Attention in SRAM on Tenstorrent Grayskull

Moritz Thüning
Technical University of Munich
moritz.thuening@tum.de

***Abstract*—When implementations of the Transformer's self-attention layer utilize SRAM instead of DRAM, they can achieve significant speedups. The Tenstorrent Grayskull architecture provides a large SRAM, distributed across a grid of cores. This work presents a fused kernel for Grayskull, that exclusively utilizes its large SRAM by combining matrix multiplication, attention score scaling and Softmax operations. Additionally, a dedicated Softmax kernel utilizing the SRAM and a CPU implementation serving as a baseline are presented. The Softmax operation consumes most of the runtime in the computation of attention weights from queries and keys on Grayskull. The speedup of the dedicated Softmax kernel compared to the CPU implementation is up to $10\times$, and the Softmax implementation inside the fused kernel is approximately $1.8\times$ faster than the dedicated Softmax kernel. The time and memory complexity of all implementations is quadratic in sequence length. Currently, the Grayskull e150 is approximately $30\times$ cheaper for the general public than an Nvidia H100 PCIe (a state-of-the-art GPU) and offers approximately $1.5\times$ more SRAM.**

I. Introduction

The Transformer [22] has become the state-of-the-art architecture in many applications, particularly in natural language processing. However, it is based on the self-attention layer which has a time and memory complexity quadratic in sequence length.

To improve the complexity, approximate attention mechanisms have been proposed, but they are not efficient or accurate enough to be widely adopted [2], [3], [5], [9], [10], [14], [23], [25]. Another approach is to improve the memory bandwidth and latency without changing the quadratic complexity. The observation that the Softmax operation in self-attention is memory-bound on GPUs led to FlashAttention [7] which achieved significant speedups. It is an exact attention algorithm that uses tiling and in the backward pass recomputation to reduce the movement of attention scores between the GPU's high-bandwidth memory (HBM) and SRAM.

The recent increase in demand for AI applications motivated the design of new hardware architectures to accelerate highly parallel AI workloads. One of those is the Tenstorrent Grayskull architecture [15], [17], [18], [21], commercially available for example as a Grayskull e150 PCIe card. Currently, it is approximately $30\times$ cheaper than the Nvidia H100 PCIe card [6] (a state-of-the-art GPU). Nevertheless, the Grayskull e150 provides 120 MB SRAM (120 Tensix cores with 1 MB each) compared to only 80 MB of the Nvidia H100 PCIe (114 Streaming Multiprocessors with 256 KB L1 each and 50 MB L2 shared).

This background provided the motivation for this work, which includes the implementation of a **dedicated Softmax kernel** leveraging the large SRAM of the Grayskull e150, as well as a **fused kernel** combining matrix multiplication, attention score scaling and Softmax operations. The fused kernel reduces overhead of dispatching kernels and of moving inputs/outputs from separate kernels to and from DRAM.

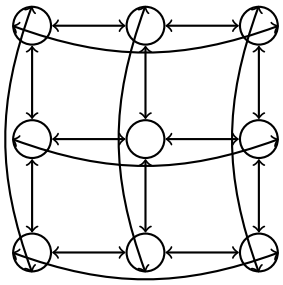

Fig. 1: Topology of the Network-on-Chip (NoC). Nodes represent Tensix cores and the edges represent bi-directional connections between them. The actual Tensix core grid of Grayskull is 10×12. It is a torus topology, since opposite ends are connected.

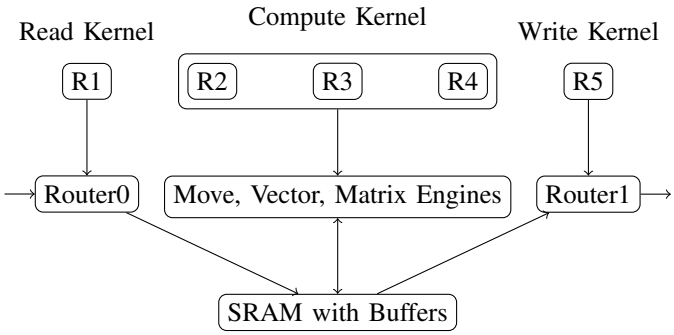

Fig. 2: Inside a Tensix core. R1 represents the first RISC-V core. The kernels run on RISC-V cores and the cores control other components. The routers are connected to the NoC and exchange data via Buffers in SRAM with the engines.

II. Tenstorrent Grayskull e150

The Tenstorrent Grayskull e150 card supports PCIe 4.0 x16 and is based on a dataflow architecture [8]. It consists of a 10×12 rectangular grid of Tensix cores (See Figure 2) connected by a Network-on-Chip [18] (NoC, See Figure 1). The cores in the last row cannot be used for computation but still provide memory. By the definition of the dataflow architecture, each Tensix core executes its individual instructions depending on

the flow of data through the grid (e.g., the availability of input data), but otherwise, it operates independently of other cores.

Tensix cores operate on tiles of 32×32 scalars in various data formats, including Bfloat16. This format has the same number of exponent bits as Float32, but the mantissa is 16 bits shorter. So it has the same range, but reduced precision. However, the precision is sufficient for many operations in machine learning and its use leads to reduced memory consumption and runtime. Therefore, all the following implementations for Grayskull utilize Bfloat16.

Each tile together with a header containing routing information and a unique id is sent as a packet via the NoC. The NoC is two-dimensional, bi-directional, and has a torus topology (opposite ends are connected). Each Tensix core is connected to the NoC with a bandwidth of 192 GB/s.

The Grayskull e150 operates at a maximum clock speed of 1.2 GHz and a maximum power of 200 W. The card has eight channels of LPDDR4 memory at the top and bottom edges of the grid with 8 GB total capacity and 118.4 GB/s bandwidth (compared to 80 GB HBM with 2 TB/s of an Nvidia H100 PCIe). It has a computational performance of 92 and 332 TFLOPs for 16- and 8-bit floats respectively (1513 and 3026 TFLOPs for Nvidia H100 PCIe). Keep in mind that the H100 PCIe is approximately $30\times$ more expensive for the general public. The chip is fabricated using a 12 nm process and the area of the core grid is $477\,\mathrm{mm}^2$.

A. Tensix core

A Tensix core has five RISC-V [24] cores, two routers, a data movement engine, a compute engine and 1 MB of SRAM. Since the SRAM has a bandwidth of 384 GB/s, 120 Tensix cores accessing their memory in parallel would have a bandwidth of approximately 46 TB/s (compared to approximately 6 TB/s L2 read bandwidth [11] of an Nvidia H100 PCIe). The routers are connected to the NoC and move tiles in and out of buffers in SRAM. The data movement engine includes a packer and unpacker for moving tiles between SRAM and the compute engine. The compute engine consists of a matrix engine (FPU, not floating-point unit) and vector engine (SFPU). The RISC-V cores dispatch instructions to the other components, which is also called "driving" those components.

Incoming tiles are processed in the following way. The first RISC-V core drives router0 to push the incoming tiles from the NoC to a circular buffer in SRAM. The second RISC-V core drives the packer to pop the tiles from the circular buffer to the compute engine. The third core drives the FPU and SFPU to compute mathematical operations. The fourth core drives the unpacker to push the tiles from the compute engine back to another circular buffer in SRAM. And finally, the fifth core drives router1 to pop tiles from the circular buffer back to the NoC.

Because of this highly parallel design, data movement and computation run in parallel which keeps the compute engine busy.

III. Tenstorrent Software

For programming Tenstorrent hardware there is the high-level, top-down software stack TT-Buda [19] and the low-level, bottom-up C++ software stack TT-Metalium [20].

A. TT-Buda: high-level, top-down

With TT-Buda, AI models can be defined in external frameworks (e.g., PyTorch [13], Tensorflow [1]) or directly in its Python interface PyBuda. Then, it compiles the model to a binary running on the hardware. The compiler is composed of a frontend and a backend.

The first step of the frontend is compiling a model from an external framework to a unified intermediate representation using Apache TVM [4], which is an open-source machine learning compiler framework. The Tenstorrent TVM backend translates this intermediate representation into PyBuda API calls. Executing those calls with dummy tensors creates an execution trace, which is used to construct an initial graph where the nodes represent mathematical operations. These operations are then decomposed into lower-level operations. Optimizations such as constant folding and operation reordering are performed, and the graph is lowered to another intermediate representation. This representation consists of operations based on hardware kernels implemented by the compiler's backend. Finally, the balancer assigns resources such as the number of Tensix cores to each operation. The placer then arranges the operations spatially on the grid. The result is a human-readable intermediate representation in YAML format called Netlist, which is passed to the compiler's backend.

The Netlist defines a graph where nodes represent mathematical operations and edges data movement operations. The backend is composed of two independent compile paths. The first one compiles the high level C++ kernels (HLK) of math operations (nodes) to binaries running on Tensix cores. The second path translates data movement operations (edges) into a data movement program and compiles it into another binary running on Tensix cores.

B. TT-Metalium: low-level, bottom-up

A simple program using TT-Metalium consists of a host program and separate kernels for reading, computing and writing. The host program runs on the CPU and the kernels on Tensix cores. Different Tensix cores can run different kernels. The reader kernel runs on the first RISC-V core of a Tensix core reading tiles from the card's DRAM to SRAM. The compute kernel runs on the three middle RISC-V cores. The writer kernel runs on the last RISC-V core writing tiles from SRAM to the card's DRAM.[1]

A simple host program first instantiates a *Device*, a *Program* and a *CommandQueue* object. The *Device* object represents the Grayskull card. The *Program* object represents the kernels together with input arguments and their spatial placement on the core grid. Once the host program pushes a command to

[1]The reader and writer kernels can also be swapped such that the reader runs on the last and the writer on the first RISC-V core.

the *CommandQueue*, it is executed on the card. Because the queue allows non-blocking execution of commands, the host program does not have to wait for their completion.

Sequentially, the host program adds kernels together with their compile-time arguments and spatial Tensix core placement to the *Program* object. Then, it can pass different runtime-arguments such as memory addresses to each Tensix core individually. To store tiles, it allocates buffers in the card's DRAM and circular buffers in the Tensix core's SRAM. Subsequently, data in the host's DRAM is tilized to ensure that elements of each tile are stored consecutively in memory. To write those tiles to the card's DRAM buffer, a *WriteBuffer* command is pushed to the *CommandQueue*. Next, the *Program* object is pushed to the *CommandQueue* to execute the kernels. Finally, a ReadBuffer command is pushed to the *CommandQueue* to read tiles from the card's DRAM back to the host's DRAM and the data gets untilized.

Most of the relevant machine learning operations are already implemented using TT-Metalium and accessible through the Python library TT-NN, which acts as a high-level interface to TT-Metalium.

IV. Multi-Head Self-Attention

Basic multi-head self-attention [22] is defined as:

$$\begin{aligned}\text{MultiHead}(X) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{head}_i &= \text{Attention}(XW_i^Q, XW_i^K, XW_i^V) \\ \text{Attention}(Q, K, V) &= \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V\end{aligned}$$

where:

- $X \in \mathbb{R}^{n \times d_{\text{model}}}$: Input matrix
- $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$: Parameter matrices for head i
- d_k, d_v: Key and value dimensions
- d_{model}: Model dimension (of input embeddings)
- h: Number of attention heads
- n: Batch size (number of tokens in sequence)

First, the input vectors are linearly transformed into queries, keys and values by matrix multiplication with the corresponding parameter matrices. Then, the query and key matrices are multiplied to produce a $n \times n$ matrix of attention scores. These are scaled by $\frac{1}{\sqrt{d_k}}$ mainly for training stability. The Softmax function is applied to the scaled attention score matrix to calculate the attention weights. By matrix multiplication of the attention weights with the values, effectively a weighted summation of values is calculated for each position in the token sequence. Higher weighted values are prioritized. Hence, the term *attention*. This process is repeated for the other heads with different parameters. Finally, the results are concatenated and linearly transformed to produce the output matrix.

The computation of queries, keys and values, the multiplication of attention weights with those values and the final linear transformation are basic matrix multiplications that were already implemented for the Grayskull architecture by Tenstorrent [16]. Concatenation is a trivial operation. Therefore, this work focuses on the efficient computation of softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ for a single head.

V. Matrix Multiplication on Grayskull

To understand matrix multiplication on Grayskull [16], imagine that the first input matrix flows from the left to the right of the Tensix core grid, while simultaneously the second input matrix flows from the top to the bottom. In the end, the output matrix is laid out on the Tensix core grid such that the top left element of the output matrix is stored on the top left Tensix core and the bottom right element on the bottom right Tensix core. The outputs stored on each core were computed exclusively on the same core by matrix multiplication.

Concretely, it is implemented in the following way. First, some cores have to read tiles from the card's DRAM into their SRAM. The first column of the Tensix core grid reads a specified number of columns from the first input matrix and the first row of the core grid the same number of rows from the second input matrix. Each core reads a block of tiles. The block width/height of the first/second input matrix is the specified number. Then, each core in the first column multicasts its block to all cores in its row and each core in the first row to all cores in its column. This process repeats with the next blocks until all tiles of the input matrices are read. Given input matrices A with dimensions 8×4, B with 4×8, a 2×2 core grid and a specified block width/height of 2, the second core of the first column would read and multicast the two bottom 4×2 blocks of A.

Simultaneously, all cores compute matrix multiplication with each incoming block from the first input matrix and the corresponding block from the second input matrix. They add each result to an intermediate output block. Once all blocks are processed, this intermediate output block is the final output block and all cores write their output blocks back to the card's DRAM.

VI. Softmax

The Softmax function applied to a matrix is:

$$\sigma(Z)_{ij} = \frac{e^{Z_{ij}}}{\sum_{k=1}^{n} e^{Z_{ik}}}$$

with:

- Z_{ij}: Element in the i-th row and j-th column of the $m \times n$ input matrix.
- $\sigma(Z)_{ij}$: Element in the i-th row and j-th column of the $m \times n$ output matrix.

The natural exponential function is applied to all elements of Z and each $e^{Z_{ij}}$ is normalized by the sum of all elements in its row i. Therefore, each output element $\sigma(Z)_{ij}$ depends on all $e^{Z_{i1}}, \ldots, e^{Z_{in}}$ of the same row i but not on exponentials of other rows. So all $e^{Z_{i1}}, \ldots, e^{Z_{in}}$ have to be computed before $\sigma(Z)_{ij}$. Caching them avoids the need to recompute $e^{Z_{ij}}$ for each $\sigma(Z)_{ij}$. However, the capacity of the cache must be sufficient to store an entire row.

n of $n \times n$	CPU		Grayskull					
	Softmax		Softmax			Matrix Multiply + Scaling + Softmax		
	Recomputing	Caching	Single-Core	Multi-Core		Total	Kernel	
			Total	Total	Kernel		Total	Softmax
1024	7.37	1.15	2.67	0.261	0.178	0.29	0.174	0.0644
2048	28.8	4.51	10.9	0.582	0.524	0.686	0.573	0.26
4096	115	17.9	43.6	1.90	1.83	2.14	2	1.05
8192	460	73.1	177	7.26	7.04	—	—	—
16384	1850	297	—	—	—	—	—	—

TABLE I: Experimental results for the runtime of different implementations in milliseconds. The total runtime of Grayskull implementations is measured on the host and includes overhead like dispatching the kernel. The kernel runtime without overhead is measured on Grayskull. The total CPU and Grayskull runtimes were averaged over 100 iterations and the kernel runtimes over 10. If a runtime is not displayed, the input matrix is too large.

When the input elements Z_{ij} are large, computing $e^{Z_{ij}}$ and $\sum_{k=1}^{n} e^{Z_{ik}}$ leads to a risk of numerical overflow. To reduce it, this equation can be used:

$$\frac{e^{Z_{ij}-m_i}}{\sum_{k=1}^{K} e^{Z_{ik}-m_i}} = \frac{e^{Z_{ij}}/e^{m_i}}{\sum_{k=1}^{K} e^{Z_{ik}}/e^{m_i}} = \frac{e^{Z_{ij}}}{\sum_{k=1}^{K} e^{Z_{ik}}} \quad (1)$$

where $m_i = \max_j Z_{ij}$.

From each input element the maximum of its row is subtracted. Therefore, the maximum input element is 0, which reduces the risk of numerical overflow.

A. Softmax on CPU

The CPU implementation of the Softmax function serves as a baseline, and to observe the effect of caching the exponentials. It processes the input matrix row by row. First, it searches the maximum and subtracts it from all elements in the current row. Then, it exponentiates the elements and sums them. Finally, it normalizes the exponentials by the sum.

To measure the effect of caching, one implementation caches the exponentials in DRAM and SRAM at the summing step, and another one recomputes them at the final step. Both are written in C and use 32-bit floats. Since the attention score matrix is $n \times n$ where n is the input token sequence length, this work focuses mainly on square matrices.

*1) **Maximum sequence length**:* Because this CPU implementation is built for experimental evaluation, it allocates memory for both input and output data to avoid resetting the inputs at each profiling iteration, which would distort the results. However, A CPU implementation for production could operate *in-place*, meaning that it operates directly on the input data. The maximum dimension for such an implementation would be $n_{\max} = \sqrt{\frac{\text{memory}}{4}}$, because a float is 4 bytes. For 8 GiB of DRAM that is $n_{\max} = 46340$ and means we could process a sequence of 46340 input tokens.

*2) **Time and memory complexity**:* Each element is accessed and processed three times with constant time complexity $O(1)$. Therefore, the total time complexity is $\Theta(n^2)$, which can be observed in the experimental results (see Table I). Clearly, an in-place implementation has a memory complexity of $\Theta(n^2)$. Since the other implementation allocates twice as much memory, it has also $\Theta(n^2)$.

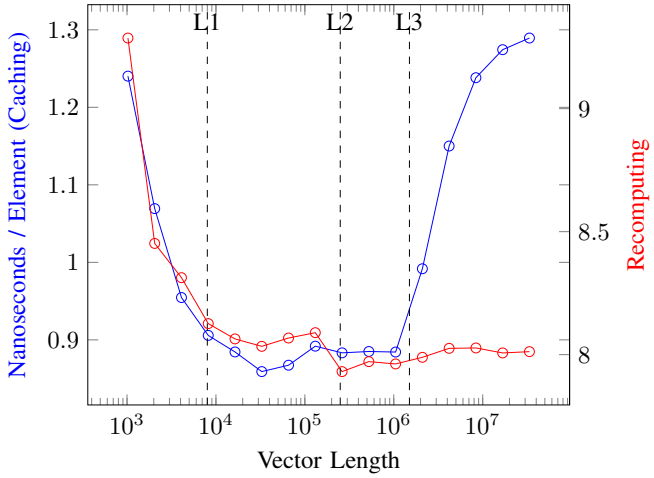

Fig. 3: Effect of caching the exponentials. Note, that the two y-axes have different scales. All CPU experiments were conducted on a single core of an Intel i5-6500 processor with 8 GB DDR4 memory, running Ubuntu 20.04. It has 32 KB L1 cache per core, 1 MB shared L2 and 6 MB shared L3 cache.

*3) **Effect of caching**:* The experimental results also show that the implementation caching the exponentials is approximately 6× faster than the one recomputing them on this specific computer configuration.

Figure 3 shows the runtime per element across varying length of a vector for both the caching and the recomputing implementation. Note, that their y-axes have different scales. Up to a vector length of 32768, the values of both implementations decrease. A likely reason is that there is a constant overhead and its relative effect on the total runtime decreases for larger vectors. Starting from a vector length of $2^{21} \approx 2.09 \times 10^6$, the runtime per element of the caching implementation increases steeply until it plateaus, while the values of the recomputing implementation remain approximately constant. The reason is that the L3 cache of this specific CPU has a capacity of 6 MB $= 1.5 \times 10^6$ Floats, which is smaller than the number of floats in the vector. With the caching implementation, this leads to cache misses. As a result, the CPU has to fetch the exponentials from DRAM, which is significantly slower. For

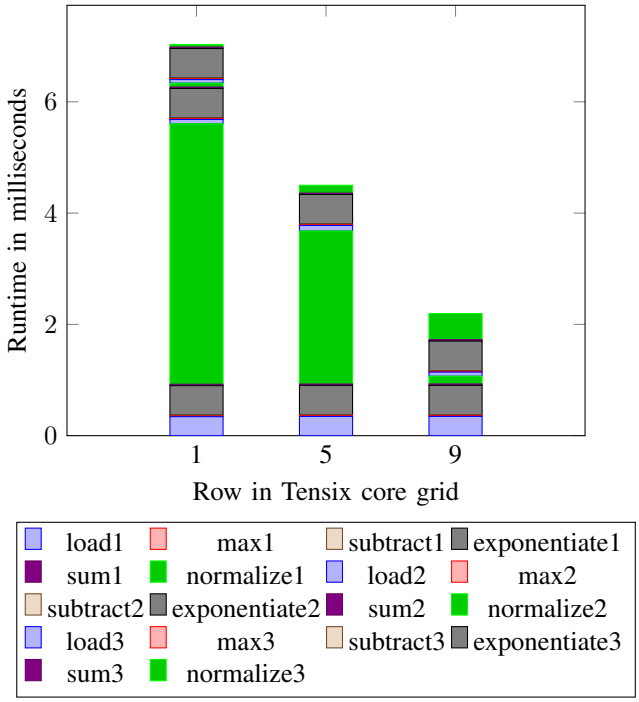

Fig. 4: Runtime distribution of the Softmax kernel with 8192×8192 input matrix. Measured on the compute core (3rd of 5 RISC-V cores) of the first Tensix core from three different rows in the core grid. For example, load1 means loading the first row of tiles. The Tensix core in the first row processes one additional row of tiles.

the L1 and L2 cache the same effect cannot be observed, likely due to less variation in bandwidth among the caches compared to the significant variation between bandwidth of L3 cache and DRAM.

In conclusion, the caching implementation is significantly faster than the recomputing one for all sequence length n, because it utilizes the DRAM for caching. But beginning from $n = 1.5 \times 10^6$, the runtime per element increases, because it cannot utilize the CPU caches anymore.

B. Softmax on Grayskull

The Softmax implementation for the Grayskull architecture can utilize a single or multiple Tensix cores. Because a row of the output matrix only depends on the same row and not on other rows of the input matrix, different rows can be processed on different cores in parallel. Since the cores compute on 32×32 tiles, the matrices are partitioned into tiles, and rows consist of tiles instead of scalars. To determine the number of rows each core processes, division with remainder is performed with the total number of rows and the number of cores:

$$\frac{n\_rows}{n\_cores} = min\_rows\_per\_core, \text{ rest: } n\_cores\_plus\_one$$

The quotient determines the minimum number of rows that each core processes. Additionally, different remaining rows are distributed to different cores starting from the upper left core, in a row-wise manner. Each Tensix core processes its rows one at a time and performs the same computation as the CPU implementation. However, it is only responsible for a subset of the rows, computes more efficiently on tiles and accesses its SRAM explicitly in contrast to implicitly accessing transparent caches.

*1) **Maximum sequence length**:* Because the SRAM of a Tensix core has a capacity of 1 MB $\approx$ 488 Tiles (Bfloat16) and a tile has a width of 32, the maximum length of a row is $488 \times 32 = 15616$ Floats. In the context of the attention operation, the maximum sequence length would be $n_{\text{max}} = 15616$. There is no limit on the number of rows.

*2) **Time and memory complexity**:* In contrast to the CPU implementation, it can process up to a constant number of rows in parallel. Since this speedup is just a constant factor, the time complexity of this implementation is $\Theta(n^2)$ as well. Because each Tensix core has to store an entire row, the memory complexity for the SRAM is $\Theta(n)$. However, the implementation has to allocate DRAM for the input and output matrices. Hence, the memory complexity is $\Theta(n^2)$.

*3) **Experimental results**:* The number of cores, that process at least one row of tiles, increases from $n = 1024$ to $n = 4096$ where all of them do. At $n = 4096$ the first 20 cores process one additional row and at $n = 8192$ all of them process 2 rows while the first 40 process one additional row.

Doubling n should quadruple the runtime according to the time complexity. However, only the runtime of $n = 8192$ is almost $4\times$ larger than the one of $n = 4096$. This is because the number of running cores increases from $n = 1024$ to $n = 4096$. For the same reason, the Grayskull multi-core vs. single-core speedup increases from approximately $10\times$ at $n = 1024$ to $23\times$ beginning at $n = 4096$. The Grayskull multi-core vs. CPU caching speedup increases from approximately $4.4\times$ at $n = 1024$ to $10\times$ at $n = 8192$.

Particularly for sequences with at least $n = 4096$ input tokens, running the Softmax operation on Grayskull would be significantly faster than on a CPU.

*4) **Runtime distribution**:* The distribution of the total runtime across all operations is shown in Figure 4 for three different Tensix cores.

First, the compute core (third RISC-V core inside a Tensix core) has to wait for the first two RISC-V cores to load the first row into SRAM and its tiles from there to registers. Because the five RISC-V cores work in parallel, the first two cores can already load the next row, while the compute core is processing the current row, ensuring that the compute core does not need to wait for subsequent rows. Hence, the runtime of the first loading operation is significant, while the runtime of the following loading operations are negligible.

The runtime of the exponentiations is significant and remains constant for different rows and Tensix cores.

The runtime of the actual normalization is a small constant. However, there is an extreme variance in the runtime of the first normalization operation across Tensix cores in different rows. The likely reason is as follows. Since normalization is the last operation, it sends the results tile by tile to the last two RISC-V cores and each time has to wait for them to write

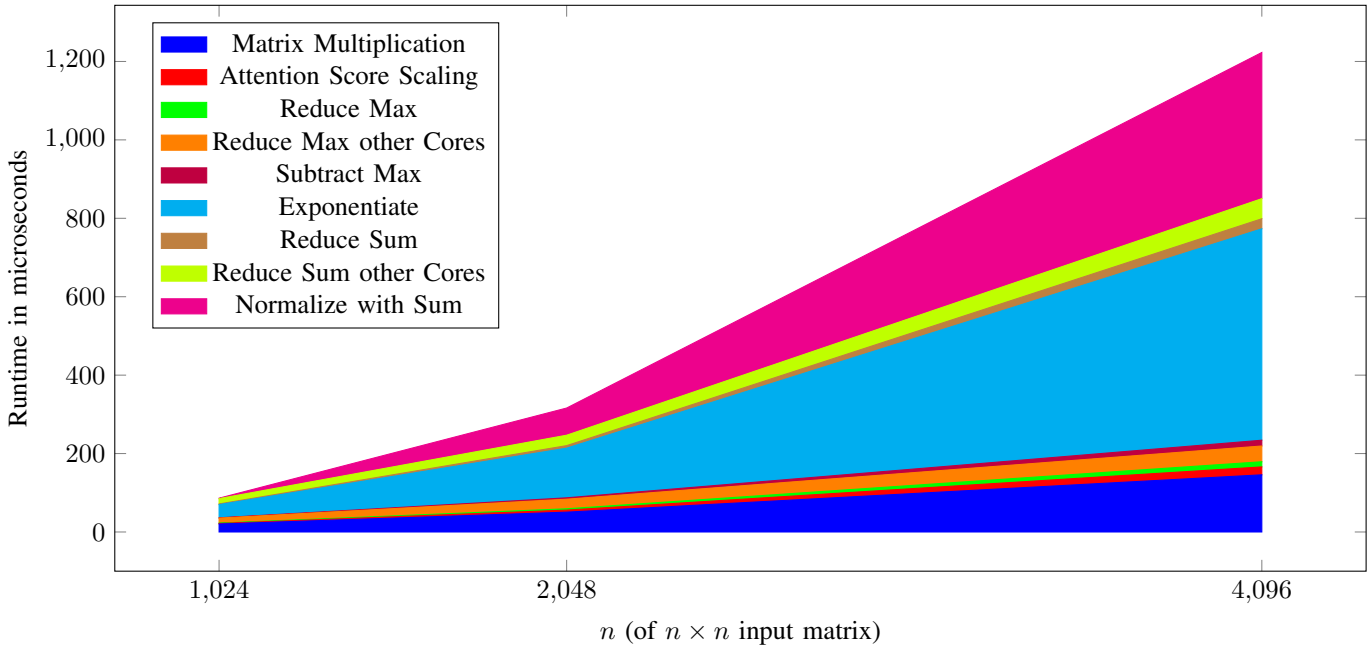

Fig. 5: Runtime distribution of the fused kernel across varying input dimensions. Measured on the compute core (3rd of 5 RISC-V cores) of the fastest Tensix core.

the tile back to DRAM. Due to the topology of the NoC, each Tensix core has to wait for the one below it to write the results to DRAM. Therefore, the operation takes longer in higher rows of the core grid.

The runtime of the reduction operations (max, sum) and of the subtraction operation are negligible.

VII. Fused Matrix Multiply + Scaling + Softmax on Grayskull

The computation of softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ is fundamental to the attention mechanism. However, separate kernels for the matrix multiplication of queries and keys, the scaling of the attention scores, and the application of the Softmax function to those scores have to read intermediate results of the previous kernel from DRAM and write their intermediate results back to it. But DRAM has lower bandwidth, higher latency, and higher energy consumption than SRAM. Additionally, the dispatching of each kernel to Grayskull introduces overhead. This motivates the use of a fused kernel reducing overhead of accessing DRAM and dispatching kernels.

The implementation of the fused kernel builds on top of a multi-core matrix multiplication implementation by Tenstorrent [16](described in Section V). After the matrix multiplication is performed, the attention score matrix is laid out on the Tensix core grid such that the top left element is stored on the top left Tensix core and the bottom right element on the bottom right core. Each Tensix core stores a subset of the attention score matrix and computes the scaling and Softmax operations on them. Since the scaling operation is defined as element-wise multiplication with $\frac{1}{\sqrt{d_k}}$, its implementation is trivial. For the Softmax operation, each Tensix core first computes the local maxima of partial rows of scaled attention scores stored in its memory. But complete rows span an entire row of Tensix cores. Therefore, each Tensix core reads the local maxima from all other cores in its row to compute the global maxima of complete rows. Then, it subtracts the maxima from its scaled attention scores and exponentiates them. In the same way, it computes the local sums, then the global sums and normalizes its exponentials with those sums. Finally, each core writes its attention weights to its individual address inside the resulting attention weight matrix in DRAM.

The following focuses on the use of the kernel in the context of the attention operation with the $n \times d_k$ input matrices Q and K, as well as a $n \times n$ output matrix of attention weights. All experiments were conducted with $d_k = 128$, since this value was used in the popular Llama3 [12].

*1) **Maximum sequence length**:* Because the SRAM of a Tensix core has a capacity of 5×10^5 floats in Bfloat16, the largest square matrix it can store is 707×707. Since the largest (for the computation utilizable) Tensix core grid is 9×12, the square output matrix has $n_{\text{max}} = 707 \times 9 = 6363$. However, in practice it is slightly less due to buffers for caching and dataflow. The largest successfully tested power of two was $n = 4096$. So with this implementation the maximum number of input tokens in a sequence is approximately 4096, which is significantly less than $n_{\text{max}} = 15616$ of the dedicated Softmax kernel.

*2) **Time and memory complexity**:* Because d_k is a constant, the dot product to produce a single attention score is $O(1)$. Since the attention score matrix has dimensions $n \times n$, the time complexity of the matrix multiplication is $\Theta(n^2)$. For the same reason, the scaling of attention scores is $\Theta(n^2)$.

Each Tensix core computes the regular Softmax operation. Additionally, it reads the local maxima and sums from other cores in its row, but this is just a constant overhead. Multiple cores provide a significant speedup, but this is just a constant factor. Therefore, the Softmax part is $\Theta(n^2)$ as well. Hence, the time complexity of the fused kernel is $\Theta(n^2)$, which is also show in the experimental results in Table I.

Since the input and output matrices are $n \times n$, the memory complexity is $\Theta(n^2)$.

*3) **Experimental results**:* Table I shows, that the speedup of the Softmax operation inside the fused kernel, compared to the dedicated multi-core Softmax kernel, is $4.1\times$ at $n = 1024$ and decreases to $1.8\times$ at $n = 4096$. The speedup is mainly due to avoiding DRAM accesses. It decreases, since the number of active cores for the dedicated Softmax kernel increases, while the fused kernel is computed on 8×8 Tensix cores for all n.

Because of this speedup and the negligible runtime of the scaling operation, the runtime of the fused kernel is at $n = 1024$ smaller and otherwise only slightly larger than the runtime of the kernel for only the Softmax operation. However, the dispatching overhead of the fused kernel is approximately twice as large and therefore, the total runtime is slightly larger.

Compared to the CPU implementation with caching, the speedup is approximately $17\times$ for $n \in \{1024, 2048, 4096\}$.

*4) **Runtime distribution**:* The primary factors influencing the runtime are matrix multiplication, exponentiation, and normalization (See Figure 5). At $n = 4096$ normalization takes approximately $2.6\times$ longer than matrix multiplication and exponentiation $1.4\times$ longer than normalization. So the runtime of the Softmax operation is significantly larger than the one of matrix multiplication. The reduction operations computing the global maxima and sums by reading the local ones from other cores influence the total runtime slightly. The runtime of all other operations is negligible.

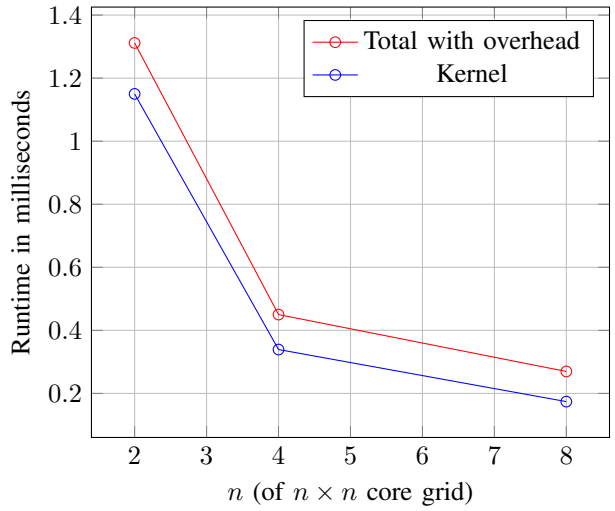

Fig. 6: Runtime of the fused kernel across varying dimensions of a square grid of cores. Measured on the host with overhead and on Grayskull without overhead.

Figure 5 also shows the $\Theta(n^2)$ time complexity.

VIII. Limitations and future Directions

Currently, the fused kernel computes the attention weights from the queries and keys, but not the queries and keys itself from the input matrix and also not the output matrix with the attention weights, output weights, and values. Future work could try to incorporate those remaining matrix multiplications into the fused kernel to reduce overhead from kernel dispatching and DRAM accesses.

The Softmax implementation inside the fused kernel computes the same global sums and maxima redundantly on all cores. Since all cores run in parallel, this has no negative effect on the runtime. Future work could implement other variants such as computing the global sums and maxima only on one core in each row and broadcasting them to the other cores. Then, one could compare energy, runtime and memory consumption.

Finally, it would be interesting to port the implementation to newer generations (e.g., Tenstorrent Wormhole) and to scale it on multiple cards.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

[3] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 17413–17426. Curran Associates, Inc., 2021.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[5] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

[6] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture, 2022. Accessed: 2024-07-11.

[7] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[8] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, dec 1974.

[9] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5156–5165. PMLR, 13–18 Jul 2020.

[10] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

[11] Chester Lam. Nvidia's h100: Funny l2 and tons of bandwidth. https://chipsandcheese.com/2023/07/02/nvidias-h100-funny-l2-and-tons-of-bandwidth/, 2023. Accessed: 2024-07-12.

[12] Meta. Introducing meta llama 3: The most capable openly available llm to date, 2024. Accessed: 2024-07-11.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[14] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[15] Tenstorrent. Commercially available tenstorrent cards. https://tenstorrent.com/cards/, 2024.

[16] Tenstorrent. Implementation of matrix multiplication on multiple cores of tenstorrent grayskull. https://github.com/tenstorrent/tt-metal/blob/main/tt_metal/programming_examples/matmul_multicore_reuse_mcast/matmul_multicore_reuse_mcast.cpp, 2024.

[17] Tenstorrent. Tenstorrent grayskull architecture. https://github.com/tenstorrent/tt-metal/blob/main/METALIUM_GUIDE.md, 2024.

[18] Tenstorrent. Tenstorrent grayskull network-on-chip. https://docs.tenstorrent.com/tenstorrent/v/tt-buda/hardware, 2024.

[19] Tenstorrent. TT-Buda. https://github.com/tenstorrent/tt-buda, 2024.

[20] Tenstorrent. TT-Metalium. https://github.com/tenstorrent/tt-metal, 2024.

[21] Jasmina Vasiljevic, Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic, Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, Utku Aydonat, Tony Zhou, Syed Zohaib Gilani, Armond Paiva, Joseph Chu, Djordje Maksimovic, Stephen Alexander Chin, Zahi Moudallal, Akhmed Rakhmati, Sean Nijjar, Almeet Bhullar, Boris Drazic, Charles Lee, James Sun, Kei-Ming Kwong, James Connolly, Miles Dooley, Hassan Farooq, Joy Yu Ting Chen, Matthew Walker, Keivan Dabiri, Kyle Mabee, Rakesh Shaji Lal, Namal Rajatheva, Renjith Retnamma, Shripad Karodi, Daniel Rosen, Emilio Munoz, Andrew Lewycky, Aleksandar Knezevic, Raymond Kim, Allan Rui, Alexander Drouillard, and David Thompson. Compute substrate for software 2.0. *IEEE Micro*, 41(2):50–55, 2021.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[23] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

[24] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[25] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17283–17297. Curran Associates, Inc., 2020.