Tiling/Blocking MHA, GEMM, LN, CONV

P.D.

Abstract

How a conversation with Zhongyi Lin about Multi-head attention became the starting point for a stand alone tiling tool for the search, description and computation of blocked algorithms for AIE overlays.

We start with a formal description of MHA and we jot down several ideas how to describe, code, and interpret tiling strategies. This is our first attempt to describe formally and optimally the tiling process. We started with pure tiling as suggestions to an FPGA IP (XDNN), with layer fusion and explicit double buffering by software pipelining (DPUV3INT8). We introduce explicit code and compute parallelism by symmetric computations for sparse convolutions (Weight block sparsity). Here, we finally explore all valid solutions and optimum by any custom function with the ability to represent complex computation using double buffering at all level of the memory hierarchy without being hold back by how to implement each instructions (but having done so in the past).

Index Terms

AI, FPGA, Performance, LLM BS and Tools

I. MHA, its Notations and an Introduction

This is a HEAD

$$\boldsymbol{R} = \mathcal{S}_{max}(\boldsymbol{Q} * \boldsymbol{K}^t) * \boldsymbol{V} \tag{1}$$

Where $\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V} \in \mathbb{R}^{L \times d}$, where $d \in \{32, 64\}$ is small and $L \in \{4096\}$ is large

$$\mathcal{S}_{max}(\boldsymbol{M}) = \boldsymbol{S}, \text{ with } \boldsymbol{S}, \boldsymbol{M} \in R^{m \times n} \tag{2}$$

$$\mathcal{S}_{max}(\boldsymbol{M})_{i,j} = \frac{\exp \boldsymbol{M}_{i,j}}{\sum_j \exp \boldsymbol{M}_{i,j}} \tag{3}$$

Where the denominator is a sum by row and the result is literally a vector normalizing the row of hte numerator. Another way is to consider it a unitary matrix multiplying on the left

$$\Gamma = \frac{1}{\sum_j \exp M_{i,j}} = [\frac{1}{\sum_j \exp M_{0,j}}, \ldots, \frac{1}{\sum_j \exp M_{m,j}}] * \boldsymbol{I} \tag{4}$$

$$\mathcal{S}_{max}(\boldsymbol{M}) = \Gamma \boldsymbol{M} = \boldsymbol{S} \tag{5}$$

$$\boldsymbol{Q} * \boldsymbol{K}^t = \boldsymbol{T} \in \mathbb{R}^{L \times L} \tag{6}$$

The submatrix of $\boldsymbol{K}^t$ are actually transposed, so I do not need to carry the transpose sign everywhere. Assume that $\boldsymbol{K}_i, \boldsymbol{Q}_j \in \mathbb{R}^{d \times d}$

$$\begin{pmatrix} \boldsymbol{Q}_0 \\ \boldsymbol{Q}_1 \\ \ldots \\ \boldsymbol{Q}_n \end{pmatrix} * \begin{pmatrix} \boldsymbol{K}_0 & \boldsymbol{K}_1 & \ldots \boldsymbol{K}_n \end{pmatrix} = \begin{pmatrix} \boldsymbol{Q}_0\boldsymbol{K}_0 & \boldsymbol{Q}_0\boldsymbol{K}_1 & \ldots \boldsymbol{Q}_0\boldsymbol{K}_n \\ \boldsymbol{Q}_1\boldsymbol{K}_0 & \boldsymbol{Q}_1\boldsymbol{K}_1 & \ldots \boldsymbol{Q}_1\boldsymbol{K}_n \\ \ldots & & \\ \boldsymbol{Q}_n\boldsymbol{K}_0 & \boldsymbol{Q}_n\boldsymbol{K}_1 & \ldots \boldsymbol{Q}_n\boldsymbol{K}_n \end{pmatrix}$$

$$\boldsymbol{R}_0 = \mathcal{S}_{max}(\begin{pmatrix} \boldsymbol{Q}_0\boldsymbol{K}_0 & \boldsymbol{Q}_0\boldsymbol{K}_1 & \ldots & \boldsymbol{Q}_0\boldsymbol{K}_n \end{pmatrix}) \begin{pmatrix} \boldsymbol{V}_0 \\ \boldsymbol{V}_1 \\ \ldots \\ \boldsymbol{V}_n \end{pmatrix}$$

To be precise,

$$\boldsymbol{R}_0 = \frac{1}{\sum_j \exp \boldsymbol{Q}_0\boldsymbol{K}_j} \begin{pmatrix} \exp \boldsymbol{Q}_0\boldsymbol{K}_0 & \ldots & \exp \boldsymbol{Q}_0\boldsymbol{K}_n \end{pmatrix} \begin{pmatrix} \boldsymbol{V}_0 \\ \boldsymbol{V}_1 \\ \ldots \\ \boldsymbol{V}_n \end{pmatrix} \tag{7}$$

$$\boldsymbol{R}_0 = \frac{\sum_j (\exp \boldsymbol{Q}_0\boldsymbol{K}_j)\boldsymbol{V}_j}{\sum_j \exp \boldsymbol{Q}_0\boldsymbol{K}_j} = \Gamma_0 \sum_j (\exp \boldsymbol{Q}_0\boldsymbol{K}_j)\boldsymbol{V}_j \tag{8}$$

The sum at the denominator is by row, the result is a vector and it can be represented also as a left diagonal matrix multiplication using Equation 4. The sum at the numerator is a matrix addition (or a left matrix multiplication and thanks to associative). Thus, Equations 7 and 8 are proper and equivalent. We can apply to the remaining $\boldsymbol{Q}_i$ and $\boldsymbol{R}_i$.

A. Range of $\exp(\boldsymbol{Q}_0\boldsymbol{K}_j)$

The function e^x has the property that $\frac{\partial e^x}{\partial x} = e^x$. The function e^x is always positive, is always increasing, and $e^0 = 1$ and also the derivative. For $x > 0$ and with numerical representation that could be limited in range, we want to stay on the left side of $x = 0$ or at a very least in a bound region to avoid saturation and overflow. Clearly, for $x > 0$ we have $\frac{\partial e^x}{\partial x} > 1$ and it really highlights small variation of x.

If we keep the range of $\boldsymbol{Q}_0\boldsymbol{K}_j$ being smaller than one, then the $\exp()$ is bound to e. Due to the nature of the exponential function, the range will be positive and increasing (well exponentially). For example, assume we compute the product $\boldsymbol{QK}$ and we determine the maximum by row. This is a column vector $\boldsymbol{M} = \max_{row} \boldsymbol{QK}$ then we can compute the final result.

$$\boldsymbol{R}_0^t = \frac{\sum_j (\exp(\boldsymbol{Q}_0\boldsymbol{K}_j - \boldsymbol{M}))\boldsymbol{V}_j}{\sum_j \exp(\boldsymbol{Q}_0\boldsymbol{K}_j - \boldsymbol{M})} = \boldsymbol{R}_0 \frac{e^{\boldsymbol{M}}}{e^{\boldsymbol{M}}} = \boldsymbol{R}_0 \tag{9}$$

The exponential range is under control, the summation at the denominator is increasing but bound to the number of additions. The denominator can be applied at the end of the computation but the normalization of the range as to be applied as soon as possible in such a way to have undesired overflow. In Section II, we combine everything into an iterative method that can be split into independent computations.

B. The identity element in $\mathcal{S}_{max}(\boldsymbol{M})$

In the ring based on the operation addition $+$, there is a identity element 0 (zero) such as $\forall a, a + 0 = 0 + a = a$. For multiplication $*$ the identity element is 1 (one). If the operands are matrices, we can extend the zero and one elements to zero matrices and to diagonal one matrices.

Take $\frac{\exp \boldsymbol{M}_{i,j}}{\sum_j \exp \boldsymbol{M}_{i,j}}$, if for any reason we need to pad or extend $\boldsymbol{M}$ to $\boldsymbol{N}$, what can we do so that $\frac{\exp \boldsymbol{M}_{i,j}}{\sum_j \exp \boldsymbol{M}_{i,j}} = \mathcal{S}_{max}(\boldsymbol{N})$? How do we pad it?

- **Padding the inputs before matrix multiplication will not work.** For example, take the last $\boldsymbol{K}_n \in d \times m$ and for the computation of the matrix multiplication $\boldsymbol{Q}_0\boldsymbol{K}_n$ we must increase m to $m+1$ (say, alignment properties). This may affect also $\boldsymbol{V}_n$. This will change the shape of the results and in practice, we would introduce d zeros into the matrix results. This will introduce 1 into the matrix $\exp(\boldsymbol{Q}_0\boldsymbol{K}_n)$ and when applied to $\mathcal{S}_{max}(\boldsymbol{Q}_0\boldsymbol{K}_n)$, then we change the shape of the numerator and we increase the denominator (by one in this case). If we need to compute the max for range purpose as in Equation 9, we are changing also the values of the numerator matrix and given the $\max$ is not bijective, we cannot reverse the effects. Padding the operands and in combination with range constraints, we cannot have a clean computation.
- **Padding the output of $\exp(\boldsymbol{M})$ with zeros.** In practice, the unitary element of the denominator sum is zero $\exp(-\infty)$, the unitary element for $\max$ is $-\infty$. Either pad the output with zero or the input with $-\infty$. In finite precision and for $\max$, any $\min$ instead of $-\infty$ value will work, however, the $\exp(\min)$ will not translate to zero in floating point representation unless we fix the result to zero.

  How do we introduce $-\infty$? or how we introduce zeros to a specific output pattern?

II. MHA, Blocked and Iterative Method

Take $\boldsymbol{N}_{0,0} = \boldsymbol{D}_{0,0} = 0$, $\boldsymbol{M} = \boldsymbol{0}$, $\boldsymbol{M}_1 = \max_{row}(\boldsymbol{Q}_m\boldsymbol{K}_t, M)$ and $\boldsymbol{E}_t^m = \exp(\boldsymbol{Q}_m\boldsymbol{K}_t - \boldsymbol{M}_1)$, and $\boldsymbol{S} = \exp(\boldsymbol{M}_1 - \boldsymbol{M})$ assume we have done the computation until step t and compute $t+1$ to $t = n$ then

$$\boldsymbol{N}_{0,t+1} = (\ \frac{\boldsymbol{N}_{0,t}}{\boldsymbol{S}} + (\boldsymbol{E}_{t+1}^0)\boldsymbol{V}_{t+1}\ ) \tag{10}$$

$$\boldsymbol{D}_{0,t+1} = \frac{\boldsymbol{D}_{0,t}}{\boldsymbol{S}} + \boldsymbol{E}_{t+1}^0 \tag{11}$$

$$\boldsymbol{R}_0 = \frac{\boldsymbol{N}_{0,n}}{\boldsymbol{D}_{0,n}} \tag{12}$$

This scaling or averiging is actually an element wise operation.

For every element of $\boldsymbol{Q}_i$ we can repeat the process and compute the final result

$$\forall i \in [0, n], \boldsymbol{R}_i = \frac{\boldsymbol{N}_{i,n}}{\boldsymbol{D}_{i,n}} \tag{13}$$

Space requirements: $\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i \in \mathbb{R}^{d\times d}$,, $\boldsymbol{Q}_i\boldsymbol{K}_j \in \mathbb{R}^{d\times d}$, $\exp \boldsymbol{Q}_i\boldsymbol{K}_j \in \mathbb{R}^{d\times d}$ $\boldsymbol{D}_{0,i} \in \mathbb{R}^d$, $\boldsymbol{N}_{0,t}\boldsymbol{N}_{0,t+1} \in \mathbb{R}^{d\times d}$.

We need to keep in memory $\boldsymbol{D}_{0,t}$ and $\boldsymbol{N}_{0,t}$, we compute $\exp \boldsymbol{Q}_0\boldsymbol{K}_{t+1}$ to add to $\boldsymbol{D}_{0,t}$, we then compute $(\exp \boldsymbol{Q}_0\boldsymbol{K}_{t+1})\boldsymbol{V}_{t+1}$. We need 5 matrices of size $d \times d$. We store $\boldsymbol{D}_{0,t+1}$ and $\boldsymbol{N}_{0,t+1}$.

A. Implementation of the head

We discuss here the case where where we perform the operations with *int16*. Consider $L = 512$, we split the matrices $\boldsymbol{Q}$ and $\boldsymbol{V}$ into horizontal parts and we split $\boldsymbol{K}^t$ into vertical parts. We choose to split $\boldsymbol{K}^t$ and $\boldsymbol{V}$ equally into four core so each core will receive $2^6 \times \lceil \frac{2^9}{2^2} \rceil = 2^7$ for a space of 2^{14} Bytes and we need to store the same amount of $\boldsymbol{V}$ for a total space of 2^{15} this is equivalent to four banks. We can split the computation so that $\boldsymbol{V}_i, \boldsymbol{K}_i \in 64 \times 64$ and thus we send into each core 2 $\boldsymbol{V}_i$ and 2 $\boldsymbol{K}_i$.

Let consider what should be the size of $\boldsymbol{Q}_0 = m \times 64$ so that we can fit a basic computation in a core.

$\boldsymbol{T} = \boldsymbol{Q}_0 \boldsymbol{K}_i = m \times 64$, $\boldsymbol{T} = \exp \boldsymbol{T} = m \times 64$, $\boldsymbol{D} += \boldsymbol{T}$, $\boldsymbol{T} = \boldsymbol{T}\boldsymbol{V}_i$, and $\boldsymbol{N} += \boldsymbol{T}$. We need to have space for 4 matrices matrices of size $m \times 64$. Thus $m 2^3 2^6$ must fit into 2 banks 2^{14}. We have $m = 2^5 = 32$.

Core A_i compute $N_{i*2,(i+1)*2-1}$ and $D_{i*2,(i+1)*2-1}$ With three reductions per column we have the computation of R_0, we repeat the process. The subvolume for $\boldsymbol{Q}_i = 32 \times 64$. We need to broad cast $\boldsymbol{Q}_i$ 16 times.

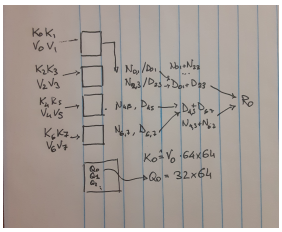

Fig. 1. HEAD for int16

In this section, we show the algorithm that can be implemented on a 2×4 having the problem size $d = 64$ and $L = 512$

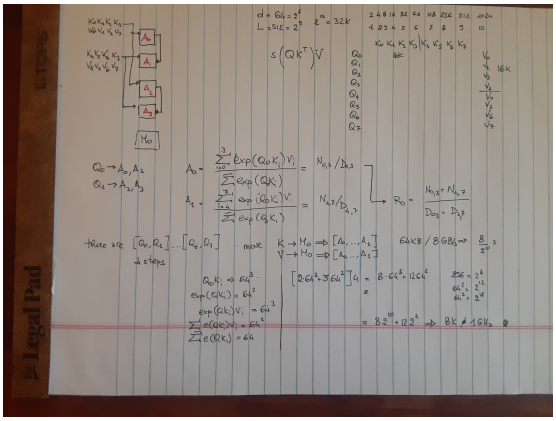

Fig. 2. HEAD for int 8

B. The curious case of the problem size $l = 77$ and $L = 768$

This is the place holder for the curious case and may be a code generator.

III. GEMM FOR ASYMMETRIC SYSTOLIC ARRAYS

We start a conversation about $\boldsymbol{C} = \boldsymbol{A} * \boldsymbol{B}$ where we imply that the right operand is larger than the left operand. If it may happen the opposite, then we may want to compute $\boldsymbol{C}^t = \boldsymbol{B}^t * \boldsymbol{A}^t$, the shapes may be different but the sizes stay the same. The algorithm in a nut shell is a systolic algorithm where the matrix C is spatially separated and computed completely before written. Any matrices we are going to present in this section will have the usual notation

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}_{0,0} & \boldsymbol{M}_{0,1} & \boldsymbol{M}_{0,2} & \boldsymbol{M}_{0,3} & \dots \\ \boldsymbol{M}_{1,0} & \boldsymbol{M}_{1,1} & \boldsymbol{M}_{1,2} & \boldsymbol{M}_{1,3} & \dots \\ \boldsymbol{M}_{2,0} & \boldsymbol{M}_{2,1} & \boldsymbol{M}_{2,2} & \boldsymbol{M}_{2,3} & \dots \\ \boldsymbol{M}_{3,0} & \boldsymbol{M}_{3,1} & \boldsymbol{M}_{3,2} & \boldsymbol{M}_{3,3} & \dots \\ \dots & & & & \end{pmatrix}$$

Where each element is a sub matrix. We will refer these as subvolumes but they can be reduce to scalar if necessary. We assume an array of AIEs in a formation 4×2. Two columns and four rows.

$$\begin{pmatrix} \boldsymbol{C}_{0,0} = \sum_k \boldsymbol{A}_{0,k}\boldsymbol{B}_{k,0} & \boldsymbol{C}_{1,0} \sum_k \boldsymbol{A}_{1,k}\boldsymbol{B}_{k,0} \\ \boldsymbol{C}_{0,1} = \sum_k \boldsymbol{A}_{0,k}\boldsymbol{B}_{k,1} & \boldsymbol{C}_{1,1} \sum_k \boldsymbol{A}_{1,k}\boldsymbol{B}_{k,1} \\ \boldsymbol{C}_{0,2} = \sum_k \boldsymbol{A}_{0,k}\boldsymbol{B}_{k,2} & \boldsymbol{C}_{1,2} \sum_k \boldsymbol{A}_{1,k}\boldsymbol{B}_{k,2} \\ \boldsymbol{C}_{0,3} = \sum_k \boldsymbol{A}_{0,k}\boldsymbol{B}_{k,3} & \boldsymbol{C}_{1,3} \sum_k \boldsymbol{A}_{1,k}\boldsymbol{B}_{k,3} \end{pmatrix} \tag{14}$$

$\boldsymbol{A}_{0,k}$ is broad cast on column zero and $\boldsymbol{A}_{1,k}$ to column one, one subvolume at a time. $\boldsymbol{B}_{k,0}$ is broad cast on row zero and $\boldsymbol{B}_{k,3}$ on row three, one subvolume at a time. This is a systolic communication.

A. Sub-Volume

The basic computation for one core is $\boldsymbol{C}_{i,j} = \sum_k \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$, the basic subvolume for this computation is $\boldsymbol{C}_{i,j}, \boldsymbol{A}_{i,k}, \boldsymbol{B}_{k,j}$. They are not same-size matrices. They need to be in the AIE cores. This has eight banks of 8 KB each (2^{13} elements). We assume that the computation and communication to the core is double buffered.

Algorithm 1 Core streaming computation for $\boldsymbol{C}_{0,0}$

1: $\boldsymbol{T}_A = \boldsymbol{A}_{0,0}, \boldsymbol{T}_B = \boldsymbol{B}_{0,0}$
2: $\forall i[1, K-1], \boldsymbol{TT}_A = \boldsymbol{A}_{0,i}, \boldsymbol{TT}_B = \boldsymbol{B}_{i,0}, \boldsymbol{C}_{0,0} += \boldsymbol{T}_A\boldsymbol{T}_B, \boldsymbol{T}_A = \boldsymbol{TT}_A, \boldsymbol{T}_B = \boldsymbol{TT}_B$
3: $\boldsymbol{C}_{0,0} += \boldsymbol{T}_A\boldsymbol{T}_B$

We decide sub-volume $\boldsymbol{C}_{0,0}$ will fit a bank 2^{13}, to keep the most balance computation we choose $(128 = 2^7) \times (64 = 2^6)$ (8 KB). This means that $\boldsymbol{A}_{0,0}$ will be 128×64 (8 KB) and $\boldsymbol{B}_{0,0}$ will be 64×64 (4 KB).

One step (tick) of the overlay computation is an *outer* product

$$\begin{pmatrix} \boldsymbol{C}_{0,0} & \boldsymbol{C}_{0,1} & \boldsymbol{C}_{0,2} & \boldsymbol{C}_{0,3} \\ \boldsymbol{C}_{1,0} & \boldsymbol{C}_{1,1} & \boldsymbol{C}_{1,2} & \boldsymbol{C}_{1,3} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{0,k} \\ \boldsymbol{A}_{1,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,0} & \boldsymbol{B}_{k,1} & \boldsymbol{B}_{k,2} & \boldsymbol{B}_{k,3} \end{pmatrix} \tag{15}$$

We read 16 KB of $\boldsymbol{A}$ (256×64) , we read 16 KB of $\boldsymbol{B}$ (64×256), and we compute 256×256 elements of $\boldsymbol{C}$. In practice, the full computation is formally, where we introduce a step in the outer loop:

$$\sum_{i=0,2}^{M} \sum_{j=0,4}^{N} \begin{pmatrix} \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3} \\ \boldsymbol{C}_{i+1,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3} \end{pmatrix} = \sum_{k}^{K} \begin{pmatrix} \boldsymbol{A}_{i,k} \\ \boldsymbol{A}_{i+1,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3} \end{pmatrix}$$

We have a ratio compute over communication per core

$$\mathbb{F} = \frac{2^7 * 2^6 * 2^7}{2 * 2^7 * 2^6} = \frac{2^{20}}{2^{14}} = 2^6 \tag{16}$$

This ratio is useful to estimate the smallest subvolume we can afford in order to hide communication latency by double buffering or figure out the ratio between the computation speed and bandwidth.

For example, assume we can send each operand by different channels at 8 GB/s = $2^3 * 2^{30}$ and 256 operations can be done in one cycle at 1 GHz. The ratio above will become

$$\mathbb{F}_c = \frac{2^7 * 2^6 * 2^7 * 2^{33}}{2 * 2^7 * 2^6 * 2^8 * 10^9} \sim \frac{2^{53}}{2^{14+8+28=50}} > 2^3$$

For any values larger than one, the execution time is compute bound. In the same way we can choose the operand sizes:

$$\frac{2^3 * 2^3 * 2^3 * 2^{33}}{2 * 2^3 * 2^3 * 2^8 * 10^9} \sim \frac{2^{42}}{2^{7+8+28=43}} > 1/2$$

any sub volume smaller than 8×8 will be communication bound and double buffering will not help. This Ratio can be used for the investigation of double buffering and latency of reading operands in Mem-Tiles.

1) Transpose Algorithm: Because the sizes and shapes of the operands does not need to be the same, we may need to consider also the transpose algorithm

$$\begin{pmatrix} \boldsymbol{C}_{0,0} & \boldsymbol{C}_{0,1} \\ \boldsymbol{C}_{1,0} & \boldsymbol{C}_{1,1} \\ \boldsymbol{C}_{2,0} & \boldsymbol{C}_{2,1} \\ \boldsymbol{C}_{3,0} & \boldsymbol{C}_{3,1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{0,k} \\ \boldsymbol{A}_{1,k} \\ \boldsymbol{A}_{2,k} \\ \boldsymbol{A}_{3,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,0} & \boldsymbol{B}_{k,1} \end{pmatrix} \tag{17}$$

We read 16 KB of $\boldsymbol{A}$ (64×256) , we read 16 KB of $\boldsymbol{B}$ (256×64), and we compute 256×256 elements of $\boldsymbol{C}$. In practice, the full computation is formally, where we introduce a step in the outer loop:

$$\sum_{i=0,4}^{M} \sum_{j=0,2}^{N} \begin{pmatrix} \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} \\ \boldsymbol{C}_{i+1,j} & \boldsymbol{C}_{i+1,j+1} \\ \boldsymbol{C}_{i+2,j} & \boldsymbol{C}_{i+2,j+1} \\ \boldsymbol{C}_{i+3,j} & \boldsymbol{C}_{i+3,j+1} \end{pmatrix} = \sum_{k}^{K} \begin{pmatrix} \boldsymbol{A}_{i,k} \\ \boldsymbol{A}_{i+1,k} \\ \boldsymbol{A}_{i+2,k} \\ \boldsymbol{A}_{i+3,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} \end{pmatrix}$$

The main difference and utility to have two algorithms (for memtile-core) is the better way to have a different ratio of the computation. As a practical point, Equation 15 leads to a reuse of $\boldsymbol{A}$ in memtile (if $\boldsymbol{A}$ smaller space) and Equation 17 leads to a reuse of $\boldsymbol{B}$ (if $\boldsymbol{B}$ is smaller). This will be evident when we write and present a code generator based on this document. For

2) Different sizes: int16: In the case, we are working with an element 2 Bytes (int 16) and we can keep the same subvolumes, the fraction $\mathbb{F} = \frac{2^6}{2}$. The number of computation does not change, the communication increases linearly.

The sub-volume computations has to change. We take $VcC_{0,0} = 64 \times 64 = 2^6 \times 2^6$, this means of size $2 * 2^12$, this is 8 KB (again). $\boldsymbol{A}_{0,k}, \boldsymbol{B}_{k,0} = 64 \times 64$, one bank again. The subvolume computation is $64 \times 64 \times 64$. Changing the subvolume does not affect the Bytes transfer per tick (16 KB for each operand). This is still balanced.

B. Mem-tiles and the case $A = 512 \times 768$ $B = 768 \times 768$

The matrix $\boldsymbol{A}$ has size $(512 = 2^9) \times (768 = 3 * 2^8)$ that is $3 * 2^{17}$ and this fits nicely in a memtile of size 512 KB, that is 2^{19} elements. The matrix $\boldsymbol{B}$ has size 768×768 that is $9 * 2^{16} > 2^{19}$. The nice property is that we can split the matrix $\boldsymbol{B}$ by column into three parts exactly and we can do double buffering

$$\sum_{j=0,4}^{12} \{ \sum_{i=0,2}^{4} \begin{pmatrix} \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3} \\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3} \end{pmatrix} = \sum_{k}^{12} \begin{pmatrix} \boldsymbol{A}_{i,k} \\ \boldsymbol{A}_{i+1,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3} \end{pmatrix} \}$$

The curly brackets show the part of the computation when the sub matrix of $\boldsymbol{B}$ of size 768×256 is double buffered.

C. Mem-tiles and the case int16 $A = 512 \times 768$ $B = 768 \times 768$

The matrix $\boldsymbol{A}$ has size $(512 = 2^9) \times (768 = 3 * 2^8)$ that is $2 * 3 * 2^{17}$, We need to tile and double buffering $\boldsymbol{A}$. We need need to feed two columns: $A_{0,*}$ and $A_{1,*}$ at a time. One ping per column will be $2 * (64)$: If we split $\boldsymbol{A}$ into 4 tiles of 128 rows (4 subvolumes), We can have 4 iterations where we double buffer $A_{i,*}$ and $A_{i,*}$. We need to keep a space of 384 KB in memtile 0.

The matrix $\boldsymbol{B}$ has size 768×768 that is $2 * 9 * 2^{16} > 2^{19}$. If we keep the same sub-volume, we need $\boldsymbol{B}_{*,0} \dots \boldsymbol{B}_{*,3}$ thus $2 * 256 \times 768$, four double buffering we need twice as much as that. To make this possible we need to change the subvolume of $\boldsymbol{B}$ and of $\boldsymbol{C}$.

To allow double buffering of the $\boldsymbol{B}$ in memtile, The sub-volume computations has to change. We take $VcC_{0,0} = 64 \times 32 = 2^6 \times 2^5$, this means of size $2 * 2^{11}$, this is 4 KB (again). $\boldsymbol{A}_{0,k} = 64 \times 64$ one bank, $\boldsymbol{B}_{k,0} = 64 \times 32$, 4 KB. The subvolume computation is $64 \times 64 \times 32$. Changing the subvolume does affect the Bytes transfer per tick (16 KB for A and 8KB for B).

$$\sum_{j=0,4}^{24} \{ \sum_{i=0,2}^{8} \begin{pmatrix} \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3} \\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3} \end{pmatrix} = \sum_{k}^{12} \{ \begin{pmatrix} \boldsymbol{A}_{i,k} \\ \boldsymbol{A}_{i+1,k} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3} \end{pmatrix} \} \}$$

$$\mathbb{F}_{cm} = \frac{2^6 * 2^6 * 2^5 / 128 / 10^9}{2(2^{12} + 2^{11})/8/2^{30}} = 0.75$$

This is communication bound. If we consider the computation of a core and the time to read a ping of $\boldsymbol{B}$ from DDR to memtile (considering we can use 2 channels 8 GBs). We can reach break even with the computation if we use 4 channels (copy to both Memtiles)

$$\mathbb{F}_{md} = \frac{2^6 * 3 * 2^8 * 2^5 / 128 / 10^9}{2 * 2^7 * 32^8 / (2 * 8 * 2^{30}} = 0.53$$

If we assume we can use 4 channels for DDR. The execution time will be $6 * 8 * 12 * 2(2^{12} + 2^{11})/8/2^{30}$ which is 411 us

Algorithm 2 Small Small still need tiling

1: **for** i=0 **to** 8 step 2 **do**
2: Ping/Pong $\boldsymbol{A}_{i,*}..\boldsymbol{A}_{i+1,*}$
3: # Reuse A but hide latency
4: **for** j=0 **to** 24 step 4 **do**
5: Ping/Pong $\boldsymbol{B}_{*,j}\dots\boldsymbol{B}_{*,j+3}$
6: $\begin{pmatrix}\boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3}\\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3}\end{pmatrix} = \sum_k^{12}\begin{pmatrix}\boldsymbol{A}_{i,k}\\ \boldsymbol{A}_{i+1,k}\end{pmatrix}\begin{pmatrix}\boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3}\end{pmatrix}$
7: **end for**
8: **end for**

D. Mem-tiles and the case $A = 512 \times 768$ $B = 768 \times 768 * 4$

The matrix $\boldsymbol{A}$ has size $(512 = 2^9) \times (768 = 3 * 2^8)$ that is $3 * 2^{17}$ and this fits nicely in a memtile of size 512 KB, that is 2^{19} elements. The matrix $\boldsymbol{B}$ has size $768 \times 768 * 4$ that is $9 * 2^{18} > 2^{19}$. The nice property is that we can split the matrix $\boldsymbol{B}$ by column into 12 parts exactly and we can do double buffering

$$\sum_{j=0,4}^{48}\{\sum_{i=0,2}^{4}\begin{pmatrix}\boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3}\\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3}\end{pmatrix} = \sum_k^{12}\begin{pmatrix}\boldsymbol{A}_{i,k}\\ \boldsymbol{A}_{i+1,k}\end{pmatrix}\begin{pmatrix}\boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3}\end{pmatrix}\}$$

E. Mem-tiles and the case $A = 512 \times 768 * 4$ $B = 768 * 4 \times 768$

The matrix $\boldsymbol{A}$ has size $(512 = 2^9) \times (768 = 3 * 2^{10})$ that is $3 * 2^{19}$. The matrix $\boldsymbol{B}$ has size $768 * 4 \times 768$ that is $9 * 2^{18} > 2^{19}$.

We present two algorithms. The first algorithm change the subvolume so that we can double buffer A and reuse a slice of A multiple times.

Consider first that a matrix of size 512×768 fits a Memtile. A slice of $\boldsymbol{A}$ of size $64 \times 768 * 4$ allows a double buffering. This means that a column will solve a subvolume of size $m = 32$ (64 divided by 2). Equivalently, a slice of $\boldsymbol{B}$ of size $t68 * 4 \times 64$ will allow double buffering and thus a sub volume of $n = 16$ (64 divided by 4).

$\boldsymbol{C}_{i,j}$ has size 32×16. $\boldsymbol{A}_{i,k}$ has size 32×64. $\boldsymbol{B}_{k,j}$ has size 64×16.

$$\sum_{i=0,2}^{8}\{\sum_{j=0,4}^{96}\begin{pmatrix}\boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3}\\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3}\end{pmatrix} = \sum_k^{192}\{\begin{pmatrix}\boldsymbol{A}_{i,k}\\ \boldsymbol{A}_{i+1,k}\end{pmatrix}\begin{pmatrix}\boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3}\end{pmatrix}\}\}$$

Algorithm 3 Large Large

1: **for** i=0 **to** 8 step 2 **do**
2: Ping/Pong $\boldsymbol{A}_{i,*}..\boldsymbol{A}_{i+1,*}$
3: # Reuse A but hide latency
4: **for** j=0 **to** 48 step 4 **do**
5: Ping/Pong $\boldsymbol{B}_{*,j}\dots\boldsymbol{B}_{*,j+3}$
6: $\begin{pmatrix}\boldsymbol{C}_{i,j} & \boldsymbol{C}_{i,j+1} & \boldsymbol{C}_{i,j+2} & \boldsymbol{C}_{i,j+3}\\ \boldsymbol{C}_{i,j} & \boldsymbol{C}_{i+1,j+1} & \boldsymbol{C}_{i+1,j+2} & \boldsymbol{C}_{i+1,j+3}\end{pmatrix} = \sum_k^{192}\begin{pmatrix}\boldsymbol{A}_{i,k}\\ \boldsymbol{A}_{i+1,k}\end{pmatrix}\begin{pmatrix}\boldsymbol{B}_{k,j} & \boldsymbol{B}_{k,j+1} & \boldsymbol{B}_{k,j+2} & \boldsymbol{B}_{k,j+3}\end{pmatrix}$
7: **end for**
8: **end for**

IV. GEMM Tiles Generator

In general, the core computation needs to be a multiple aligned to 8, say, for the channels (columns of the operands). Subvolumes being a power of two are nice and not necessary. What if we can explore the sub-volume solution space with some type of properties? In Equation 16, the ratio $\mathbb{F}$ is a function of the subvolume ($\mathbb{F}(m,n,k)$). This sub-volume can be for a core (streaming) or the representation of the cluster computation (all cores with all memtiles). We argue for three requirements: we should have a subvolume that fit into a core (basic subvolume), we want to have largest number of operations per subvolume ($\mathbb{C}(m,n,k)$), and among them we prefer the one that have the largest ratio $\mathbb{F} = \frac{\mathbb{C}}{\mathbb{S}}$ (where $\mathbb{S}$ stands for space ans 24 KB represent the space for 6 banks in a core). We can call $\mathbb{T}$ the set of valid solutions and we call $\mathbb{W}$ the sorted solution space as follows:

$$\mathbb{T} = \{[m,n,k] \text{ s.t. } \mathbb{S}(m,n,k) < 24KB\} \tag{18}$$

$$\mathbb{W} = sort(\mathbb{T}, \lambda x : (-\mathbb{C}(x), -\mathbb{F}(x)) \tag{19}$$

We call $\mathbb{Q}_0$ the sum over k of Equation 15 and $\mathbb{Q}_1$ of Equation 17. These represent two algorithms and also subproblems. The cluster of cores work together reading from Mem-Tile and in the context of the larger problem we can double buffer either $\boldsymbol{A}$ or $\boldsymbol{B}$.

If we chose $(m,n,k) = \mathbb{W}[0]$ as starting block for the core computation. Then $\mathbb{Q}_0$ will have a memtile subvolume $M_0 = (2*m, 4*n, K)$ and $\mathbb{Q}_1$ $M_1 = (4*m, 2*n, K)$. Both computations will be the reduction of K/k iterations. And we can choose the algorithm so that $\mathbb{F}(M_0)$ and $\mathbb{F}(M_1)$ is the largest. The core subvolume is determined by space and $\mathbb{F}$, so it is the Mem-tile subvolume and thus the algorithm itself.

At this time, we will not deal with padding and alignment per core and per memtile. We summarily say that the core subvolume (m,n,k) will determine a minimum zero padding so that a core will produce the correct result. Padding at cluster level does not need to be, we can accept either the turn off or columns/rows or the computation of garbage and not writing back to DDR.

A. Double Buffering aka DB

Rarely $\boldsymbol{A}$ and $\boldsymbol{B}$ will fit the Memtile and some double buffering may be required. Given a core subvolume, If we can do double buffering for both operands, there is no problem.

$\mathbb{Q}_1$ is more efficient for large $\boldsymbol{B}$, we tend to reuse $\boldsymbol{A}$, and at a minimum we need to double buffer $\boldsymbol{B}$. Clearly the core subvolume is related to the memtile subvolume and whether we can double buffer. If we must tile the K dimension, we can design the algorithm and estimate the execution time. Then we can restart the search space asking for a problem size that we can do DB of $\boldsymbol{B}$ and compare.

$\mathbb{Q}_0$ is more efficient for large $\boldsymbol{A}$, we tend to reuse $\boldsymbol{B}$, and at a minimum we need to double buffer $\boldsymbol{A}$. Clearly the core subvolume is related to the memtile subvolume and whether we can double buffer. If we must tile the K dimension, we can design the algorithm and estimate the execution time. Then we can restart the search space asking for a problem size that we can do DB of $\boldsymbol{A}$ and compare.

This search is almost complete. This is simply a set of heuristics for the selection of the solution space, the algorithms, and the tiling. In the, following section we shall present the ideas for a automatic selection and exploration of the solution space and its optimal solution.

V. Solution space and its exploration

We are working on a software tool that will explore a solution space for a given problem. For GEMM this is triplet (M,N,K) (i.e., $\boldsymbol{C} \in M \times N, \boldsymbol{A} \in M \times K, \boldsymbol{B} \in K \times N$). Briefly, tiling (M,N,K) to memtile is another triplet (M_m, N_m, K_m). This can describe the (largest) problem we solve using data in memtile. We can go down further and describe the tiling at core level as (M_c, N_c, K_c). In practice, $(M,N,K) = P$, $(M_m, N_m, K_m) = m$, and $(M_c, N_c, K_c) = c$ are enough to describe a complete computation.

The solution space is the composition of all valid $\mathcal{S} = \{(P,m,c) \text{ s.t. valid}\}$. A tiling tools have two main goals: First, we need to describe a complete set of solutions. In general, this can be quite large; however hardware requirements and designs strategies make the solution space a limited one. Second, we can address cost functions and their effect on the overall solution space simply by translating the solution P,m,c either into actual code or other realistic and consistent measures.

In all previous work, we did compute algorithm choices (such as tiling) by back of the envelop cost estimates, code generation cost estimate, simulations, and actual time measures. Here, we reduce to a minimum the description of the algorithm and hardware. We compute tiling for the application of double buffering at core and memtile level. We aim at symmetric computations (so that we can have uniform and predictable data movement).

In these subsections, we introduce the main idea for tiling and solution space we use for each computation. We assume in all cases that the computation start from DDR and we end in DDR. Problem P will start in DDR and the result will be in DDR. The operand memory allocation in DDR is given and the memtile is considered clean and clear. If the P fit memtile, then $(P, P, *)$ may be a valid solution. For this tool, we do not use addresses and we assume that allocation is always possible if space is available. We are aware this last statement is not always true.

A. GEMM

The current architecture is 4×2 architecture and thus the computation is rectangular by nature. We have two main algorithms Equation 15 and 17. We search first valid core solutions $(M_c, N_c, K_c) = c$ using property of alignment specific to the HW, then to the space requirements for the operands (with double buffering).

For Equation 15, we need to read from memtile $(2 * M_c, 4 * N_c, K) = m$ and for 17 we need $(4 * M_c, 2 * N_c, K) = m$. If K is too large, we will split K accordingly ($(4 * M_c, 2 * N_c, K/4) = m$, say) and stream the computation.

First, the data in memtile is split perfectly across the cores. We aim at padding at core level and if we can avoid better. If we can double buffer A or B or both we try, if we split by K, we double buffer portions of A and B on the K dimension. Second we can split the computation between DDR and memtile m and thus we have the final solution P, m, c. This describes completely our solution space from core to original problem size. We use a cost function, time, space, compute and ratio to explore the solution space and the Pareto curve.

We provide two different classes and approaches: we provide a solution space stemming from a single problem and from a set. A single problem create a list of solutions but we do not take advantage of any repetition. For a list of problem we create dictionary of solutions. This can help us to explore solutions that would not be available and consider in case extra constraints are introduced.

B. MHA

To exploit performance is to read into cores the matrices $\boldsymbol{K}$ and $\boldsymbol{V}$ so that to perform the largest block inner product $\sum_j \exp(\boldsymbol{Q}_0 \boldsymbol{K}_j) * \boldsymbol{V}_j$. The idea is to compute the space requirement in order to compute locally in a core the numerator Equation 10 and the denominator Equation 11. Then do a reduction by a single core for the final computation Equation 13.

The minimum number of cores is one, if it is possible we rather use a single column. The reduction is simpler done by a single column, and the next column can work on $\boldsymbol{Q}_1$ and $\boldsymbol{R}_1$.

We start with a problem of size $P = (d, L_0, L_1, r)$ so that $\boldsymbol{Q} \in L_0 \times d$, $\boldsymbol{K} \in d \times L_1$, and $\boldsymbol{V} \in L_1 \times r$. We find all the problem $c = (d, l_c, \ell_c, r)$ that fit in a core and we can estimate the execution time.

C. Layer Norm

The solution space and cost estimation follow the same ideas developed for MHA in the previous sections.

D. Convolution

The way we build the solution space is by observing the output tensor from the smallest to the largest and compute the projection to the input. Projection is a function that given an output tensor of size $(1, H, W, C_{out})$ and a convolution with stride $(0, s_h, s_w, 0)$ and kernel $(C_{out}, k_h, k_w, C_{in})$, the projection is $(1, (H-1)s_h + k_h, (W-1)s_w + k_w, C_{in})$. We have an starting problem $\mathbf{P}$.

Consider now that the output tensor will be created by two (or more) columns of cores. We consider the computation split by width so that each column compute may compute any $v \in [0, w/2]$ and each core compute $c_o \in [0, C_{out}/4]$ with a proper granularity (alignment). We assume we are streaming the computation by H (this is important for the computation description and its time estimates)

The core will use a space for $\mathbf{c}$

- output $(1, h, v, c_o)$
- input $(1, (h-1)s_h * k_h, (v-1)s_w + k + w, C_{in})$
- weights (c, k_h, k_w, C_{in}) + $(1, 1, 1, c_o)$

For each core subvolume, we consider in memtile only multiple of the core subvolume $\mathbf{m}$; that is, $H = m * h$ and $V = n * v$.

- output $(1, H, V, c_o)$
- input $(1, (H-1)s_h * k_h, (H-1)s_w + k + w, C_{in})$
- weights (c, k_h, k_w, C_{in}) + $(1, 1, 1, c_o)$

So for the same $\mathbf{c}$ we may have multiple $\mathbf{m}$ that can fit in memtile with double buffering.

We have then a complete representation of the solution by our usual triplet $(\mathbf{P}, \mathbf{m}, \mathbf{c})$ and we can estimate the performance by a simple cost function.

Notice, If we want to split the computation by input channel, C_{in}, we may just transform the convolution into two or more convolutions followed by element wise additions. If we want to split the computation by C_{out}, we can transform the convolution into two convolutions and a concatenation.