# Norms: A Unified Approach

PAOLO D'ALBERTO, AMD

Layer Norms are important tools in today's AI systems. Also, they are the little pesky layers that inference-engine tools have often problems in either fusing them with other layers for speed-up or because they introduce numerical unwanted oddities at low precision. Norms have linear complexity. That is, they are communication bound, they require to read the input at least twice, and there is a satisfying and contrasting asymmetry of the computation (i.e., row Gaussian normalization and column scaling).

Here, we unify the norm formalism. We want to clarify the general computation, the opportunities for fusion, and numerical properties. So we can generate consistent and correct code for our AIE AMD computes.

**ACM Reference Format:**
Paolo D'Alberto. 2025. Norms: A Unified Approach. 1, 1 (May 2025), 12 pages.

## 1 DEFINITIONS AND COMPUTATIONS

A matrix is $\boldsymbol{X} \in \mathbb{R}^{M\times N}$ with $M$ rows and $N$ columns and we represent a single scalar element as $x_{i,j}$ with obvious bounds. A vectors is $\boldsymbol{V} \in \mathbb{R}^{M}$ with scalar element $v_i$. Last, a scalar is $\alpha \in \mathbb{R}$.

A matrix norm is a two-part computation. We have a *projection* where we compute a factor, a vector, and then a *normalization* where we take the projected factor and distribute it back to the elements of the matrix. In the literature and implementations, these computations have also other names: Projection is a reduction because it is frequently associated with sums, dot products. Normalization is a broadcast. Now that we have a names, let us define them formally and constructively.

Let us start with scalar operations on Matrix/Vectors:

Definition 1. *A scalar operation on a matrix $\boldsymbol{X}$*

$$G_f : \boldsymbol{X} \longrightarrow \boldsymbol{Y} \text{ and } y_{i,j} = f(x_{i,j}) \tag{1}$$

For example, we will use $G_{x*x}(\boldsymbol{X}) = x^2_{i,j}$ and $G_{\exp}(\boldsymbol{X}) = \exp(x_{i,j})$.

Definition 2. *An associative Projection $P_+ : \boldsymbol{X} \in \mathbb{R}^{M\times N} \longrightarrow \boldsymbol{F} \in \mathbb{R}^{M}$, is a function where the element*

$$f_i = \sum_{j=1}^{N} x_{i,j}.$$

*The $+/\sum$ sign is not in general a summation, We choose it because this is a natural symbol for the associative property:*

$$f_i = (x_{i,1}) + (\sum_{j=2}^{N} x_{i,j}) = (\sum_{j=1}^{N-1} x_{i,j}) + (x_{i,N}).$$

Author's address: Paolo D'Alberto, , AMD, 2100 Logic Dr, San Jose, California, 95124.


XXXX-XXXX/2025/5-ART $15.00
https://doi.org/

The definition is a row projection. There are norms where the projection is by column. The formalism can easily take care of that by specifying a direction (i.e., in the code we introduce it). Let us digest our formalism a little. As we will read our matrix, we are going to maintain the order and projecting of matrix $\boldsymbol{X}$ (row elements $x_{i,*}$). We do not need the projection to be commutative in general but all the projections of interests are. Also, we require the computation of multiple projections for the computation of the normalization factor.

We abuse the notation of the symbol + in the associative expression $f_i = (x_{i,1} + (\sum_{j=2}^{N} x_{i,j}))$, in the sense that the combination of the temporary sub-projection may require some extra work during the computation or at the end when we have a complete projection into a proper normalization factor. However, such simplification does not detract any information and the implementation can address it.

What could $P_+(G_f(X))$ be? That is, we apply scalar operations and then projection. The associativity should be a property of the projection and not of the value of the matrix. SoftMax Norm in finite precision is an example where we have *to work* to enforce the associativity (e.g., $\exp(x)$ can be large and in finite precision overflow easily especially their sum).

Definition 3. *An optional summary function* $S_+ : \{\boldsymbol{F}\}_i \in \mathbb{R}^M \rightarrow \boldsymbol{E} \in \mathbb{R}^M$ *and* $\boldsymbol{F}_i = P_i(G_{f_i}(\boldsymbol{X}))$

So we can summarize the first part of the computation by an associative projection or a set of projections and an optional summary function for the final composition of the normalization factors.

Now let start formulating the normalization computation.

Definition 4. *We have two normalization functions by row*

$$N_* : \boldsymbol{X} \in \mathbb{R}^{M\times N} \times \boldsymbol{T} \in \mathbb{R}^M \rightarrow \boldsymbol{Y} \in \mathbb{R}^{M\times N}, \text{ where } y_{i,j} = x_{i,j} * t_i \tag{2}$$

*and by column*

$$N_* : \boldsymbol{X} \in \mathbb{R}^{M\times N} \times \boldsymbol{S} \in \mathbb{R}^N \rightarrow \boldsymbol{Y} \in \mathbb{R}^{M\times N}, \text{ where } y_{i,j} = x_{i,j} * s_j \tag{3}$$

*Again the $*$ is not necessarily a multiplication and the normalization can be by row or by column. For presentation purpose we will use a single symbol (unless confusing).*

If we extend the normalizing factor $\boldsymbol{F}$ into a diagonal matrix in $\boldsymbol{D}_F \in \mathbb{R}^{M\times M}$ so that $[\boldsymbol{D}_F]_{i,i} = f_i$ and zero everywhere else (same idea for $\mathbb{R}^{N\times N}$), and the normalization is a based on a scalar product, then we reduce the computation to matrix-matrix multiplication products.

$$\boldsymbol{D}_F * \boldsymbol{X} \text{ and } \boldsymbol{X} * \boldsymbol{D}_F \tag{4}$$

Notice that using matrix multiplication we can see that a row normalization is associative $(\boldsymbol{D}_f * \boldsymbol{X}) * \boldsymbol{Y} = \boldsymbol{D}_f * (\boldsymbol{X} * \boldsymbol{Y})$ then giving an hint how the normalization could be postponed if neceesary. Also a column mormalization in combination with a matrix multiplication $(\boldsymbol{X} * \boldsymbol{D}_f) * \boldsymbol{Y} = \boldsymbol{X} * (\boldsymbol{D}_f * \boldsymbol{Y})$ is actually a row normalization of the following operand.

We introduce the last definiition, a matrix partition.

Definition 5. *A row partition of a matrix is simply defined as*

$$[\boldsymbol{X}]_{r=2} = \begin{pmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \end{pmatrix} \tag{5}$$

*and a column partition*

$$[\boldsymbol{X}]_{c=2} = \begin{pmatrix} \boldsymbol{X}_1 & \boldsymbol{X}_2 \end{pmatrix} \tag{6}$$

If we want to take a matrix $\boldsymbol{X} \in \mathbb{R}^{M \times N}$ and normalize its rows by a square norm $||x||_2$, see Algorithm 1. As we define $P_+(\boldsymbol{X})_i$ as the summation $\sum_j x_{i,j}$ and $N_*(\boldsymbol{X}, \boldsymbol{T})_{i,j} = x_{i,j} * t_i$, we have the complete computation of the norm : $f_i = \sqrt{\frac{1}{N} \sum_{j=1}^{N} x_{i,j}^2}$ and $x_{i,j} = \frac{x_{i,j}}{f_i}$.

**Algorithm 1** $\boldsymbol{X}/||\boldsymbol{X}||_2$ algorithm

$\boldsymbol{T} = P_+(G_{x*x}(\boldsymbol{X}))$
$---$
$\boldsymbol{T} = \sqrt{\boldsymbol{T}/N}$ { # Summary}
$---$
$\boldsymbol{X} = N_*(\boldsymbol{X}, \boldsymbol{T})$

When we introduce a new projections $P_{max}(\boldsymbol{X})_i = \max x_{i,*}$, which is associative, and we normalize the additive projection with the result of the new projection, we can see that the Softmax computation has the same properties and composition (Algorithm 2).

**Algorithm 2** $SoftMax(\boldsymbol{X})$ algorithm

$\boldsymbol{M} = P_{max}(\boldsymbol{X})$
$\boldsymbol{T} = P_+(G_{exp}(\boldsymbol{X}) * G_{exp}(-\boldsymbol{M}))$
$---$
$\boldsymbol{T} = \boldsymbol{T} * \exp \boldsymbol{M}$ { # Summary}
$---$
$\boldsymbol{X} = N_*(G_{exp}(\boldsymbol{X}), \boldsymbol{T})$

Let us go back using the original notations and block the input matrix so that to exploit its natural composition, independent computations by row and we exploit the associativity of the projection in Algorithm 3

**Algorithm 3** Blocked $3 \times 3$

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{1,1} & \boldsymbol{X}_{1,2} & \boldsymbol{X}_{1,3} \\ \boldsymbol{X}_{2,1} & \boldsymbol{X}_{2,2} & \boldsymbol{X}_{2,3} \\ \boldsymbol{X}_{3,1} & \boldsymbol{X}_{3,2} & \boldsymbol{X}_{3,3} \end{pmatrix}$$

**while** $i$ in [1,2,3] # Independent **do**
  $\boldsymbol{T} = \boldsymbol{0}$ { # Zero element for $P_+$}
  **while** $j$ in [1, 2,3] # Projection **do**
    $\boldsymbol{T} = \boldsymbol{T} + P_+(G_f(\boldsymbol{X}_{i,j}))$ { # Reduction}
  **end while**
  $---$
  $\boldsymbol{T} = S_+(\boldsymbol{T})$ { # Summary}
  $---$
  **while** $j$ in [1,2,3] # Normalization **do**
    $\boldsymbol{X}_{i,j} = N_*(\boldsymbol{X}_{i,j}, \boldsymbol{T})$
  **end while**
**end while**

The application of the blocked algorithm for the Euclidean Norm and for Softmax requires very little re-work, just remember that softmax will require two projections:

| SoftMax associative projections |
|---|
| $\boldsymbol{M}_0 = -\infty; \boldsymbol{T}_0 = 0$ |
| $\boldsymbol{M}_{j+1} = max(M_j, P_{max}(\boldsymbol{X}_{i,j})$ |
| $\boldsymbol{T}_{j+1} = \boldsymbol{T}_j * (exp(-(\boldsymbol{M}_{j+1} - \boldsymbol{M}_j))) + P_+(G_{\exp}(\boldsymbol{X}_{i,j} - \boldsymbol{M}_{j+1}))$ |

Once we recognizing the particular projection, the matching of the proper computation should be natural. Also, the layer norm $(x_{i,j} - \mu_i)/\sqrt{\sigma_i^2}$ will require the computation of two projections so that average ($\mu$) and variance ($\sigma^2$) can be computed in a single pass but with two projections (Algorithm 4).

**Algorithm 4** Layer Norm $3 \times 3$ with $\gamma, \beta$

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{1,1} & \boldsymbol{X}_{1,2} & \boldsymbol{X}_{1,3} \\ \boldsymbol{X}_{2,1} & \boldsymbol{X}_{2,2} & \boldsymbol{X}_{2,3} \\ \boldsymbol{X}_{3,1} & \boldsymbol{X}_{3,2} & \boldsymbol{X}_{3,3} \end{pmatrix}$$

$$\boldsymbol{\gamma} = \begin{pmatrix} \boldsymbol{\gamma}_1 & \boldsymbol{\gamma}_2 & \boldsymbol{\gamma}_3 \end{pmatrix}$$

$$\boldsymbol{\beta} = \begin{pmatrix} \boldsymbol{\beta}_1 & \boldsymbol{\beta}_2 & \boldsymbol{\beta}_3 \end{pmatrix}$$

**while** $i$ in [1,2,3] **do**
  $\boldsymbol{T} = \boldsymbol{0}$
  $\boldsymbol{S} = \boldsymbol{0}$
  **while** $j$ in [1, 2,3] **do**
    $\boldsymbol{T} = \boldsymbol{T} + P_+(\boldsymbol{X}_{i,j})$ {Projection + sum}
    $\boldsymbol{S} = \boldsymbol{T} + P_+(G_{x*x}(\boldsymbol{X}_{i,j}))$ {Projection + sum squares}
  **end while**
  ——————————
  $\boldsymbol{T} = \boldsymbol{T}/N$ {Summary, scalar}
  $\boldsymbol{S} = (\boldsymbol{S} - (\frac{\boldsymbol{T}}{\sqrt{N}})^2)/N$ {Summary, scalar}
  ——————————
  **while** $j$ in [1,2,3] **do**
    $\boldsymbol{X}_{i,j} = \boldsymbol{X}_{i,j} - \boldsymbol{T}$ {Normalization Row}
    $\boldsymbol{X}_{i,j} = \boldsymbol{X}_{i,j}/\boldsymbol{S}$
    $\boldsymbol{X}_{i,j} = \boldsymbol{X}_{i,j} * \gamma_j$ {Normalization Column}
    $\boldsymbol{X}_{i,j} = \boldsymbol{X}_{i,j} + \beta_j$
  **end while**
**end while**

## 2 COMPUTATION MODES

The mathematical notation provides a clear description of the computation, clearly pedantic but constructive in nature. Also, the normalization factor and its computation can be used when we would like to fuse the norm operation with the following (or the previous) computation.

Here, we are more interested in describing how we would implement the norm computation using different engine: GPU, GPU, and AIE FGPA engines. We show direct implementations for

CPU and AIE and we present a short discussion about the GPUs. This is always related to AMD chips and codes.

The main difference is about the computation of the Projection and how the hardware implicitly helps the computation. Let's start with the GPU that uses heavily the associativity for parallelization of the computation.

### 2.1 GPU conversation

The projection is associative and we can exploit the parallelism by blocking the computation in warps, compute independent and separated $\boldsymbol{T}_i = P_+(\boldsymbol{X}_{i,j})$, then reduce the computation in cores by using a tree like computation and built in HW utilities (register computations).

$$\begin{array}{lll} \boldsymbol{T}_1 = P_+(\boldsymbol{X}_{i,1}), & \ldots & \boldsymbol{T}_n = P_+(\boldsymbol{X}_{i,n}) \\ \boldsymbol{T}_1 = \boldsymbol{T}_1 + \boldsymbol{T}_2, & \ldots & \boldsymbol{T}_{\frac{n}{2}} = \boldsymbol{T}_{\frac{n}{2}} + \boldsymbol{T}_{\frac{n}{2}+1} \\ \boldsymbol{T}_1 = \boldsymbol{T}_1 + \boldsymbol{T}_2, & \ldots & \boldsymbol{T}_{\frac{n}{4}} = \boldsymbol{T}_{\frac{n}{4}} + \boldsymbol{T}_{\frac{n}{4}+1} \\ \ldots & \ldots & \ldots \\ \boldsymbol{T}_1 = \boldsymbol{T}_1 + \boldsymbol{T}_2 & & \end{array} \tag{7}$$

The small size of the computation that can be done as unit and the HW support for the reduction makes this computation appealing. Numerically, the binary three computation makes the computation balanced and well partitioned. If there is no use of high precision accumulators this will have numerical advantages. That is, a balanced binary tree with $N$ elements has the longest addition path length of only $K = \log_2(N)$ and the fish spine tree has $K = N - 1$ and for additions we accumulate an error of $O(K)$.

For a Layer Norm of size $512 \times 768$ on a MI100 with 1.2 TB/s HBM2 we can achieve 9 us (microseconds) latency (bandwidth utilization 800GBs) for layer norm in fp16 and fp32 precision. This is the fastest performance across the AMD products at our disposal.

### 2.2 CPU conversation

CPUs have deep memory hierachies to exploit temporal locality. If we observe the norm computations there is little temporal locality. Each row is read twice, but there will be $N$ accesses in between. As soon as we realize that $\boldsymbol{T}+ = P_+(\boldsymbol{X}_{i,j})$ has consecutive accecesses to fast L1 caches we can exploit the AVX instruction set to transform scalar operations into parallel vector operations.

The partition on $\boldsymbol{X}$ will associate a set of row to a thread/core. We stream the input rows and we store in the core the temporary $T$ (registers), we compute the factor(s), then we stream the rows again for the propagation of the normalization factor(s). We exploit temporal locality because the smallest and fastest cache in the memory hierarchy containing the rows will be read twice naturally without any further assistance. When the computation moves to the next rows, we clear the memory hierarchy safely. C threads can deploy this strategy naturally and without the need to much careful blocking. However, threading is heavy and they are really applicable only for very large matrices.

```
Vendor ID:               AuthenticAMD
  Model name:            AMD EPYC 7F52 16-Core Processor
Caches (sum of all):
  L1d:                   1 MiB (32 instances)
  L1i:                   1 MiB (32 instances)
  L2:                    16 MiB (32 instances)
  L3:                    512 MiB (32 instances)
```

By writing naive code for x86 ISA and using as reference the $512 \times 768$, the base code reference is about 600 us. By exploiting a different style in exploiting parallel row computations and using the AVX instructions (using *–fast-math*), then we can achieve 121 us using a single core and float point 32bits.

For comparison to an older threadripper with share L3 cache

```
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen Threadripper 1950X 16-Core Processor
Caches (sum of all):
  L1d:                   512 KiB (16 instances)
  L1i:                   1 MiB (16 instances)
  L2:                    8 MiB (16 instances)
  L3:                    32 MiB (4 instances)
```

we can achieve 269 us.

## 3 AIE CORE COMPUTATION, TILING GENERATION, AND VALIDATION

As matter of fact, we use float 64bit precision in the code generation presented in this repository. However, the AIE instruction set has vector operations per core (like GPU), for int8, int16, bfloat16 and float32. The AIE system has a three level memory hierarchy: L3 connected to four L2 (memtile one for each column) and each L2 is connected with one column of four cores L1 and one specific row with four core L1.

The reduction using GPU style codes is possible and encouraged for each core internally but there is no simple equivalent for extra-core reduction. We suggest a Norm computation on AIE similar to the CPU in the sense that we use the memory hierarchy to stream row partitions and accumulate in core the projections. Then, we do another pass. The main difference that the data movement has to be orchestrated explicitly because there is not hardware assisted data replacement policy.

Let us rewrite the blocked algorithm here once again to show the logical division of the computation but we want to draw attention to the same inner loop reading the input. A reminder the temporary projections, summary, and normalizing factors will be store in core (L1). We will have a section related to the case when and where we must spill (to L2).

**Algorithm 5** Blocked $3 \times 3$ (Projection in L1)

$$X = \begin{pmatrix} \boldsymbol{X}_{1,1} & \boldsymbol{X}_{1,2} & \boldsymbol{X}_{1,3} \\ \boldsymbol{X}_{2,1} & \boldsymbol{X}_{2,2} & \boldsymbol{X}_{2,3} \\ \boldsymbol{X}_{3,1} & \boldsymbol{X}_{3,2} & \boldsymbol{X}_{3,3} \end{pmatrix}$$

**while** $i$ in [1,2,3] # Independent **do**
  $\boldsymbol{T} = \mathbf{0}$ { # Zero element for $P_+$}
  **while** $j$ in [1,2,3] # Projection **do**
    $\boldsymbol{T} = \boldsymbol{T} + P_+(G_f(\boldsymbol{X}_{i,j}))$ { # Reduction}
  **end while**
  ——————————
  $\boldsymbol{T} = S_+(\boldsymbol{T})$ { # Summary}
  ——————————
  **while** $j$ in [1,2,3] # Normalization **do**
    $\boldsymbol{X}_{i,j} = N_*(\boldsymbol{X}_{i,j}, \boldsymbol{T})$
  **end while**
**end while**

### 3.1 Core computation

Take the prospective of a core $C_i$ that, somehow, can access from a stream of data a block $\boldsymbol{X}_{i,k}$. In a CPU implementation, the core starts the reading of the block by a load and the HW may go as far the OS page to procure the data (page, DRAM Row, L3, L2, L1d). By the AIE programming model, imagine we create a stream where $\boldsymbol{X}_{i,1}$ ... $\boldsymbol{X}_{i,N}$ is send through and it is repeated twice, we call the location where data arrives $L_1$. The core just need to wait for the block, release the block

when done, and wait for the next. Now by literally counting to $N$, the core knows when to do the projection, the summary, and the normalization. The Algorithm 6 provides a clear boundary of the core computation as a kernel and how the functionality is based on counting the number of partitions read and when to write.

**Algorithm 6** Blocked $3 \times 3$

```
while i in [1,2,3] # Independent do
  while r in [1,2] followed j in [1, 2, 3] do
    { # Core Computation}
    if r * j == 1 then
      T = 0 { # Zero element for P+}
    else if r == 1 then
      T = T + P+(Gf(X_{i,j}))
    else if r == 2 and j == 1 then
      T = S+(T)
    else
      X_{i,j} = N*(X_{i,j}, T)
    end if
  end while
end while
```

Using a common terminology for the AIE programming: $r$ is the repetition, $\boldsymbol{X}$ is the buffer, $\boldsymbol{X}_{i,k}$ is the tile, and $\boldsymbol{X}_{i,1}$ ... $\boldsymbol{X}_{i,N}$ is the (time) traversal of the buffer by tiles with the specified repetition. In this particular norm, Projection and Reduction require the same memory foot print because they both read $\boldsymbol{X}_{i,N}$ and only one write $\boldsymbol{X}_{i,N}$. If we introduce column normalization such as in Algorithm 4, the normalization pass will require to read $\boldsymbol{X}_{i,k}$, $\gamma_k$, and $\beta_k$. We will address space constraints and tiling in the next section.

### 3.2 Tiling

Let us start with our interpretation of tiling by using an abstract representation of the memory hierarchy and connections as in Figure 1: L3 = DDR = infinity, L2 is composed of four mem-tiles of 512KB, and each mem-tile has a connection to one column of cores and one different connection to a row. Here, we assume that the connections are broadcasts: the same data is broadcast to all cores and the core will select a subpart (if it likes). Each core has eight banks of 8KB each composing the lowest level L1, we give two banks for inputs for ping pong, two for weights, two for outputs, and two banks for Temporary Space. We use this information in the following Tiling generation section **??**

Intuitively, Tiling is a *spatial* and a *temporal* partition. Where, a partition describes a way to take a matrix and split into (non-overlapping) parts in order to cover the whole original matrix and in principle move it. We assume a row-major layout.

Definition 6. *Consider a matrix* $\boldsymbol{X} \in \mathbb{R}^{M \times N}$,

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}^1 \\ \boldsymbol{X}^2 \\ \boldsymbol{X}^3 \\ \boldsymbol{X}^4 \end{pmatrix} = \dot{\sum_i^r} \boldsymbol{X}^i \tag{8}$$

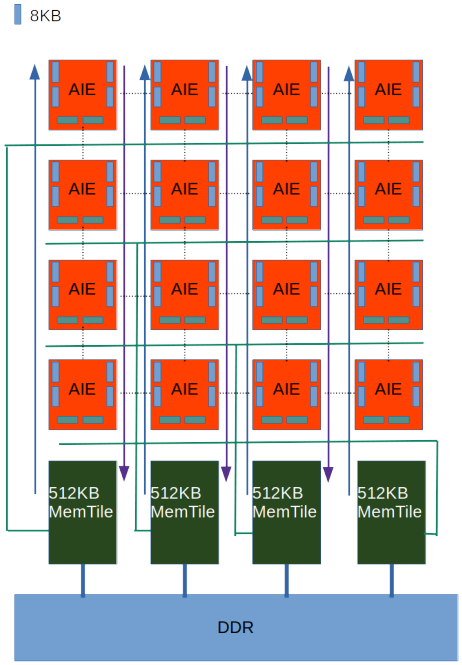


Fig. 1. 4x4 AIE representation

*a row partition is easily specified by an offset address Address($X_i$) = $(i-1)\frac{M}{4}N$ and the tiles are in contiguous memory addresses. In similar fashion, in a column partition*

$$X = \begin{pmatrix} X^1 & X^2 & X^3 & X^4 \end{pmatrix} = \dot{\sum_i^c} X^i \tag{9}$$

*the offset addresses are Address($X_i$) = $(i-1)\frac{N}{4}$ and the tiles are stored in a strided spaces. These represent* **spatial partitions**, *each partition has a buffer $X^i$ and each is independent.*

Definition 7. *Consider a buffer $X^k$ and partition it into tiles $X_{i,j}$ and the* **temporal partition** $X^k = \sum_{i,j}^{rc} X_{i,j}^k$ *(by row and then columns). Think of a stream where $X_{i,j}^k$ is streamed in order so that the whole buffer eventually has been touched/transfer. There is $\sum_i^r X_i^k$ by row and $\sum_i^c X_i^k$ by column as a temporal partition.*

Definition 8. *The L3 to L2 traversal of a matrix $X$ is the composition of a spatial and temporal partition: For example, $T(X) = \dot{\sum_k^r} X^k = \dot{\sum_k^r} \sum_i^r X_i^k$. $X$ is the original buffer, there are four spatial tiles $X^k$, each is the buffer that will be streamed by temporal partition $\sum_i^r X_i^k$ with tile $X_i^k$ (by row, by column, and both)*

*3.2.1 Tiling LLM.* A L3 to L2 traversal is valid if the tile size fits L2 (without context and without double buffering). More constraints are introduced when more information are available. For Norms we need double buffering and we stream at very least $X$ and the output of the Norm $Y$. In general, a complete tiling/traversal for L3, L2, and L1 could be summarized using our notation by loops: spatial by row, temporal by row, spatialby row, temporal by row (all independent computations).

$$T(X) = \dot{\sum_k^r} X^k = \dot{\sum_k^r} \sum_i^r X_i^k = \dot{\sum_k^r} \sum_i^r \dot{\sum_l^r} X_i^{k,l} = \dot{\sum_k^r} \sum_i^r \dot{\sum_l^r} \sum_m^r X_{i,m}^{k,l}. \tag{10}$$

Note, the second spatial partition is a way to take the matrix in L2, split it so that it will feed each core L1 separately, but because we use broadcast, there is not second spatial partition and all core share the same data:

$$\dot{\sum_k^r} \sum_i^r \sum_m^r \boldsymbol{X}_{i,m}^k. \tag{11}$$

- If we observe just the tiling and the first partition using split by column, this split marks the beginning of temporal reuse, the computation will come back for the normalization and this split will be *repeated* twice. This computation will require blocked reduction.
- Using our notation, the following tiling formulations imply formal reuse in L1 (we read L1 twice)

$$\dot{\sum_k^r} \sum_i^r \sum_m^r \boldsymbol{X}_{i,m}^k, \quad \dot{\sum_k^r} \sum_i^r \sum_m^C \boldsymbol{X}_{i,m}^k, \quad \dot{\sum_k^r} \sum_i^r \sum_m^{RC} \boldsymbol{X}_{i,m}^k. \tag{12}$$

- These imply reuse in L2

$$\dot{\sum_k^r} \sum_i^C \sum_m^c \boldsymbol{X}_{i,m}^k, \quad \dot{\sum_k^r} \sum_i^{RC} \sum_m^c \boldsymbol{X}_{i,m}^k. \tag{13}$$

- This implies reuse in L3

$$\dot{\sum_k^C} \sum_i^c \sum_m^c \boldsymbol{X}_{i,m}^k. \tag{14}$$

- This implies two separate passes and spill to L2/L3 of $P_+(\boldsymbol{X})$ for which. The first split is a temporal split repeating the overall process twice (this is not possible in MLADF but we can split the main computation into smaller ones).

$$\sum_\ell^r \dot{\sum_k^C} \sum_i^c \sum_m^c \boldsymbol{X}_{\ell,i,m}^k. \tag{15}$$

The AIE traversal implementation is more like the notation above. That is, the inner loops describe only one sub partition simplifying the implementation and enforcing a partition that has same shapes. Also some repetition are not possible

However, the logical partitioning of any matrix is a tool completely general and with some experience, is applicable for different levels and connections. We use it for the representation of bilinear matrix multiplication algorithms in the past.

In practice, we implement Tiling as recursive function, and the data structure is like a tree once built and unfolded.

$$T[L3 \to L2](X) = \dot{\sum_k^r} \sum_i^r T[L2 \to L1](\boldsymbol{X}_i^k). \tag{16}$$

*3.2.2 Tiling Unified.* The unified overlay use the connection between DDR and L2 memtiles in a different pattern.

We still consider DDR infinite. From the DDR, First, the inputs and the outputs will be able to access a logical L2 memory (IOL2) composed of up to 3 memtiles, thus 512*3*KB and memtile M1, M2, and M3. Second the weights will access a separate and single memtile M4 (WL2). The logical IOL2 has 4 vertical connections one for each column to each core. Each column communication is a broadcast. The WL2 has 4 row connection, and each row connection is a broadcast. Each core in each column produces a separate output and it has is custom channel back to IOL2.

The tiling in this architecture follows this rule: temporal, spatial, temporal.

$$T(\boldsymbol{X}) = \sum_k^r \boldsymbol{X}_k = \sum_k^r \dot{\sum_i^r} \boldsymbol{X}_k^i = \sum_k^r \dot{\sum_i^r} \sum_l^r \boldsymbol{X}_{k,l}^i = \tag{17}$$

### 3.3 Tiling generation

There are always computation and hardware constraints that enforce granularity in the shape and size of the tiles. At this time we do not aim to an optimal tiling but we explore and present constraints-based heuristics. Two main ideas are applied: a top down partition and double buffering is preferred. We have a fit function, see Algorithm 7.

**Algorithm 7** Fit : Tiling, L, SplitFunction, multiplicity, granularity

```
Components = Tiling.spatial-parts # list of matrices
while  c in Components do
   T = None; t = 1; q = True
   while  q is True do
      if t*gran is too large then
         q = False; continue
      else if t*gran has reminder  then
         continue
      end if
      t = SplitFunction(c,t*gran)
      A = t[0] { # Assume the first partition is the largest}
      if multiplicity*A.space()>L then
         q = False
      else
         T = A
      end if
      t+=1
   end while
   c = Tiling(c, T)
end while
```

We choose the largest partition for which the tiles size fit the level capacity L with or without double buffering (multiplicity=1 or 2). Making sure that the size of the tile has a proper granularity is to make sure that the following tiles may have the same granularity satisfied without awkward reminder making the index computation and problem size not properly balanced.

In general, we start by trying splitting the computation by row $\dot{\sum}_k^r \sum_i^r T(\boldsymbol{X}_i^k)$. We fit recursively the Tiling. If it does not fit, then we try by column $\dot{\sum}_k^c \sum_i^c T(\boldsymbol{X}_i^k)$ and we know that the rest will be by column.

What is the implication of the latter case, when we start splitting temporally by columns? We split the matrix in space by row in L3 into 4 parts. Each spatial partition will transfer at least a complete column, this will be transfer to L2 as it is, and to L1 (broad cast). L1 is 8 KB of data, if we assume that each element is two Bytes, L1 and double buffering can hold 4*K elements, if we have a single projection, we are good to go and we can handle a $M = 16K$ elements. For layer norm, we need to keep two projections and thus the maximum size is 2*K read elements and thus ($M = 8K$). For even larger, we need to split the computation into several Layer Norms. Other oddity to consider for the space constraints is the addressable space, often we need to read at least 4 Bytes (if not 32B) so in L1 we need to read at least two columns (reducing $M$).

### 3.4 Tiling: *gamma* and *beta*

Notice in Algorithm 4 and 6, the input is $\boldsymbol{X}_{i,j}$, it is read twice, it is written once and normalized $\boldsymbol{X}_{i,j} * \gamma_j + \beta_j$. For a broad cast communication $\boldsymbol{X}_{i,j}$ takes at least space $4 \times k$, we need $2 \times k$ for the

parameters $\gamma$ and $\beta$, and the partial results for the reduction (for these norms) is about 2. We can see that the tiling of the $\boldsymbol{X}$ really drives the tiling of the the weights. In fact, consider

$$\dot{\sum_k^x} \sum_i^y \sum_m^z \boldsymbol{X}_{i,m}^k. \tag{18}$$

then the column partition will be the same

$$\dot{\sum_k^x} \sum_i^y \sum_m^z \boldsymbol{\gamma}|\boldsymbol{\beta}_{i,m}^k. \tag{19}$$

### 3.5 Tiling: Computation Validation

For simplicity, assume we have a Tiling and remember that the temporary space for the partial projection computation will be actually done in core without spills.

- $\dot{\sum_k^r} \sum_i^r \sum_m^r \boldsymbol{X}_{i,m}^k$.

```
# ----
for k in K:
    for i in I do:
        for m in J:
            T= Projection(X[k,i,m])
            T = S(T)
            X[k,i,m] = X[k,i,m]/T
# ----
```

- $\dot{\sum_k^r} \sum_i^r \sum_m^c \boldsymbol{X}_{i,m}^k$.

```
# ----
for k in K:
    for i in I do:
        T = 0 # the first time we start the projection
        for m in J:
            T+= Projection(X[k,i,m])
        T = S(T) # the first time we start the normalization
        for m in J:
            X[k,i,m] = X[k,i,m]/T
# ----
```

- $\dot{\sum_k^r} \sum_i^c \sum_m^c \boldsymbol{X}_{i,m}^k$.

```
# ----
for k in K:
    T = 0# the first time we start the projection
    for i in I do:
        for m in J:
            T+= Projection(X[k,i,m])
    T = S(T)  # the first time we start the normalization
    for i in I do:
        for m in J:
            X[k,i,m] = X[k,i,m]/T
# ----
```

- $\dot{\sum_k^c} \sum_i^c \sum_m^c \boldsymbol{X}_{i,m}^k$.

```
# ----
T=0 # the first time we start the projection
for k in K:
    for i in I do:
        for m in J:
            T+= Projection(X[k,i,m])
T = S(T) # the first time we start the normalization
for k in K:
    for i in I do:
        for m in J:
            X[k,i,m] = X[k,i,m]/T
# ----
```

In practice, we formulate the computation as a recursive descent visit into the tiling data structure. Instance Norm has parallel computation by columns and other Norms are by row.

## 4 CONCLUSION

In this work, we show that Norms can be abstracted in such a way that mathematical and computation properties can be easily represented.

(1) We show that Tiling (for AIE) of every Norm can be generated by a single heuristics.
(2) The tiling is correct,
(3) We can perform actual computations to validate in practice the solution
(4) the code for this project shows that the analysis help in the writing of concise codes.