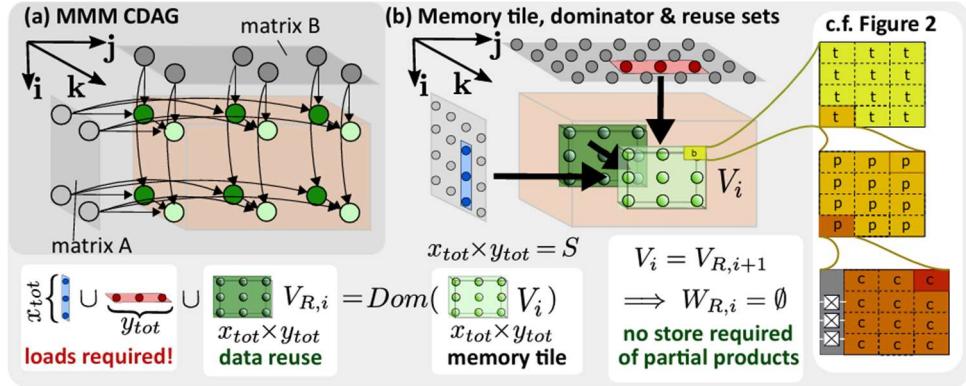
而 memory tile 內包含的 block ram 數量也可透過下列公式算出：

$$x_b \cdot y_b = \left\lceil \frac{N_b}{N_{b,min}} \right\rceil$$

因為一個 block ram 可以儲存 s_b 個 words，因此一個 block tile 可以支援 compute tile s_b 次 parallel processing，進而可以求得一個 memory tile 可以支援 $x_b \cdot y_b$ 次 block tile。在最好的情況下， N_b 是 $N_{b,min}$ 的整數倍，而最糟情況下 $N_b = 2N_{b,min} - 1$ 。

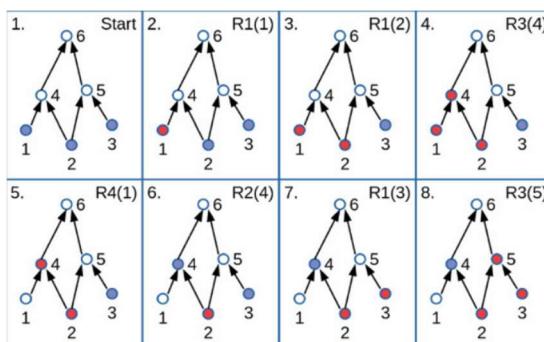
2. I/O

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。



可以看到 A 是一個 $i \times k$ 的 matrix，而 B 是一個 $k \times j$ 的 matrix，而 C 則是 $i \times j$ 的 matrix。此外，也能觀察到 intermediate term 總共有 $i \times j \times k$ 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。



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 $S = N_b \cdot s_b$ ，代表 on-chip memory 能儲存

的總 word 數。 $x_{tot} = x_c \cdot x_p \cdot x_t \cdot x_b$, $y_{tot} = y_c \cdot y_p \cdot y_t \cdot y_b$ ，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： $x_{tot} + y_{tot} \leq S$, $x_{tot} y_{tot} \leq S$ ，盡可能提升 computation intensity，也就是

$$x_{tot} y_{tot} / (x_{tot} + y_{tot})$$

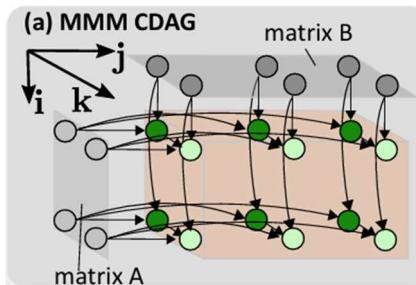
最終結果是：

$$Q = mn \left(1 + k \left(\frac{1}{x_{tot}} + \frac{1}{y_{tot}} \right) \right) \quad x_{tot} = y_{tot} = \sqrt{S}$$

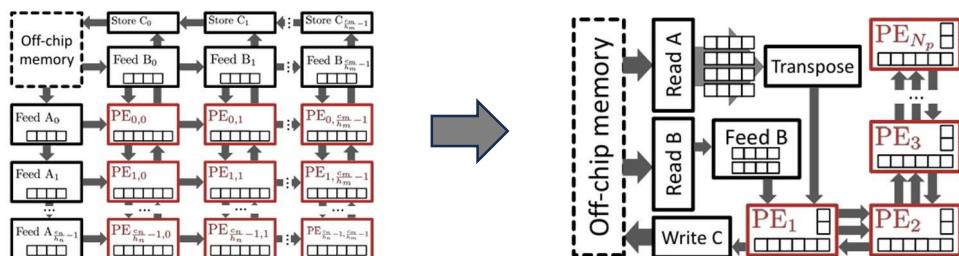
3. Hardware implementation

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 $x_p \cdot y_p$ 。為了緩解這項問題，paper 採用 systolic array 方式。



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。



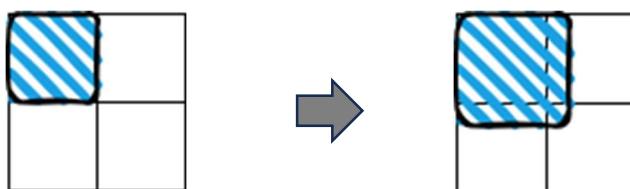
接著是 matrix A 的 column-wise read 問題。由於 outer product 需要讀取 matrix A 的部分 column vector，這種 column-major access 會造成 I/O 效率低下，因此需要採用 *on the fly transposition* 方式，具體細節由另一位同學在 code 部分做介紹。

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 nm/y_c cycles 來 write back，以及 mnk/N_c 個 cycles compute。如果 $k/N_c \gg 1$ 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 $x_p x_m \cdot y_p y_m$ 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 $x_p x_m \cdot y_p y_m$ 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。

4. 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



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

```
std::default_random_engine rng(kSeed);
typename std::conditional<std::is_integral<Data_t>::value,
                           std::uniform_int_distribution<unsigned long>,
                           std::uniform_real_distribution<double>>::type
dist(1, 10);

bool emulation = false;
bool verify = true;
hlslib::UnsetEnvironmentVariable("XCL_EMULATION_MODE");
std::string path = "MatrixMultiplication_hw.xclbin";
#ifndef MM_DYNAMIC_SIZES
if (argc > 6 || argc < 4) {
    PrintUsage();
    return 1;
}
const unsigned size_n = std::stoul(argv[1]);
const unsigned size_k = std::stoul(argv[2]);
const unsigned size_m = std::stoul(argv[3]);
int next_arg = 4;
if (size_k % kMemoryWidthK != 0) {
    std::cerr << "K (" << size_k
        << ") must be divisible by the memory width in K ("
        << kMemoryWidthK << ")." << std::endl;
    return 1;
}
```

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。

```

    std::vector<Data_t> a, b, cRef;
    std::vector<MemoryPackK_t, hlslib::ocl::AlignedAllocator<MemoryPackK_t, 4096>>
        aMem;
    std::vector<MemoryPackM_t, hlslib::ocl::AlignedAllocator<MemoryPackM_t, 4096>>
        bMem, cMem;
    std::cout << "Initializing host memory..." << std::flush;
    if (verify) {
        a = decltype(a)(size_n * size_k);
        std::for_each(a.begin(), a.end(),
                      [&dist, &rng](Data_t &in) { in = Data_t(dist(rng)); });
        b = decltype(b)(size_k * size_m);
        std::for_each(b.begin(), b.end(),
                      [&dist, &rng](Data_t &in) { in = Data_t(dist(rng)); });
        cRef = decltype(cRef)(size_n * size_m, 0);
    }

    aMem = Pack<kMemoryWidthK>(a);
    bMem = Pack<kMemoryWidthM>(b);
    cMem = Pack<kMemoryWidthM>(cRef);
}
std::cout << " Done.\n";

```

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。

```

try {
    std::cout << "Initializing OpenCL context...\n" << std::flush;
    hlslib::ocl::Context context;

    std::cout << "Programming device...\n" << std::flush;
    auto program = context.MakeProgram(path);

    std::cout << "Initializing device memory...\n" << std::flush;
}

```

初始化 OpenCL。

```

    if (verify) {
        std::cout << "Copying memory to device...\n" << std::flush;
        aDevice.CopyFromHost(aMem.cbegin());
        bDevice.CopyFromHost(bMem.cbegin());
        cDevice.CopyFromHost(cMem.cbegin());
    }

    std::cout << "Creating kernel...\n" << std::flush;
#ifndef MM_DYNAMIC_SIZES
    auto kernel = program.MakeKernel("MatrixMultiplicationKernel", aDevice,
                                      bDevice, cDevice);
#else
    auto kernel = program.MakeKernel("MatrixMultiplicationKernel", aDevice,
                                      bDevice, cDevice, size_n, size_k, size_m);
#endif

#ifdef MM_POWER_METER
    PowerMeter pm(10); // 10 ms sampling rate
    pm.Start();
#endif

    std::cout << "Executing kernel...\n" << std::flush;
    const auto elapsed = kernel.ExecuteTask();

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。


const auto perf = 1e-9 *
    (2 * static_cast<float>(size_n) * size_k * size_m) /
    elapsed.first;

std::cout << "Kernel executed in " << elapsed.first
    << " seconds, corresponding to a performance of " << perf
    << " GOp/s.\n";

```

計算 performance。

```

    if (verify) {
        std::cout << "Copying back result...\n" << std::flush;
        cDevice.CopyToHost(cMem.begin());
    }

} catch (std::runtime_error const &err) {
    std::cerr << "Execution failed with error: \'" << err.what() << "\'";
    << std::endl;
    return 1;
}

```

將 kernel 計算完的結果回傳到 Host。

```

// Run reference implementation
if (verify) {
    std::cout << "Running reference implementation...\n" << std::flush;
    ReferenceImplementation(a.data(), b.data(), cRef.data(), size_n, size_k,
                           size_m);

    std::cout << "Verifying result...\n" << std::flush;
    // Convert to single element vector
    const auto cTest = Unpack<kMemoryWidthM>(cMem);

    for (size_t i = 0; i < size_n; ++i) {
        for (size_t j = 0; j < size_m; ++j) {
            const auto testVal = make_signed<Data_t>(cTest[i * size_m + j]);
            const auto refVal = make_signed<Data_t>(cRef[i * size_m + j]);
            const Data_t diff = std::abs(testVal - refVal);
            bool mismatch;
            if (std::is_floating_point<Data_t>::value) {
                mismatch = diff / refVal > static_cast<Data_t>(1e-3);
            } else {
                mismatch = diff != 0;
            }
            if (mismatch) {
                std::cerr << "Mismatch at (" << i << ", " << j << "): " << testVal
                    << " vs. " << refVal << "\n";
            }
        }
    }
}

return 1;

```

驗證結果是否正確。

2. TOP.cpp

```

    // Memory accesses and pipes for A
#ifndef MM_TRANSPOSED_A
    Stream<Data_t, 2 * kOuterTileSizeN> aSplit[kTransposeWidth];
    #pragma HLS STREAM variable=aSplit depth=2*kOuterTileSizeN
    Stream<Data_t> aConvert("aConvert");
#else
    Stream<MemoryPackN_t, 2 * kOuterTileSizeNMemory> aMemory("aMemory");
#endif
    Stream<ComputePackN_t, kPipeDepth> aPipes[kComputeTilesN + 1];

```

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 width。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。

```

    // Memory accesses and pipes for B
    Stream<MemoryPackM_t, 2 * kOuterTileSizeMMemory> bMemory("bMemory");
    Stream<ComputePackM_t, kPipeDepth> bPipes[kComputeTilesN + 1];

```

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

```

// Pipes for C
Stream<ComputePackM_t> cPipes[kComputeTilesN + 1];

```

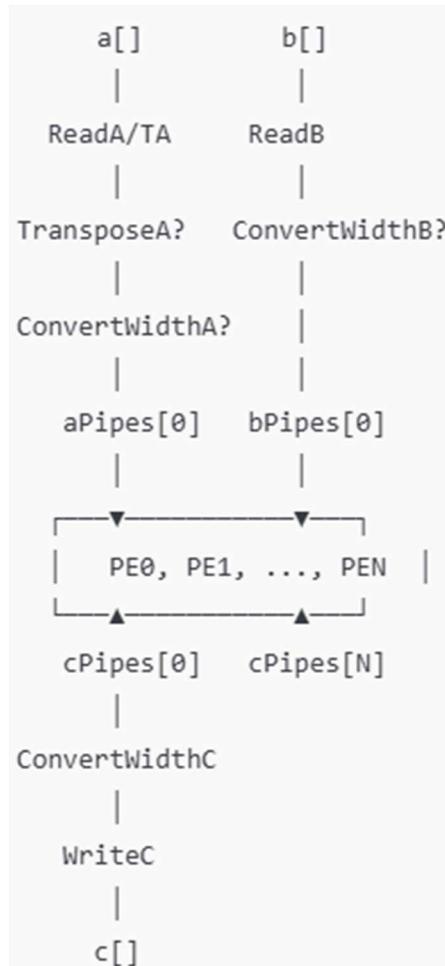
C 的話則直接用 cPipes[]把 PE 執行完的結果送回 memory。

```

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);
}

```

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。



3. Memory.cpp

```

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;
}

#else // MM_TRANSPOSED_A

unsigned IndexATransposed(const unsigned k, const unsigned n0,
                           const unsigned n1m, const unsigned size_n,
                           const unsigned size_k, const unsigned size_m) {
    #pragma HLS INLINE
    const auto index =
        k * SizeNMemory(size_n) + (n0 * kOuterTileSizeNMemory + n1m);
    // assert(index < size_k * SizeNMemory(size_n));
    return index;
}

```

對於未轉置矩陣 A， $\text{index} = n * \text{stride_k} + k$ ，適用於 column-major。而
 對於已轉置矩陣，則是以 row-major，因此 $\text{index} = k * \text{stride_n} + n$ 。

```

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;
}

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;
}

```

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。

```

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]);
        }
}

```

```

template <unsigned innerReads>
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);

```

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)。

```

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) {
                _ReadAIInnerLoop<kTransposeWidth / kMemoryWidthK>(
                    a, aSplit, n0, n1, k0, size_n, size_k, size_m);
            }
        }
    }
}
}

```

從 memory 讀取打包過的矩陣 A，拆解後推入 stream aSplit[]。
 n0(outer tile row index), m0(outer tile column index), k0(index of k dimension), n1(number of inner tile)。最後呼叫 ReadAIInnerLoop 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]。

```

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);
            }
        }
    }
}

```

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)。

```

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);
        }
    }
}

```

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。

```

#define MM_CONVERT_A
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);
            }
        }
#endif

```

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。

```

void ReadATransposed(MemoryPackN_t const memory[], Stream<MemoryPackN_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 * kouterTileSizeNMemory *
            MemoryPackN_t::kwidth) == TotalReadsFromA(size_n, size_k, size_m));

ReadA_OuterTile_N:
    for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
        ReadA_OuterTile_M:
            for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
                ReadA_K:
                    for (unsigned k = 0; k < size_k; ++k) {
                        ReadA_BufferA_N1:
                            for (unsigned n1m = 0; n1m < kOuterTileSizeNMemory; ++n1m) {
                                #pragma HLS PIPELINE II=1
                                #pragma HLS LOOP_FLATTEN
                                pipe.Push(
                                    memory[IndexATransposed(k, n0, n1m, size_n, size_k, size_m)]);
                            }
                        }
                    }
                }
}

```

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。

```

void ReadATransposed(MemoryPackN_t const memory[], Stream<MemoryPackN_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 * kouterTileSizeNMemory * 
            MemoryPackN_t::kwidth) == TotalReadsFromA(size_n, size_k, size_m));

    ReadA_OuterTile_N:
        for (unsigned n0 = 0; n0 < OuterTilesN(size_n); ++n0) {
    ReadA_OuterTile_M:
        for (unsigned m0 = 0; m0 < OuterTilesM(size_m); ++m0) {
    ReadA_K:
        for (unsigned k = 0; k < size_k; ++k) {
    ReadA_BufferA_N1:
        for (unsigned n1m = 0; n1m < kOuterTileSizeNMemory; ++n1m) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
            pipe.Push(
                memory[IndexATransposed(k, n0, n1m, size_n, size_k, size_m)]);
        }
    }
}
}
}

```

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。

```

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;
                                }
                            }
                        }
                    }
    }
}

```

n_0, m_0 是 outer tile 索引， n_1, m_{1m} 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。

4. Compute.cpp

```

ComputePackN_t aBuffer[2 * kInnerTilesN];

// This is where we spend all our T^2 fast memory
ComputePackM_t cBuffer[kInnerTilesN * kInnerTilesM][kComputeTileSizeN];
#pragma HLS ARRAY_PARTITION variable=cBuffer complete dim=2

```

`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。

```

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();
        }
    }
}

```

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。

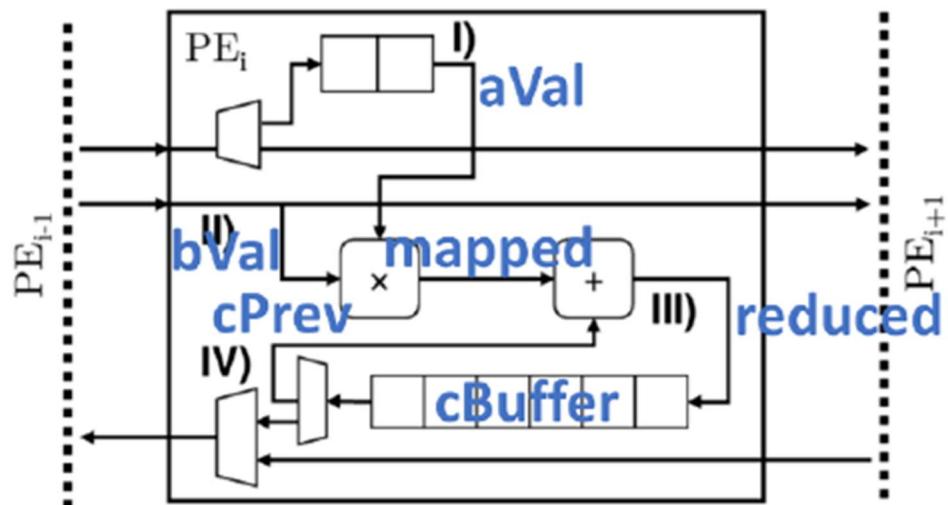
```

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);
    }
}
}
}

```

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。

Schematic



```

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

```

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。

```

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;
}

```

計算 $reduced = cPrev + (a \times b)$ ，使用兩個 Operator：

OperatorMap::Apply(a, b)：定義 $a \times b$ 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。

```

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;
        }
    }

```

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。

- I. 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。
- II. 原本我們打算在 U50 硬體上執行程式，但發現除了 Vitis 版本之外，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。
- III. 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
```

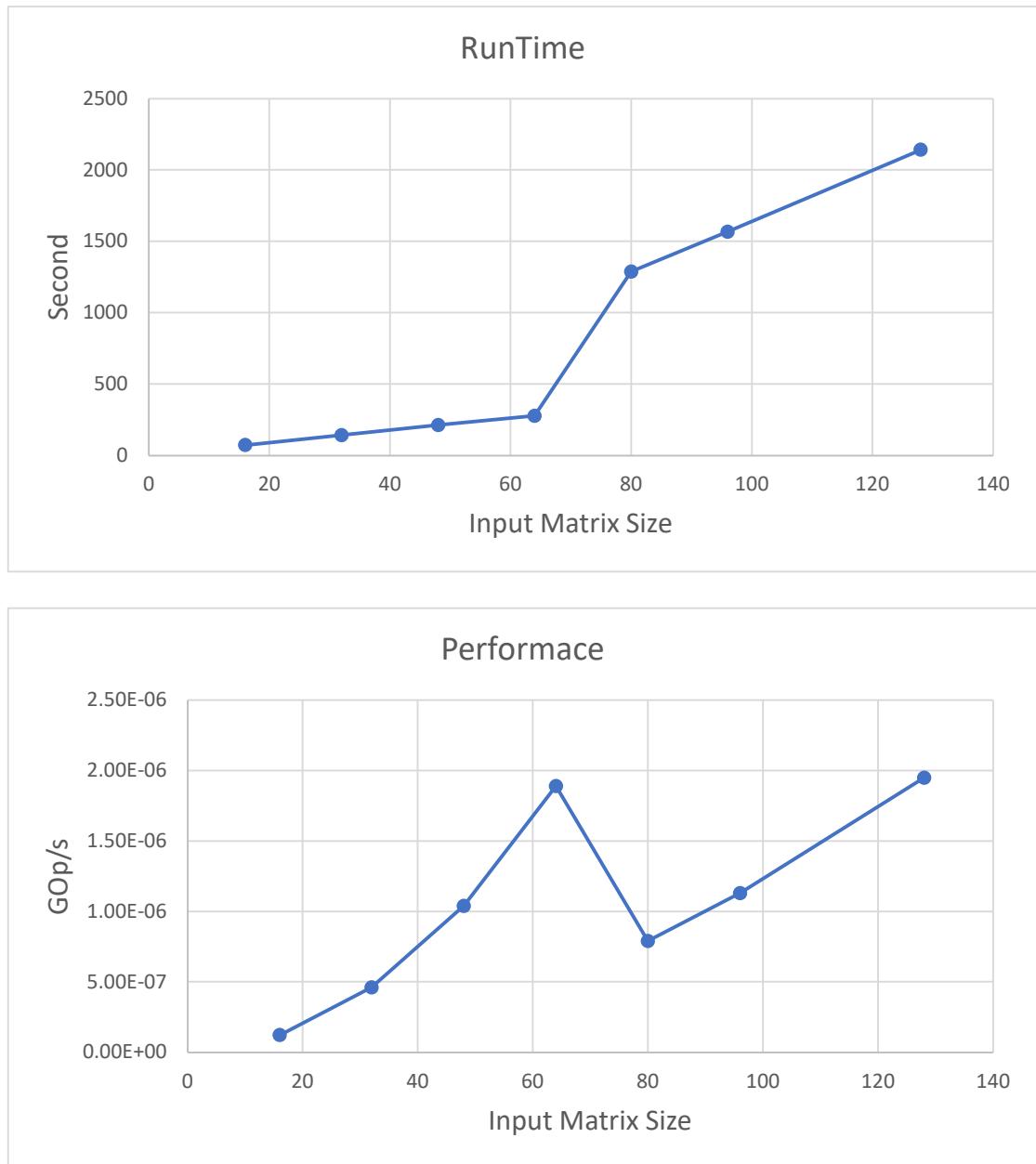
```
./RunHardware.exe 64 64 64 hw
```

```
Executing kernel...
Kernel executed in 277.022 seconds, corresponding to a performance of 1.89258e-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.
```

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。

- 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 Hoefer. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Denver, Colorado) (SC ’19)*.