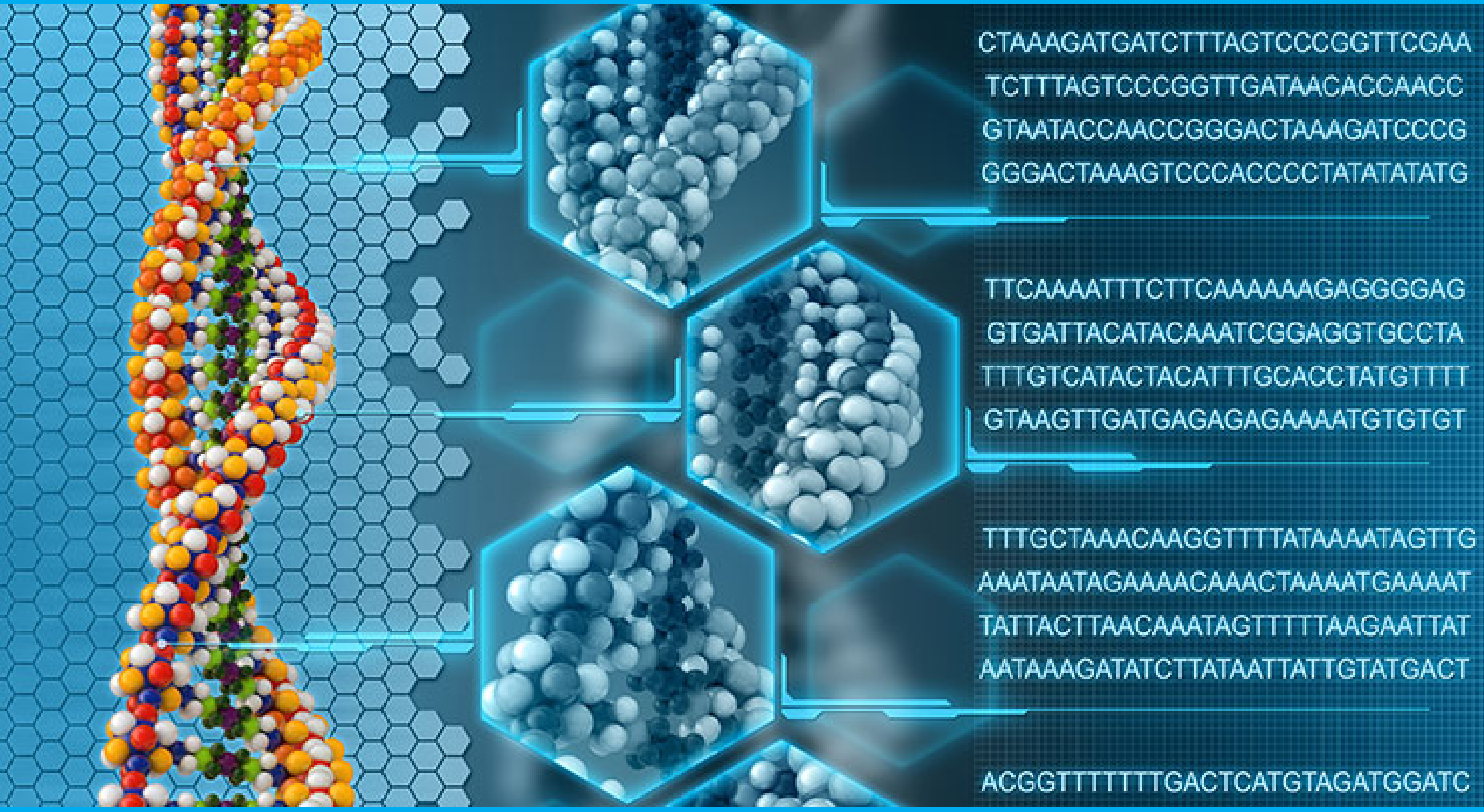
High Performance ASIC Processor Design for DNA Basecallers

William Trinh


CTAAAGATGATCTTTAGTCCCGGTTCGAA
TCTTTAGTCCCGGTTGATAACACCAACC
GTAATACCAACCGGGACTAAAGATCCCG
GGGACTAAAGTCCCACCCCTATATATATG
TTCAAAATTTCTTCAAAAAAGAGGGGAG
GTGATTACATACAAATCGGAGGTGCCTA
TTTGTCATACTACATTTGCACCTATGTTTT
GTAAGTTGATGAGAGAGAAAATGTGTGT
TTTGCTAAACAAGGTTTTATAAAATAGTTG
AAATAATAGAAAACAAACTAAAATGAAAAT
TATTACTTAACAAATAGTTTTTAAGAATTAT
AATAAAGATATCTTATAATTATTGTATGACT
ACGGTTTTTTTGACTCATGTAGATGGATC

TUDelft

High Performance ASIC Processor Design for DNA Basecallers

by

William Trinh

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday October 7, 2024 at 10:00 AM.

Student number:	4995236	
Project duration:	December 13, 2023 – October 7, 2024	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Prof. dr. H. P. Hofstee,	TU Delft
	Dr. R. T. Rajan,	TU Delft

This thesis is confidential and cannot be made public until December 31, 2024.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Abstract

Genomics has revolutionized medicine and biological research by providing deeper insights into the genetic makeup of organisms, advancing our understanding of diseases, and enabling personalized medicine. These breakthroughs are driven by advancements in genome sequencing technologies, bioinformatics, and the aid of neural networks. The advent of third-generation sequencing technology has further accelerated progress by allowing for long-read sequencing, which enhances the accuracy and efficiency of genome assembly. Oxford Nanopore Technologies (ONT) offers advanced sequencers that use nanopore-based technology to read DNA sequences in real-time. However, the raw sequenced data contains noise and requires a basecalling stage to read DNA sequences with the required accuracy. Basecalling relies on deep neural networks (DNNs) to achieve high-accuracy reads, but the significant computational power required makes basecalling a costly process, especially in real-time applications. Current hardware accelerators used for these compute-intensive basecallers are state-of-the-art GPUs that cost over $\$10,000$ per unit.

This thesis explores the design of a custom hardware accelerator for ONT's basecalling program, Bonito, which aims to provide a cost-effective alternative to existing accelerators such as Nvidia GPUs and Groq Tensor Streaming Processors (TSPs). Bonito's DNN is dominated by five Long Short-Term Memory (LSTM) layers, accounting for 90% of its execution time. The custom accelerator targets these compute-intensive LSTMs to reduce execution time. This work provides a comprehensive analysis of LSTM performance and behavior on GPUs and Groq TSPs. It emphasizes the architectural benefits and limitations in the context of basecalling. Furthermore, it also evaluates an Application-Specific Integrated Circuit (ASIC) implementation of an existing FPGA-based LSTM accelerator design.

The analysis shows that Bonito's High-Accuracy model (HAC) LSTM layers contain many sequential matrix multiplications that are compute-intensive and require high memory bandwidth to accommodate the data transfers. Furthermore, Bonito's small problem size, with 384 features per vector input, causes GPUs to not fully utilize available compute cores. Combined with slow per-core performance, GPUs executing LSTMs achieve only 13.5% of the maximum TFLOP/s with FP16 precision.

Groq uses a heterogeneous architecture with fast separate MXM units executing matrix multiplications, and VXM units executing point-wise operations. Data travels between these different compute units through streaming channels and MEM units. The LSTM analysis on Groq showed three main issues. First, Groq uses a 320-element wide data channel to transfer data across the chip, whereas Bonito has 384 hidden features as input. This leads to the 384-element input being sliced in two 192-element partial inputs as Groq supports physical tensors up to 320-element long, effectively doubling the cycle cost by using two slices instead of one. Second, performing matrix multiplications requires these slices to be transferred from MEM to MXM, by executing reload operations to the MXM weight buffer. This data transfer consumes the bandwidth on the streaming channel, which introduces stalls in the pipeline and clock cycles are spent on MEM operations, instead of compute operations in MXM or VXM. Lastly, the analysis shows that the VXM forms a bottleneck in the LSTM execution, accounting for 50% of the clock cycles, whereas the MXM accounts for the other 50%. This shows that the special functions and additions inside an LSTM cell are slowing down Groq's overall performance.

After the existing architecture analysis, the ASIC evaluation showed a synthesis result, where one block of 384 LSTM Processing Engines (PEs) achieves a clock speed of 434MHz and costs $8.07mm^2$ on a 40nm process node. Putting these PEs on a Groq-based chip layout of $725mm^2$ in area size, the 40nm-based PEs can achieve 79.3 TFLOP/s at FP16 precision. By correcting the 40nm process node to Groq's 14nm process node, the 14nm-based PEs achieved 448 TFLOP/s at FP16. These results suggest that a custom LSTM accelerator could compete in performance with state-of-the-art solutions while being more cost-effective.

Future work is suggested to investigate a cycle reduction in the multiply-accumulate stage and to evaluate the ASIC design using a modern process node technology, as the current ASIC design uses a 2008-based 40nm process node. This work helps future development in further optimizing a custom LSTM accelerator specified for Bonito's DNN requirements, paving the road toward more affordable genome sequencing.

Preface

The process of completing this thesis has been both an educational and rewarding challenge, offering valuable insights and growth at every step. At the outset of the thesis, I had a relatively clear vision of the end goal, but the path to get there was largely uncharted. Navigating through the unknown, with limited prior knowledge of hardware development, was both daunting and challenging at times. As a Computer Science student, originally focused on software, I decided to head into the direction of low-level embedded engineering in 2022. Before I knew it, I was standing here with a thesis in computer engineering, with newfound knowledge of accelerators, hardware design, and genomics.

I would like to thank Zaid and Peter for their excellent guidance and supervision during this thesis. Their experience and positive attitude helped me overcome the huge wall of jumping from a software background to a hardware background. Also, I would like to thank the ABS Group of TU Delft and my fellow master's students for their interesting discussions and insights. Lastly, I would like to thank my friends and family for their support and motivation throughout this journey.

William Trinh
Delft, October 2024

Contents

1
Introduction

Genomics is a field of biology that studies the complete set of genes of organisms. Genomics allows researchers to gain a deeper understanding of how genes function, how organisms and their genes have evolved throughout history, and how certain diseases interact with certain medications. A genome consists of many genes, where genes contain segments of DNA. The average human genome is believed to contain 25000 genes. To show the importance of genomics, some breakthroughs in recent years include the Human Genome Project [1] (1990), the CRISPR-Cas9 system [2] (2013), and constant advancements in the field of pharmacogenomics. These discoveries, respectively, resulted in the first complete DNA reading of the human genome, the first successful DNA modification treatment, and effective epidemic control of COVID-19 [3]. The impact of genomics on our society is unprecedented and significantly changes the lives of many for the better.

To advance genomics, scientists developed different sequencing techniques to read genomes. These techniques form the cornerstone of DNA data reading inside the genome. Currently, third-generation sequencing, also known as long-read sequencing, is widely used by scientists to read DNA. Third-generation sequencing is available through PacBio or Oxford Nanopore Technologies (ONT).

This thesis focuses on ONT, which developed tiny *nanopores* inside a *flowcell* to read the DNA bases in a genome. The reading of DNA bases is done by pulling the DNA strand through a nanopore, where the nanopore acts as a biosensor and measures the change in current, depending on the type of DNA base passing through the nanopore [4]. The *flowcell* sequencing devices contain 512 or 2675 nanopores, named MinION or PromethION respectively [5] [6]. However, the raw data from these nanopores are noisy and not accurate without additional data processing of the reads. Hence, the *basecalling* step processes this noisy signal and “calls” the correct bases with high accuracy, using deep neural networks (DNN).

Basecalling cost

Basecalling is performed on computers that require expensive hardware, which drives up the cost of genome sequencing. Inside the basecaller, the DNNs use compute-intensive matrix operations. Graphical Processing Units (GPUs) are widely used for DNNs, with varying types of GPUs. There are consumer-grade and server-grade GPUs, where large DNNs, such as Chat-GPT, use many server-grade GPUs to enable larger neural network sizes. Furthermore, modern GPUs have CUDA cores focused on general matrix operations, and Tensor cores focused on pure matrix multiplications.

In order to perform basecalling on real-time nanopore sequencing reads, state-of-the-art GPUs are required, which are very powerful but also very expensive, costing more than $10,000 per GPU. The following example illustrates the computational cost, using ONT’s basecaller program Bonito and the High-Accuracy (HAC) basecalling model at FP16 precision. Each individual nanopore samples at 3200Hz. To give an indication of the hardware required for basecalling, a single small MinION flowcell with 512 nanopores requires at least a compute power of 20 Tera Floating-Point Operations (TFLOP) per second at FP16. A single large PromethION flowcell with 2675 nanopores requires at least 97 TFLOP/s at FP16. In contrast, a consumer-grade high-end GPU, such as the Nvidia RTX 2080 Ti and priced at $1000, can deliver up to 27 TFLOP/s at FP16 precision and up to 104 TFLOP/s using

Tensor cores at FP16. The consumer-grade RTX 2080 Ti can keep up with a small MinION for real-time sequencing of a Bonito HAC model.

However, to process multiple large PromethION flowcells, it becomes clear how compute intensive the basecalling process is. To process multiple PromethION flowcells in real-time, a last-generation enterprise-grade Nvidia A100 GPU is required. This A100 GPU is priced at $10000 and delivers up to 78 TFLOP/s at FP16 and 312 TFLOP/s using Tensor cores at FP16. In other words, performing real-time basecalling is expensive compute-wise, requiring a large investment to obtain suitable hardware. If a cheaper accelerator can achieve the same performance as an Nvidia A100 GPU for less than $10000, the cost of genome sequencing is reduced.

Hence, this thesis looks at accelerating the basecalling process, with the use of existing accelerators and the use of a custom ASIC accelerator. Specifically, the Long Short-Term Memory (LSTM) layers in ONT's basecaller program Bonito are researched and accelerated. The goal is to enable real-time sequencing for a PromethION flowcell, and potentially multiple PromethION flowcells, while lowering the cost of sequencing a genome.

1.1. Problem definition and research questions

The main research goal of this thesis is to address the possibilities of designing an Application-Specific Integrated Circuit (ASIC) for LSTM neural network models and to determine whether this area of research is viable for further development to make genome sequencing cheaper. This leads us to the following research questions:

1. What are the computational limitations within the LSTM neural network models?
2. How do existing hardware accelerators enable LSTM neural network models to achieve higher performance?
3. What can be learned from existing accelerators that can be used to design a high-performance ASIC for LSTM neural network models?

1.2. Thesis layout

First, Chapter 2 covers the basics of basecalling and ONT's Bonito basecalling tool. Furthermore, the basic hardware architecture of a GPU and a Groq accelerator, specialized for neural networks, is presented. Chapter 3 covers the LSTM algorithm, the computational analysis, and the cost of real-time basecalling with ONT's MinION and PromethION sequencing devices. Chapter 4 presents an architectural analysis of LSTM performance and behavior on GPUs and Groq accelerator cards. From this analysis, a comparative analysis is performed to determine the strengths and weaknesses of each accelerator, when used to accelerate Bonito's basecalling process. Chapter 5 covers the research on ASIC design and how well the current design performs, compared to existing GPUs or Groq accelerators. Chapter 6 presents an overview of the limitations and potential improvements for each type of accelerator discussed in this thesis, followed by an insight into what improvement avenues are possible for the current ASIC design. Furthermore, a performance comparison between GPUs, Groq TSPs, and the ASIC design is presented. Chapter 7 closes this thesis with conclusions and recommendations for future research directions.

2
Background

Genome sequencing is crucial in genomics, providing insights into genetic variations, disease mechanisms, and evolutionary biology. Many modern lifesaving medicines exist thanks to genome sequencing [7]. Currently, Oxford Nanopore Technologies (ONT) released their *nanopore* sequencing devices, called *flowcells*, that use third-generation sequencing, also known as long-read sequencing. The raw data from these nanopores require a basecalling process to convert the noisy raw data into accurate DNA bases.

This chapter further explains the process of genome sequencing and the process of basecalling. Additionally, the specific basecalling program Bonito is introduced. Afterward, the architecture of different hardware accelerators is presented, which has been used to analyze basecalling performance in the next chapters.

2.1. Genome sequencing

A genome contains the complete set of chromosomes, genes, and DNA of a living organism. For example, a human genome contains 23 pairs of chromosomes. Inside each human chromosome are genes that encode certain proteins that define cell behavior, such as our eye color. These genes are built by sequences of DNA, where the DNA consists of different DNA base types. Figure 2.1 shows the different elements present in a genome.

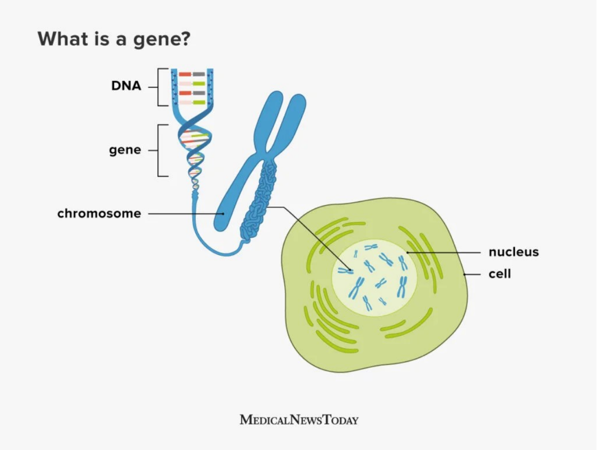


Figure 2.1: Diagram explaining how genomes, genes, and DNA are related. [8]

Genome sequencing and DNA sequencing are terms often used interchangeably. However, their use cases differ slightly. Genome sequencing focuses solely on sequencing the entire genome, whereas DNA sequencing focuses on a segment of the DNA and is a subset of genome sequencing. DNA sequencing can be used for small DNA sections, individual genes, and entire genomes.

Nowadays DNA sequencing is in its third generation, where scientists use a technique called long-read sequencing. Long reads consist of 60K up to 2M bases [9]. Previously, second-generation short-read sequencing produced shorter reads, usually between 50 to 400 bases per short read [9]. On average, the human genome is estimated to contain 3 billion base pairs. To sample a genome with sufficient accuracy, a concept called oversampling is used, where an actual DNA base, passing through a biosensor, is sampled multiple times instead of only once. This is done to reduce the negative influence of any outliers from the biosensor.

For example, using second-generation sequencing, reading a human genome with 10 samples per DNA base can produce 450 million short-reads that need to be assembled. With longer read lengths from third-generation sequencing, it is possible to represent the genome sequence using only 900,000 reads that need to be assembled [10]. Therefore, third-generation sequencing has the benefit of requiring less effort to reassemble all the reads into the sequenced DNA string.

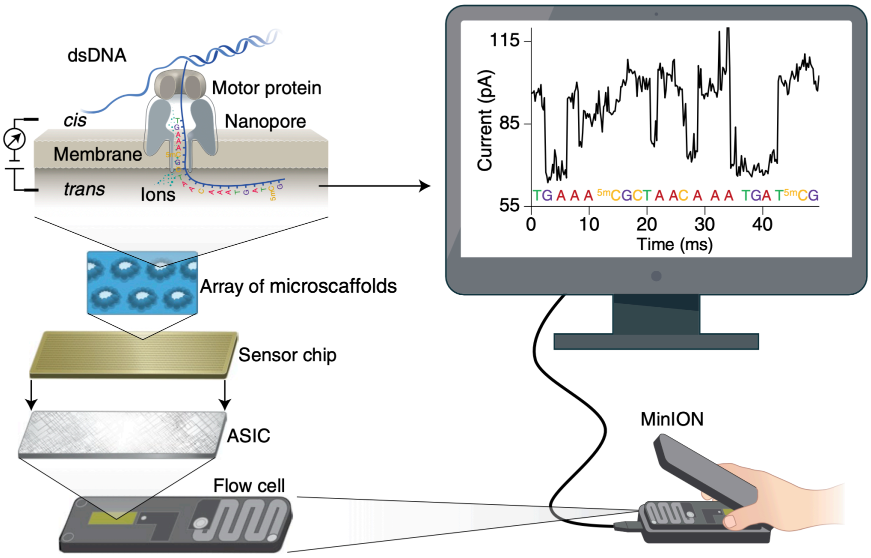


Figure 2.2: Diagram showing the working of a nanopore inside an ONT MinION sequencing device. The output signal from a nanopore is displayed as a time-series sequence, with the basecalling process assigning the correct bases [11].

Different techniques have been developed over the years to read the DNA bases. In this thesis, the focus lies on ONT and their *nanopore* technique. ONT has developed a new long-read DNA sequencing technique, named nanopore sequencing [12]. These sequencing devices use small nanopores, with up to 512 nanopores on a MinION flowcell and up to 2675 nanopores on a larger promethION flowcell. In Figure 2.2, the diagram illustrates the DNA strand being pulled through a nanopore. While the DNA strand gets pulled through the nanopore at a certain speed, the nanopore measures the change in current in pico-ampère (pA) and creates a time-series signal. These signals are noisy, which on its own is not accurate enough to decode the signal into a read of DNA bases. Therefore, a basecalling tool is used to process the raw signal and ”call” the bases with a high accuracy, using deep neural networks.

Each nanopore has a certain sampling rate, expressed in Hertz (Hz), and has a certain pull-through rate, expressed in bases per second (bps). Consider a MinION flowcell, containing 512 nanopores, each generating 4000 samples per second. Ideally, the neural network can basecall 4000 samples at batch size = 512 within 1 second, to keep up with the real-time sampling of the nanopores. This shows that a single MinION flowcell generates a lot of sampling data, which need to be processed by a basecaller.

2.2. Basecalling and Bonito

Basecalling is the process of converting the electrical signals of a DNA strand into base types (that is, the DNA bases C, G, A, and T and RNA bases C, G, A, and U). Figure 2.2 shows how a DNA strand is pulled through the *nanopore* “reader” and the computer assigns the correct base type. Assigning the correct base type is often done with advanced algorithms that use neural networks to achieve high accuracy.

ONT currently has two open-source basecalling models that are actively being developed on. First, their flagship program Dorado is written in C++ and optimized for performance [13]. Second, their development program Bonito is written in Python and is intended for research and development [14]. The thesis focuses on Bonito as the research topic due to the faster development time with Python.

Bonito uses a machine learning model consisting of different types of neural networks. Figure 2.5 shows a diagram of the neural network layers in Bonito for Fast, High Accuracy (HAC), and Super Accuracy (SUP) models. The model starts with 3 convolution (Conv) layers, followed by 5 Long Short-Term Memory (LSTM) layers. Afterward, a decoder layer finishes and produces an output. In Figure 2.3, the bottom diagram shows that LSTMs contribute to nearly 90% of the computational load.

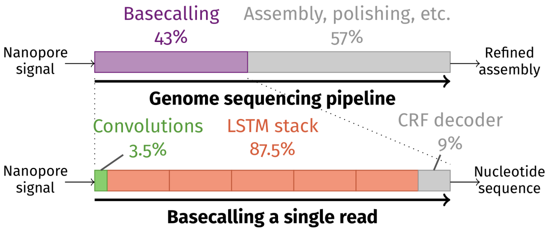


Figure 2.3: A breakdown showing how 43% of the genome sequencing pipeline is spent on basecalling. Inside this 43%, nearly 90% is spent on LSTM computation. [15]

Bonito has 3 hyperparameters that influence both accuracy and inference speed: the number of model weights (also known as the hidden size), the batch size, and the chunk size. The number of model weights depends on the type of accuracy desired by the user, where a Fast, High Accuracy (HAC), or Super Accuracy (SUP) model can be selected. The batch size represents how many reads are simultaneously processed during each model inference. The chunk size represents how long each read should be. The type of accuracy model affects how many model parameters are used. Figure 2.5 shows for each type of accuracy how many features are used in the neural network layers. For example, the HAC model uses 384 as hidden size.

First, a Conv layer is applied. Figure 2.4 is an example Conv layer, which has a 7-element wide 1D *input* vector, a 3-element *kernel* filter, and a 1-channel 5-element wide *output* vector. The Conv layer will apply a kernel filter on the first 3 elements in the input, and *stride*, or move, to the right by 1 element. The kernel filter will perform a dot-multiplication with the input, before summing the results into a 1-element kernel output. These kernel outputs result in a 5-element wide output vector. With more convolution channels, multiple kernels can be applied in the same convolution layer. Then, this 1D input can lead to a 16-channel 2D matrix output, with each vector 5-element wide.

Returning to the Bonito HAC model example, the first 3 Conv layers convert the input sequence of the nanopore into vectors represented by 384 channels, which become the 384-feature vectors for the LSTM layers. Without too many details, the Conv layers look at 5 elements in the input sequence, then apply the kernel filters based on the trained weights, and produce a 384-feature vector representation of those 5 elements. These 384-feature vectors will be used in the LSTM layers. Then, the Conv layer *strides* 1 element to the right and repeats the steps.

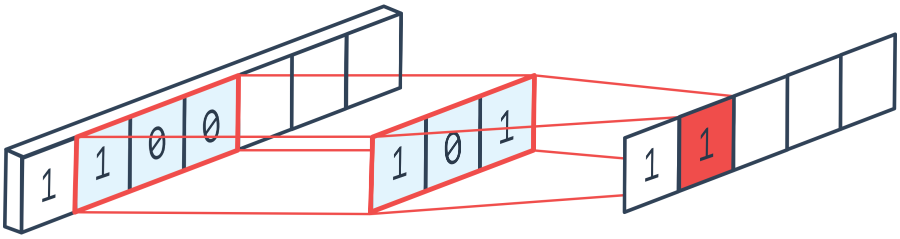
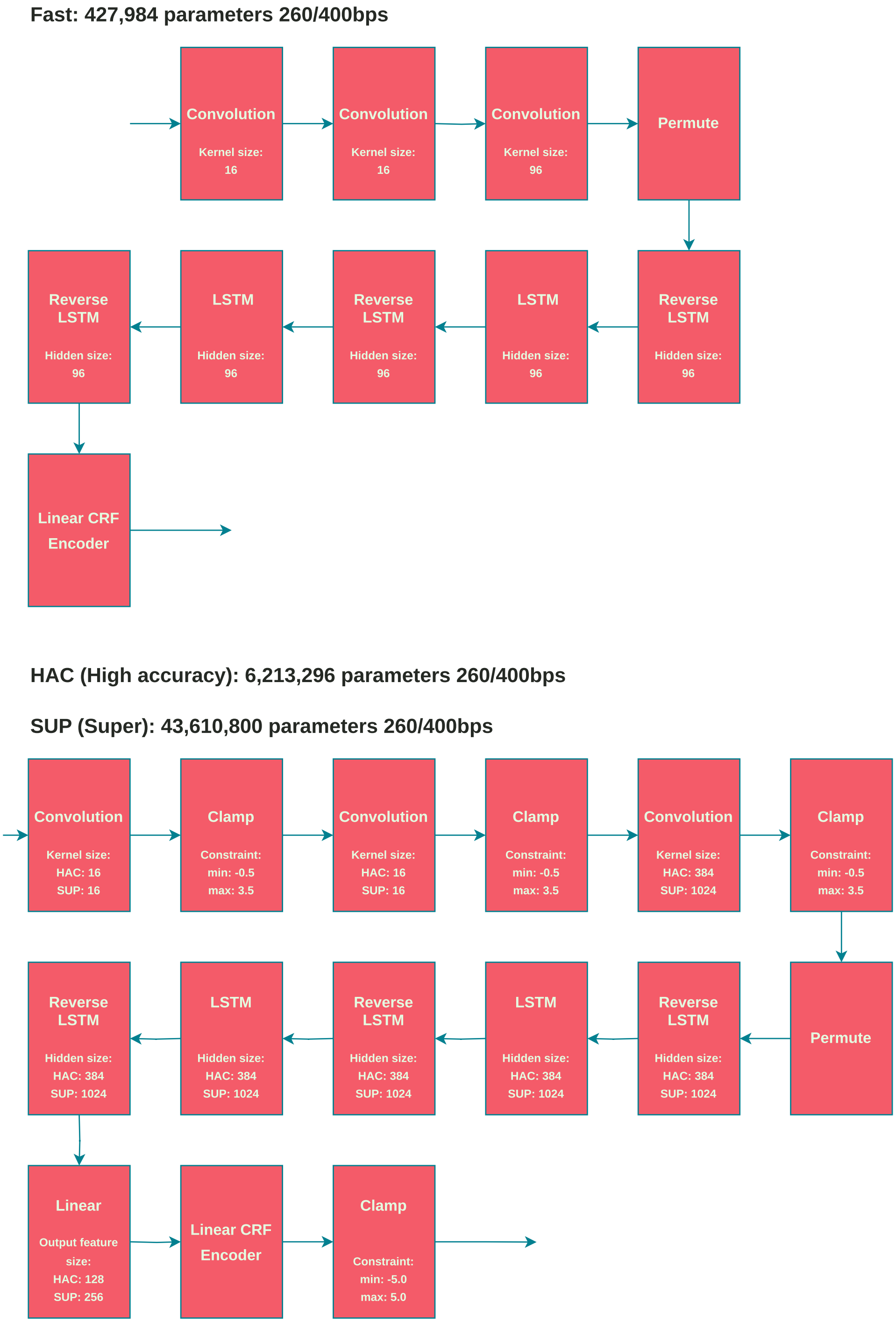


Figure 2.4: This figure shows how a 1D vector is processed in a convolution layer, using a kernel filter of 3 elements. The output of the convolution layer is a vector with each output element based on the kernel filter applied. This output is also called an output channel. [16]

The chunk size represents how many samples are present in the input read, and the chunk size also determines how long the LSTM sequence becomes. Due to the sequential nature of LSTM, the longer a sequence becomes, the longer its execution runs to process the read. To mitigate this problem, using the massive parallelism in GPUs allows Bonito to execute a very large batch size that processes multiple reads per inference. Doing so mitigates the long execution time for each individual time-consuming LSTM sequence. However, the GPU still faces bottlenecks in the LSTM layers, which consume close to 90% of the execution time. In chapter 3, an in-depth analysis of LSTM's computational limit is presented.


Fast: 427,984 parameters 260/400bps

Convolution
Kernel size: 16
Convolution
Kernel size: 16
Convolution
Kernel size: 96
Permute
Reverse LSTM
Hidden size: 96
LSTM
Hidden size: 96
Reverse LSTM
Hidden size: 96
LSTM
Hidden size: 96
Reverse LSTM
Hidden size: 96
Linear CRF Encoder

HAC (High accuracy): 6,213,296 parameters 260/400bps

SUP (Super): 43,610,800 parameters 260/400bps

Convolution
Kernel size: HAC: 16 SUP: 16
Clamp
Constraint: min: -0.5 max: 3.5
Convolution
Kernel size: HAC: 16 SUP: 16
Clamp
Constraint: min: -0.5 max: 3.5
Convolution
Kernel size: HAC: 384 SUP: 1024
Clamp
Constraint: min: -0.5 max: 3.5
Permute
Reverse LSTM
Hidden size: HAC: 384 SUP: 1024
LSTM
Hidden size: HAC: 384 SUP: 1024
Reverse LSTM
Hidden size: HAC: 384 SUP: 1024
LSTM
Hidden size: HAC: 384 SUP: 1024
Reverse LSTM
Hidden size: HAC: 384 SUP: 1024
Linear
Output feature size: HAC: 128 SUP: 256
Linear CRF Encoder
Clamp
Constraint: min: -5.0 max: 5.0

Figure 2.5: Overview of Bonito models [17]

2.3. GPU architecture

Graphics Processing Units (GPUs) are specialized accelerators that enable massive parallelism, as they are equipped with many small processing cores. Figure 2.6 shows the typical Nvidia GPU architecture with many small processing cores. Compared to an average CPU with 8 cores, a modern Nvidia A100 GPU has 6912 CUDA cores [18], specialized in parallel calculations. However, these CUDA cores are typically slower individually, compared to CPU cores [19]. GPUs were originally intended to accelerate graphics on a consumer PC, because many parallel matrix operations are used in computer graphics applications, such as in 3D games. With the rise of neural network applications, GPUs gradually saw increased uses in machine learning, big data analysis, and other applications. Many operations in these applications can be parallelized, such as executing large matrix operations on many data points. Nowadays, there is a substantial difference in GPUs, such as consumer-grade GPUs focused on 3D graphics performance, and server-grade GPUs focused on machine learning performance.

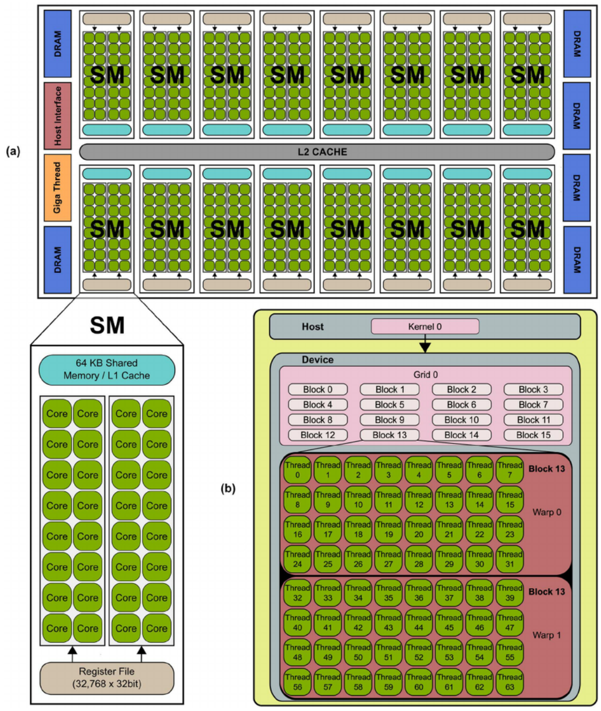


Figure 2.6: Typical Nvidia GPU Architecture [20]

For Nvidia's architecture, the CUDA cores are clustered into several Streaming Multiprocessors (SM). These SM units are equipped with the following compute resources as shown in Figure 2.7. Since the release of the Volta architecture in 2017, SM units are also equipped with Tensor cores, besides CUDA cores. In each SM unit, the majority of hardware is allocated to matrix operations and a small portion to registers and special function units (SFUs). Figure 2.7 shows that Nvidia uses three types of compute cores, INT32 and FP32 inside CUDA cores, and Tensor cores. These Tensor cores are optimized for matrix multiplications. Each SM unit is responsible for one task and distributes the requested operation across the smaller CUDA cores or Tensor cores inside each SM. Hence, applications that can be parallelized into small chunks benefit the most from the GPU's architecture. A prime example is matrix multiplication. These large matrix multiplications form the basis for many deep learning applications, such as in genomics or Large Language Models (LLMs) like Chat-GPT, where GPUs are mainly used.

However, utilizing GPUs efficiently can be challenging at times, as not all applications can be optimally executed on such an architecture. For example, a sequential application or latency-critical application would not fit a GPU, because GPU cores are relatively slow compared to fast general-purpose processors, such as a CPU. Sequential applications typically benefit from lower latency per individual core, rather than having many slower parallel cores available, as each iteration is dependent on the previous result.

Furthermore, utilizing GPUs leads to a larger memory overhead, as data needs to be transferred from the host CPU to the device GPU. Hence, small applications experience a performance hit due to slower transfer times between devices, compared to transfer times between CPU and RAM. With large applications that can fully utilize all the available cores, these transfer times become negligible, as the speedup is more than the overhead cost.

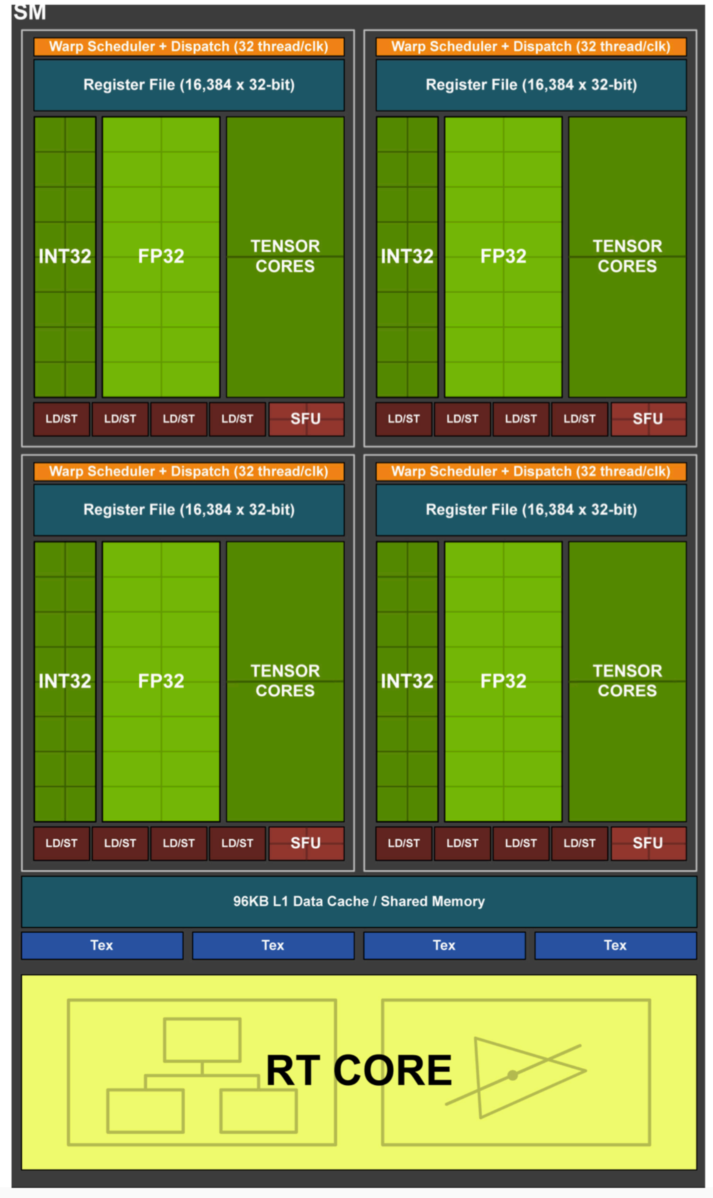

SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Figure 2.7: SM unit computational overview on Turing architecture [21]

2.4. Groq architecture

The AI startup Groq released their first Tensor Streaming Processor (TSP) chip at ISCA 2020 [22]. Fundamentally, Groq applies a different architecture compared to mainstream processors, such as the Nvidia Ampere architecture or Intel Raptor Lake architecture. Groq places emphasis on software, such as the Groq compiler or Groq assembler, where the architecture philosophy is design simplicity and to allocate as many tasks and instruction optimizations as possible to the software. This leads to the Groq TSP using its available chip area for pure computational execution.

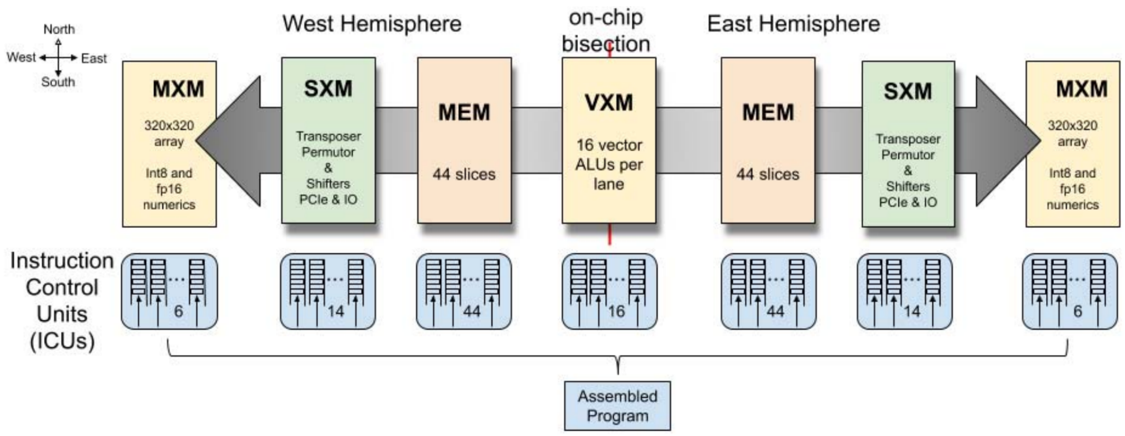


Figure 2.8: Overview of the heterogeneous modules on the Groq TSP, showing 1 of the 320 data lanes [22]

First, Groq uses a heterogeneous design, characterized by its unique APIs for operation. Figure 2.8 shows four different units. First, the VXM represents the Vector Execution Module, where all element-wise computation is performed. The MEM represents the Memory Execution Module and contains the SRAM slices on-chip to store and retrieve data during execution. The SXM represents the Switch Execution Module that is responsible for data manipulation, such as transposing, permuting, but also on-chip data movement across the 320 lanes in the north-south direction. In Figure 2.9, the y-axis represents the 320 lanes, which act similarly to SIMD. Last, the MXM represents the Matrix Execution Module, which handles the matrix operations. Using this approach allows the Groq chip to optimize each module and simplify the components for their specific tasks.

In Figure 2.9, a 3D diagram of the chip is shown, where the interconnects between each module are shown from the z-axis. These east or westbound stream lanes handle the data transfer between modules, for which there are 64 streams in total.

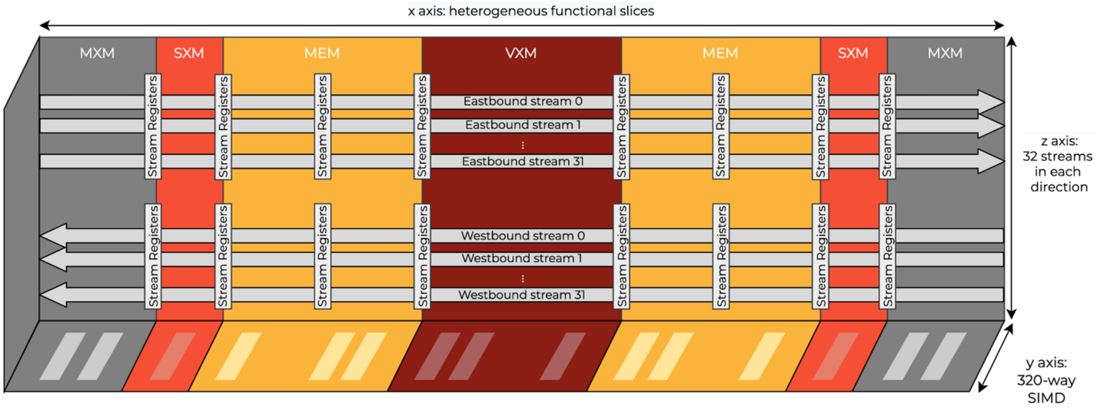


Figure 2.9: This figure rotates Figure 2.8 90 degrees, showing the streaming registers and streaming channels [22]

Second, Groq's TSP is a deterministic processor. By determining the exact instruction sequence off-chip, the Groq compiler knows exactly when and where every piece of data is located. This is possible because Groq does not use any predictive branching, speculative execution, or any control circuitry during its runtime. Instead, the Groq compiler determines every instruction step during compilation and loads all the instructions onto the Groq chip, before execution. This makes the Groq TSP deterministic and predictable in performance.

Furthermore, Groq's TSP has no caching hierarchy. For reference, an Nvidia A100 GPU has 96MB of the fastest L1 cache and additional L2 and L3 level caches. Groq has 88 SRAM memory slices on-chip that provide 220MB of on-chip memory capacity at a maximum of 55TB/s bandwidth, and 64 streaming registers to transfer data between slices, at a maximum of 20TB/s. Compared to the traditional cache hierarchy approach, Groq has allocated 41% of the chip area to SRAM modules that are directly accessible by the compute units, as shown in Figure 2.10. This enables very fast data transfers between compute units, which significantly improves inference latency. However, the instructions for execution are also stored on the same SRAM memory on-chip. This results in the user-available SRAM memory to be lower than 220MB.

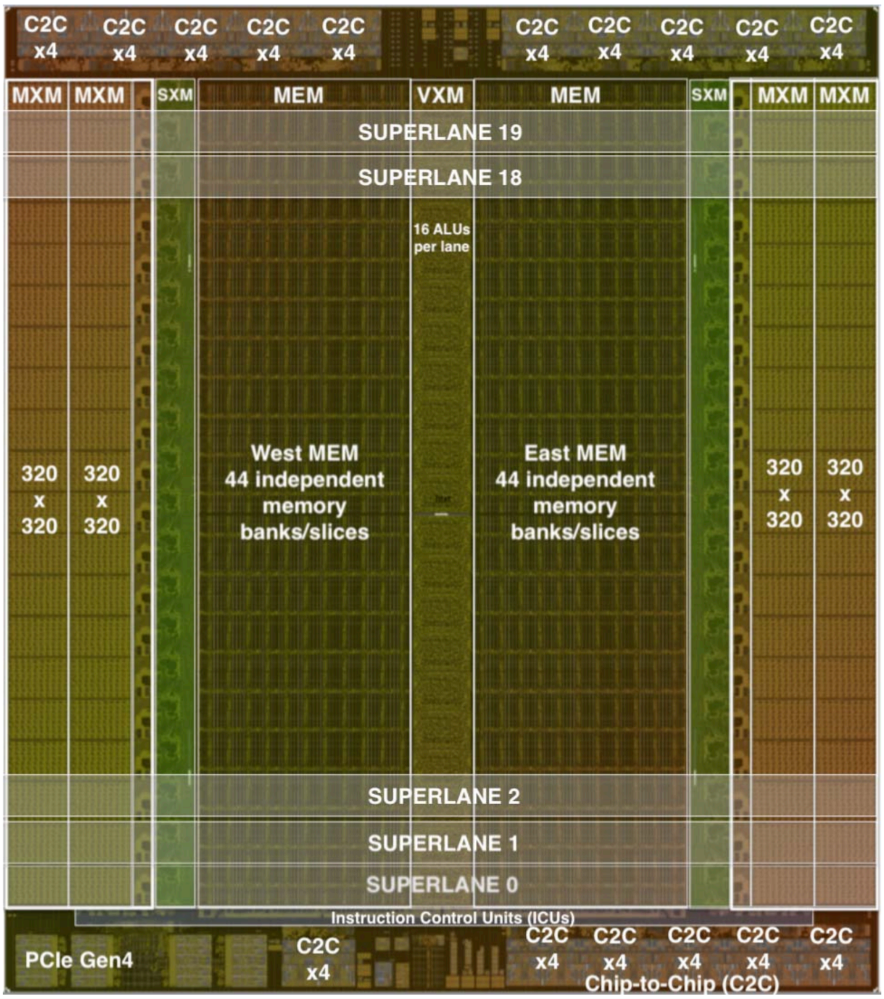


Figure 2.10: Die photo of the Groq TSP, annotated by its components [22]

Additionally, the Groq architecture is highly dependent on the Groq development suite. In Figure 2.11, a diagram is shown on the steps to create a Groq executable. In this thesis, Bonito has been implemented with PyTorch and converted to ONNX, in order for the Groq compiler to handle the remaining steps. There is another option to develop custom applications with Groq's bare-metal GroqAPI. However, the GroqAPI documentation and any related programming insights or tutorials are publicly unavailable on the internet. Hence, the GroqAPI was disregarded as a viable path.

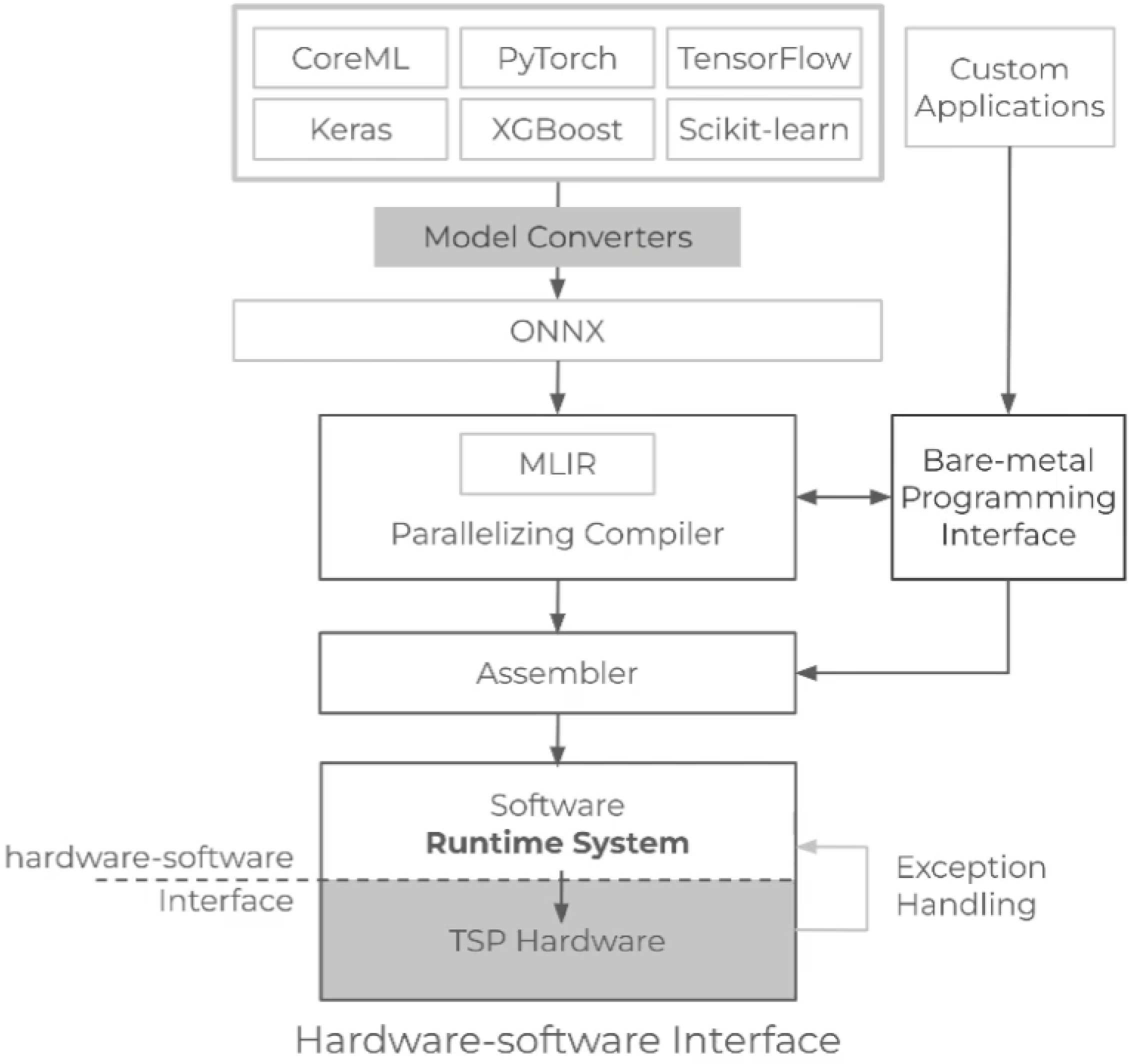


Figure 2.11: Overview of the design stages in the Groq software suite [23]

3 LSTM analysis

This chapter covers a comprehensive analysis of the LSTM algorithm and its performance limitations. Afterward, a short analysis is presented on the performance cost of real-time basecalling with ONT's MinION and PromethION sequencing devices.

LSTM algorithm

An LSTM model is a type of Recurrent Neural Network (RNN) that is commonly used to process sequential data. In Figure 3.1, a 3-cell LSTM diagram shows how each cell is dependent on the previous cell. Inside each cell, are multiple mathematical operations present. In Figure 3.2, there are three gates visible, named (from left to right) the *forget gate* f_t, *input gate* i_t, g_t, and *output gate* o_t. The forget gate regulates what portion of information from the previous cell should be remembered. The input gate regulates the importance of the new input that needs to be remembered for the current cell. The output gates regulate what portion of information needs to be forwarded to the next cell. These gates allow the LSTM model to determine what is relevant for the current and future dependency in the LSTM sequence.

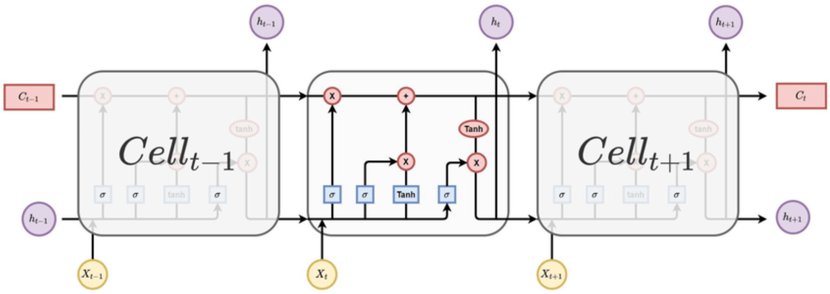


Figure 3.1: In this figure, a detailed scheme of a single LSTM cell is shown. In addition, the information flow and the hidden state and cell state flows are highlighted. [24]

To better understand how LSTMs operate within Bonito, it is useful to consider the Bonito HAC model and the model parameters. The batch size = 512, the hidden size = 384, and the chunk size is 10,000. section 2.2 explained how the convolution layer applies a kernel filter on the input vector DNA sequence and transforms each kernel output into a 384-feature vector for the LSTM layer. This means that each LSTM time step processes only a small section of the initial input vector DNA sequence. When the chunk size is 10,000, the input for the LSTM sequence consists of 10,000 time steps, or x_0 goes to $x_{10,000}$ where each x_t is the input for LSTM cell $cell_t$. For example, $cell_0$ uses input x_0, processes the LSTM cell and outputs h_0, c_0, before $cell_1$ uses the new input x_1 and h_0, c_0 from $cell_0$. This repeats until $cell_{10,000}$ produces the final $h_{10,000}, c_{10,000}$. Based on the hidden size, input x_t itself is a vector with 384 weight parameters, represented in dimensions [1, 384]. Lastly, applying a batch size of 512 means that you process 512 independent x_t input vectors at the same time. Instead of processing 512 x_t vectors across 512 threads separately, using batches allows the LSTM to process 512 independent

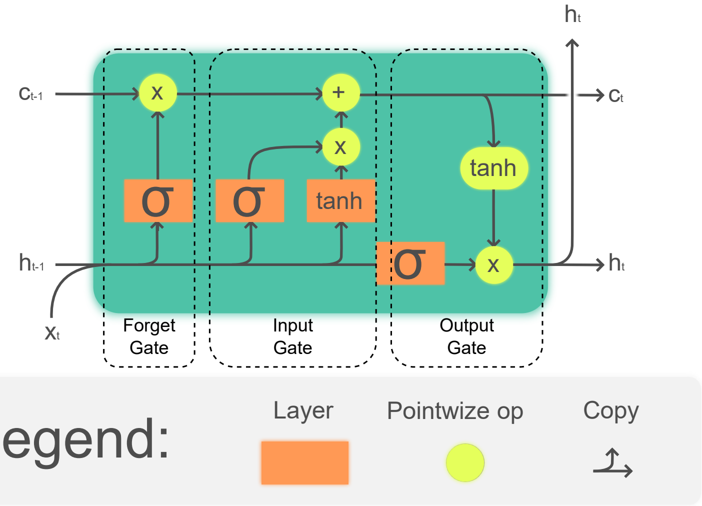


Figure 3.2: Internal structure of an LSTM cell. [17]

x_t vectors as 1 large matrix. Batching leads to an input matrix of [512, 384], instead of 512 separate [1, 384] x_t vectors. This greatly improves efficiency, as modern accelerators are optimized for matrix calculations, rather than individual vector calculations.

Each LSTM time step computes the following equations, shown in Equation 3.8. In Equation 3.8, W represents the different types of weight matrices used as model parameters, with the specific type annotated by its subscript. In Bonito, the LSTM parameters are expressed as batch size = k, hidden = input size = $m = n$. W is a square matrix, based on the hidden size m. x_t is a vector [1, m] with batch size = 1, or a matrix [k, m] with batch size = k. $W \times x_t$ results in a matrix [k, m]. Likewise, h_{t-1} also is of shape [k, m].

$$f_t = \sigma(W_{f\_input} x_t + W_{f\_hidden} h_{t-1} + b_f) \tag{3.1}$$

$$i_t = \sigma(W_{i\_input} x_t + W_{i\_hidden} h_{t-1} + b_i) \tag{3.2}$$

$$g_t = \tanh(W_{g\_input} x_t + W_{g\_hidden} h_{t-1} + b_g) \tag{3.3}$$

$$o_t = \sigma(W_{o\_input} x_t + W_{o\_hidden} h_{t-1} + b_o) \tag{3.4}$$

$$c_t = c_{t-1} \odot f_t + i_t \odot g_t \tag{3.5}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.6}$$

$$\tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{3.7}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.8}$$

Figure 3.3: Equations used in LSTM

The first four formulas represent the calculations for the 3 gates in Figure 3.2. Instead of calculating the four formulas sequentially, they can be calculated in parallel, as these calculations are independent of each other. By concatenating the weight matrices, it is possible to reduce the original eight separate multiplications to only two multiplications. By concatenating the four weight input matrices into a single W_{input} matrix, a single multiplication can be executed to calculate the intermediate results for the four weight input matrices. Likewise, this idea applies to the hidden weight W_{hidden} as well. Afterward, W_{input} and W_{hidden} are split to obtain the results for the eight multiplication steps. This is an important property that modern neural network libraries use to optimize performance.

Computational cost of LSTMs

In the mathematical equations, there are five types of operations present: matrix multiplication, matrix addition, sigmoid, tanh, and hadamard product. Each type of operation has a certain Floating Point Operation (FLOP) cost. Of all these operations, matrix multiplication is the most expensive operation. As an example, multiplying matrix $A[k,m]$ and matrix $W[m,n]$ gives $A \times W = C[k,n]$, with batch size = k, hidden = input size = $m = n$. This would result in a total FLOP cost of $2 \times m \times n \times k - kn$ per matrix multiplication. There are three steps involved when performing a matrix multiplication. To calculate one output element in C, the first step is an element multiplication per pair of row-column combination that costs m FLOP. Then, there is an addition of $m - 1$ to add all results of the first step, such that it gives the final result for a single element in output matrix C. These two steps occur for every single element in matrix C. Hence, to calculate a single output element, it costs $2m - 1$ FLOP and to calculate the entire output matrix, it will cost $kn \cdot (2m - 1)$ FLOP.

Addition for matrix $C[k,n]$ and matrix $D[k,n]$ will cost $k \times n$ FLOP, as it is essentially an element-wise addition per matrix element. The sigmoid(σ) and tanh activation functions are assumed to have a linear cost of n operations for n elements, due to the availability of SFUs.

Bonito's LSTM layers use hidden size = input size = $m = n$ and batch size = k. Thus, the final cost for each LSTM cell is as follows:

$$f_t => kn + kn + kn(2m-1) + kn(2m-1) = 2kn + 2kn(2m-1) = 4kmn \tag{3.9}$$

$$i_t => 4kmn \tag{3.10}$$

$$g_t => 4kmn \tag{3.11}$$

$$o_t => 4kmn \tag{3.12}$$

$$c_t => 3kn \tag{3.13}$$

$$h_t => 2kn \tag{3.14}$$

Figure 3.4: FLOP cost per formula

The total FLOP cost is then $16mnk + 5kn$. For example, the HAC model has input size and hidden size = 384, and batch size = 1. This gives a FLOP cost of $16 \cdot 384^2 \cdot 1 + 5 \cdot 1 \cdot 384 = 2361216$ FLOP. With batch size = 512, the cost per LSTM cell would be roughly $16 \cdot 384^2 \cdot 512 + 5 \cdot 512 \cdot 384 = 1.21$ GFLOP.

LSTM performance limitations

Figure 3.2 represents a single LSTM cycle. With a larger sequence length, the neural network becomes a long sequential network with multiple LSTM cells in sequence. Hence, the outputs h_t, c_t of cell $Cell_t$ becomes the inputs h_{t-1}, c_{t-1} of cell $Cell_{t+1}$. This leads to a problem where LSTM models with long sequences tend to be limited in terms of latency, due to the sequential cells. Every LSTM cell must wait for its previous LSTM cell result. Figure 3.1 shows an example of what longer LSTM sequences would look like. The problem is that long LSTM sequences cannot map each individual LSTM cell to a GPU core to calculate in parallel. Only increasing batch sizes enables for more parallel computation.

In the context of Bonito, the default hyperparameters for the HAC model use batch = 512, chunk = 10,000, hidden = 384. The corresponding input tensor for the LSTM for this configuration becomes a tensor of 384 hidden features, 10000 sequence samples per read, and 512 reads per batch. The calculations on the 384 hidden features can be efficiently parallelized, as these are matrix operations independent for each sequence sample. The biggest hurdle for LSTMs are the long sequences of samples per read. Each iteration in the sequence depends on the previous result. Thus, for each read, only 1 LSTM cell can be processed at a time using only a handful of GPU cores. Therefore, using a large batch size of 512 enables a GPU to efficiently utilize its computational resources by processing multiple read sequences in parallel. However, larger batch sizes cost more VRAM to store the weights, inputs, and intermediate results. Furthermore, a longer sequence length means that more input data need to be stored in the VRAM as well. In the next sections, the impact of sequence length on memory usage will be discussed.

Real-time inference requirements with flowcells

To process the reads generated by ONT's flowcells, there are three important numbers to consider. First, the *number of nanopores* determines the batch size in which the data is generated. Second, the *bases per second* determine the speed at which the DNA strand is pulled through a nanopore. Third, the *sampling rate* at which the bases are sampled determines the final number of reads in a sequence per second.

One MinION flowcell samples at 3200 Hz on average, with 512 nanopore channels. Thus, every second, there are $3200 \cdot 512 = 1,638,400$ data points that need to be processed by Bonito's neural network. Processing 1 sample with batch size = 512 costs 1.21 GFLOP. Considering the focus on the 5 LSTM layers present in Bonito, it would require $1.21 \cdot 5 = 6.05$ GFLOP per 1 sample with batch size = 512. Thus, processing 3200 samples would require $3200 \cdot 6.05 = 19,360$ GFLOP/s or 19.36 TFLOP/s.

When using one larger PromethION flowcell with 2675 nanopores, it would require around x5 more TFLOP/s, at 97 TFLOP/s. In other words, the flowcell generates data at a batch size of 2675, compared to the batch size of 512 for the MinION. Now, the largest *PromethION 48* sampling machine contains 48 individual PromethION flowcells. Thus, this *PromethION 48* would require $46,560$ TFLOP/s of compute performance. That converts to a batch size of $2675 \cdot 48 = 128,400$. The large PromethION 48 machine is equipped with 4 Nvidia A100 GPUs to enable real-time sequencing on the HAC model, proving how compute-intensive the basecalling process is.

4

Bonito and LSTM performance

Bonito's basecalling model mainly consists of LSTM layers. This section will cover a comprehensive analysis on LSTM performance with a GPU and Groq accelerator, where a thorough investigation is presented on execution behavior and resource utilization. In chapter 6, the limitations and suggested improvements are further discussed. There, an estimation on performance increase is given, based on the suggested improvements.

4.1. Bonito on GPUs

In this section, Bonito's LSTM performance is analyzed on the GPU and the execution behavior is presented, in the context of the GPU architecture. Furthermore, the results focus on Bonito's HAC model, using hidden size = 384. batch size = 512, and chunk size = 10,000.

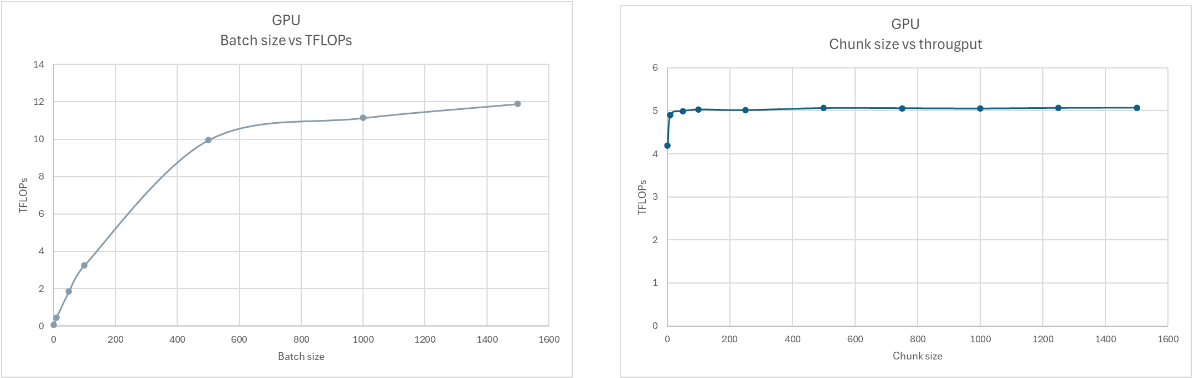
The baseline GPU used is an Nvidia RTX 2080 Ti, based on the Turing architecture. This GPU is designed on a 12nm process node, with a chip-die area of $754mm^2$. Furthermore, the RTX 2080 Ti can achieve up to 26.90 TFLOP/s on normal FP16 operations, and up to 107.60 TFLOP/s on Tensor FP16 operations using Tensor Cores. These Tensor cores are used for matrix multiplications only, whereas the 26.90 TFLOP/s is peak performance for generic vector or matrix operations. For the remaining sections, the GPU is assumed to have a peak performance of 107.60 TFLOP/s due to the majority of FLOP being matrix multiplications.

This GPU is equipped with 68 SM units [21]. The majority of the SM area is dedicated to matrix operations and only a small portion is dedicated to the special function units (SFU), responsible for LSTM's Sigmoid and Tanh functions. A large portion of calculations per LSTM cell performs a matrix multiplication. After multiplication follows an addition, executed on the same compute units as matrix multiplication. After that, a Sigmoid and Tanh calculation follows, utilizing the limited-available SFU. The following section covers the performance results of LSTM executions on the baseline GPU. Afterward, the limiting factors that explain the performance results are discussed.

GPU performance results

To understand the impact of batch size and chunk size, the following diagrams from Figure 4.1 to Figure 4.3 show the cost in memory, latency, and throughput with Bonito's HAC LSTM layer.

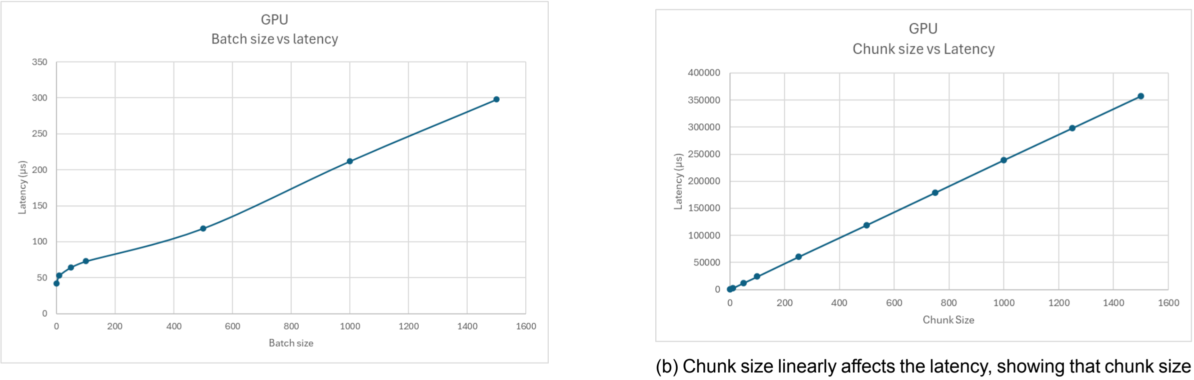
The most important observation from Figure 4.1 to Figure 4.3 is the impact of batch size on throughput. With a large enough batch size, the throughput reaches a ceiling as seen in Figure 4.1a. This shows that scaling batch size does not linearly increase throughput. Furthermore, Figure 4.2a shows that latency increases with larger batch sizes. However, the latency increases at a slower rate than the overall throughput. Comparing batch = 100 to batch = 1000, the latency increases by x2.9, while throughput increases by x10. This confirms the increase in overall throughput in Figure 4.1a. Lastly, the impact of chunk size does not impact the overall throughput. This is expected, as increased chunk size leads to an increased number of LSTM cells in a sequence. Only the VRAM usage increases with chunk size, which is logical, since more inputs are required to be stored on the VRAM, with longer sequences. Likewise, batch size affects VRAM usage as well, although at a slightly different rate.



(a) Displaying the performance on GPU, for different batch sizes. The peak achieved TFLOP/s is around 12 TFLOP/s.

(b) Increasing chunk size does not affect overall throughput performance. Results are based on batch = 512.

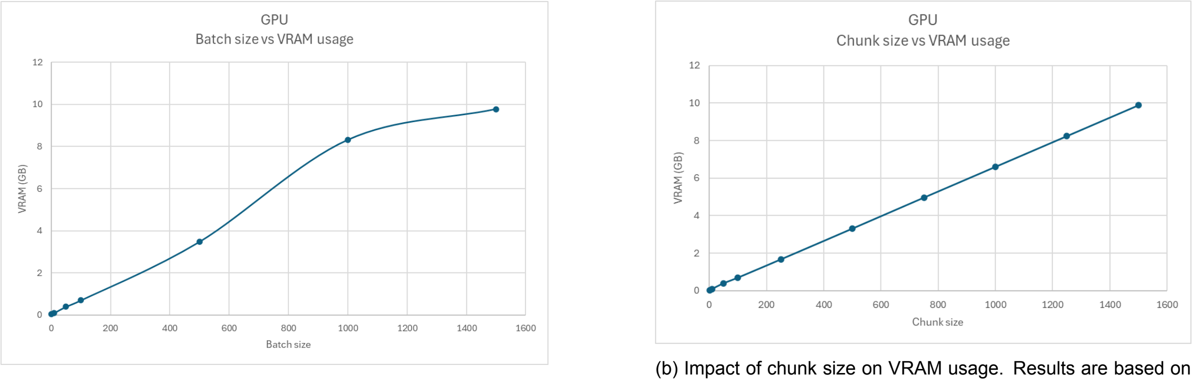
Figure 4.1: How batch and chunk size affect total throughput.



(a) The GPU shows higher latency with increased batch sizes per 1 LSTM cycle.

(b) Chunk size linearly affects the latency, showing that chunk size does not alter overall throughput. Instead, chunk size increases the LSTM sequence linearly. Results are based on batch = 512.

Figure 4.2: How batch and chunk size affect total latency.



(a) Increased batch size has increased memory cost on GPU

(b) Impact of chunk size on VRAM usage. Results are based on batch = 512.

Figure 4.3: How batch and chunk size affect VRAM usage.

With a general idea of how LSTMs perform for certain batch sizes and chunk sizes, the next step is to understand the execution behavior of LSTMs in terms of cycles. Nvidia Nsight Compute is a performance profiling tool that provides a comprehensive report on clock cycles and execution behavior. Figure 4.4 shows the distribution for each arithmetic type when processing 1 LSTM cell in Bonito's HAC model. It becomes apparent that matrix multiplications contribute to the majority of the total clock cycles. In Figure 4.4, with a batch size = 512 and hidden size = input size = 384, the total clock cycles would cost $156,681$ cycles. This translates to a performance of 13.71 TFLOP/s, not accounting for cycles handling data movement. Additionally, Nvidia Nsight Systems provided a GPU kernel trace with

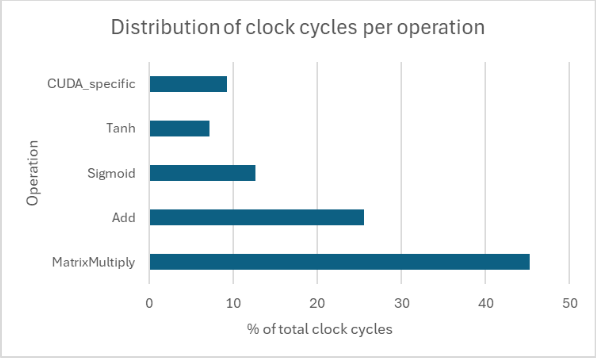
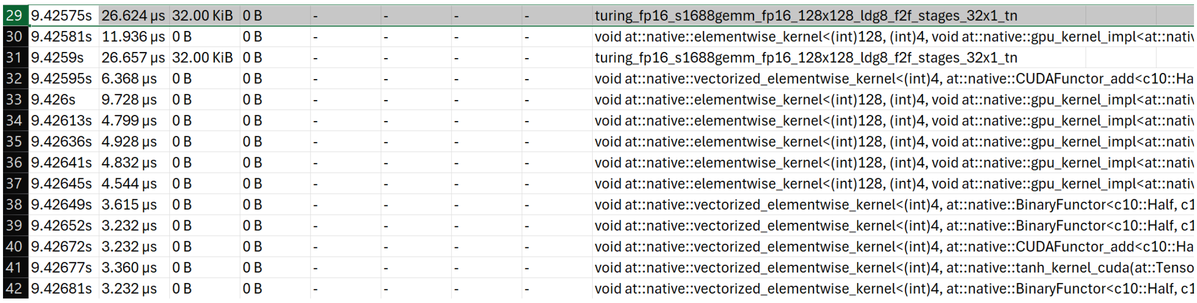


Figure 4.4: Distribution of cycles for all LSTM arithmetic.

timing numbers, as shown in Figure 4.5, resulting in an execution time of $117\mu s$ per 1 LSTM cell. Since the RTX 2080 Ti is rated at 1350MHz, we can calculate the average number of LSTM cells executed per second. This yields 8540 LSTM executions per second. Based on the FLOP per LSTM cell, the performance is estimated at $8540 \cdot 1.21 = 10333$ GFLOP/s or 10.3 TFLOP/s. The numbers from Nsight Compute and Nsight Systems confirm the average LSTM performance of 11.5 TFLOP/s, when using Bonito's HAC model parameters.

Interestingly, Nsight Compute reports a memory throughput of 138.34 GB/s, occupying 27% of available memory bandwidth. It shows that the LSTM execution is not memory-bound, but rather compute-bound. Returning to the TFLOP per second used, the Bonito HAC model LSTM execution only uses roughly 10% of the available TFLOP/s capability. This shows that the current Bonito LSTM layer is too small to fully utilize all available resources while being latency-bound by the individual GPU core performance.



29	9.42575s	26.624 μs	32.00 KiB	0 B	-	-	-	-	turing_fp16_s1688gemm_fp16_128x128_ldg8_f2f_stages_32x1_tn
30	9.42581s	11.936 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
31	9.4259s	26.657 μs	32.00 KiB	0 B	-	-	-	-	turing_fp16_s1688gemm_fp16_128x128_ldg8_f2f_stages_32x1_tn
32	9.42595s	6.368 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CUDAFunctor_add<c10::Ha
33	9.426s	9.728 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
34	9.42613s	4.799 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
35	9.42636s	4.928 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
36	9.42641s	4.832 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
37	9.42645s	4.544 μs	0 B	0 B	-	-	-	-	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl<at::nativ
38	9.42649s	3.615 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<c10::Half, c1
39	9.42652s	3.232 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<c10::Half, c1
40	9.42672s	3.232 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CUDAFunctor_add<c10::Ha
41	9.42677s	3.360 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::tanh_kernel_cuda(at::Tenso
42	9.42681s	3.232 μs	0 B	0 B	-	-	-	-	void at::native::vectorized_elementwise_kernel<(int)4, at::native::BinaryFunctor<c10::Half, c1

Figure 4.5: GPU Trace of kernel activities, when performing 1 Bonito HAC model LSTM cell with batch size = 512.

With these results, it is evident that Bonito HAC model's LSTM layers runs significantly below the maximum TFLOP/s capacity of 107.60 TFLOP/s at FP16 using Tensor cores. To understand why the numbers are low, an analysis is presented that discusses the observed behavior.

Limiting factors on GPU

According to Nsight Compute, the multiplication kernel achieves around 25% SM occupancy, meaning it utilizes around 25% of the available compute resources. This SM occupancy number is most likely lower than 100% due to the size of the matrix multiplication being too small to occupy all available compute units. Figure 4.6 shows how the GPU maps its parallel tasks across the available resources. Based on the 25% SM occupancy, it could mean that blocks or grids are not fully occupied, as the optimized sizes will vary per type of computation problem [25]. However, the high cost of 35,177 cycles is primarily caused by the slow execution and low performance of each GPU core. For instance, by halving the HAC batch size, the matrix multiplication takes 23,130 clock cycles and achieves an SM occupancy of 19%. Half the FLOP/s are executed with batch size = 256, while clock cycles decrease by only 34%. It becomes clear that the individual core compute performance is low. The best way to increase TFLOP/s performance is by increasing batch size, such that the task can be parallelized across more compute cores. However, higher batch size comes at the cost of a larger data footprint on cache and memory to store the required inputs, weights, and intermediate results.

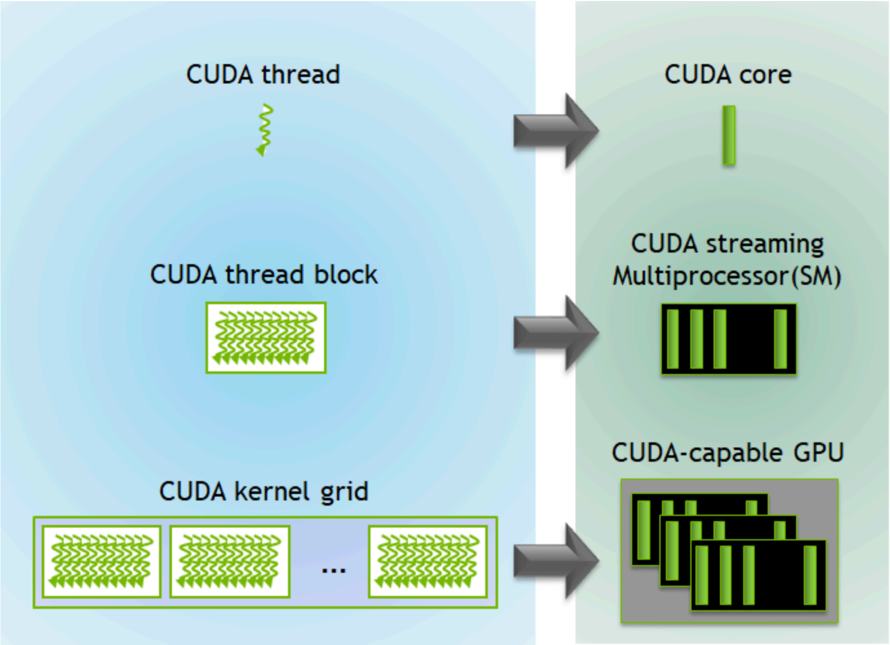


Figure 4.6: Diagram showcasing the CUDA kernel hierarchy [25].

A useful metric to consider is the *arithmetic intensity*, expressed in FLOP per byte. It is used to determine whether a certain operation is compute- or memory-bound. For the LSTM computation, the arithmetic intensity can be calculated. With batch size = 512 and hidden size = 384, each LSTM cell will process $16 \cdot 384^2 \cdot 512 + 5\dot{3}84^2 = 1,208,696,832$ FLOP or 1.21 GFLOP. This requires $384 \cdot 512 \cdot 2 \cdot 2 = 786,432$ Bytes of data_in and hidden_in. For data outbound for the next LSTM cell, also $786,432$ bytes are transferred. Lastly, the four types of weight transfer a total of $384^2 \cdot 4 \cdot 2 = 1,179,648$ bytes. Hence, a total of 2.75 MB is moved between the cores and L2 cache. This gives an arithmetic intensity of $\frac{1{,}208{,}696{,}832}{2{,}752{,}512} = 439.125$ FLOPs per byte.

The current Bonito HAC model is not memory-bandwidth bound. However, when the L2 cache is too small for larger models, the memory bandwidth becomes a limiting factor with a GPU architecture [26]. The L2 cache, which acts as on-chip cache memory, is around 5MB large for the RTX 2080 Ti. When large calculations are performed, the required data does not fit on the L2 cache of 5MB. This requires data to travel from VRAM to the L2 cache before it can be used to perform calculations on. This can be interpreted as a GPU being unable to accommodate peak request bandwidth.

Model	Input/Hidden Size	Input & Hidden Storage (MB)	Weight Storage (MB)
Super model	1024	4.0	16.8
HAC model	384	1.5	2.4
Fast Model	98	0.38	0.15

Figure 4.7: Memory requirements for a single Bonito LSTM layer

Figure 4.7 shows the memory usage for a single LSTM cell execution. The weight data of 2.34 MB with the HAC model is static throughout the whole LSTM execution sequence, but the input and hidden data are different for every LSTM cell execution. Consider the HAC model, requiring 1.5MB of cache storage for the input and intermediate data. This number comes from a matrix with 384 features and a batch of 512. With 2 bytes at FP16 per element and 4 matrix instances from the 4 LSTM gate outputs, this forms a cache memory cost of $384 \cdot 512 \cdot 2 \cdot 4 = 1564672$ bytes. In between LSTM cells, the L2 cache stores the data previously required for LSTM cell $Cell_t$, containing input x_{t+1}, hidden h_t, and memory c_t. Then, the input data for LSTM cell $Cell_{t+1}$ takes in a new input x_{t+1} that needs to be loaded from VRAM to cache and after LSTM cell $Cell_{t+1}$ finishes, this data reloading repeats for each

LSTM cell in the sequence. Loading input x_{t+1} costs $384 \cdot 512 \cdot 2 = 393$KB. The input data that moves from VRAM to L2 cache is not a severe bottleneck if the static weight data + volatile input/output data do not exceed 5MB. The calculations above use the Bonito HAC model parameters, which show that the L2 cache does not exceed the 5MB limit. However, when a larger hidden size is used, such as in the SUP model, memory movement between VRAM and L2 cache becomes a bottleneck due to the constant VRAM reloading. Nvidia Nsight Compute reported a VRAM throughput of 80% with the SUP model, indicating that L2 cache reloading is forming a bottleneck in LSTMs with larger hidden sizes.

In summary, section 4.1 analyzed the GPU performance and behavior on Bonito's LSTM layers. The experiments show that the RTX 2080 Ti achieved only 12% of the maximum FLOP performance at batch size = 1600 and 10.6% at batch size = 512. The observed limitations will be further discussed in chapter 6, where additional improvements are suggested, and potential performance increases are estimated.

4.2. Bonito on Groq

In this section, the results of using Groq to execute Bonito are presented. First, the implementation of Bonito on Groq is explained. Then, the experimental numbers are shown. Afterward, an in-depth analysis is presented to understand the behavior of the Groq TSP and to find an explanation for the observed experimental results.

4.2.1. Groq implementation

This section discusses the performance and execution behavior of the Groq TSP in conjunction with Bonito's LSTM layers. The Groq TSP is made on a 14nm process node, with a chip area of $725mm^2$. It is rated at 188 TFLOP/s, with FP16 precision. Furthermore, it is equipped with 220 MB of on-chip SRAM and no additional caching layers.

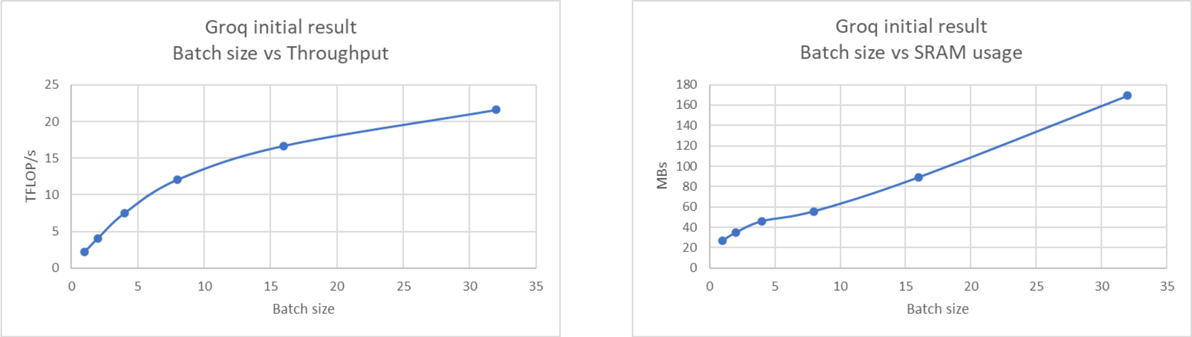
To use a Groq TSP, a Groq executable is required. Creating such a Groq executable for a neural network model is done with the Groq compiler. According to the Groq compiler manual, any valid ONNX model can be used to compile a Groq executable. Since Bonito is based on PyTorch, exporting to ONNX is straightforward, using the built-in export function in PyTorch. Afterward, the ONNX-based Bonito model is further optimized for Groq, by applying FP16 conversion and graph optimization. With that done, the Groq compiler is ready to start compiling an executable program for the Groq TSP. Within Bonito, the Groq function call is loaded in the same place as where PyTorch normally loads the neural network model into CUDA. Instead of executing inference on a GPU or CPU, a modification in Bonito was made by using Groq's TSP runner library in Python.

With the initial setup ready, converting the full Bonito model to be Groq compatible proved to be not as straightforward. Bonito uses several CUDA-specific libraries and functions, in addition to PyTorch. For Bonito to work, these functions had to be replaced with CPU-compatible versions. Additionally, due to the large model size, the Groq compiler would return “does not fit on-chip memory” errors. By tuning the 3 hyperparameters, which are the model size, batch size, and chunk size, the Groq compiler managed to get some initial numbers for Bonito on Groq. To get a better overview of what is happening under the hood of the Groq chip, Groq provided a tool called GroqView, that acts as a profiler. In the next section, these results will be further discussed.

4.2.2. Groq initial results

The Groq compiler can determine at compile time what the utilization of resources will be for a neural network model, which will be useful for later parts of the analysis. Additionally, the Groq compiler returns an overview text file, including all the statistical numbers, such as execution per second, cycle count, on-chip memory usage, and many more. The more interesting ones are the cycle count, on-chip memory usage, and the executions per second.

In Figure 4.8a and Figure 4.8b, the Bonito HAC model with different batch sizes has been compiled and the Groq compiler results show that a higher batch size improves overall performance. In these figures, the letter *h* represents the Bonito model type, the letter *b* in *1b, 2b, 4b, etc.* represents the different batch sizes, and the letter *c* in *1248c* represents the chunk size. For example, comparing batch size = 16 and batch size = 32, we see an increase of 29% in overall throughput. With larger batch sizes, overall throughput improves. However, the memory usage in Figure 4.8b shows that the Groq TSP is already approaching its limit of 220MB SRAM with batch size = 32. Furthermore, Figure 4.8a shows that larger batch sizes reach a slight plateau in the throughput curve. This indicates that increasing batch size does not increase throughput linearly. The results show that the Groq TSP reaches up to 22.4 TFLOP/s.



(a) This diagram shows the change in throughput, when using different batch sizes with Bonito HAC model, using chunk size = 1248.

(b) This diagram shows the change in SRAM usage, when using different batch sizes with Bonito HAC model, using chunk size = 1248.

Figure 4.8: Results for the HAC model with chunk size = 1248

Contrary to Groq's TSP, a GPU is not deterministic. For a comparison between the Groq TSP and a GPU, an actual Bonito execution was initially run to see how the two processing units compare. For the baseline GPU, an Nvidia RTX 2080 Ti is used. The Nvidia RTX 2080 Ti has a chip area size of $754mm^2$, is built on a 12nm process node, and is equipped with 11GB of VRAM. The RTX 2080 Ti achieves up to 26.90 TFLOP/s with FP16 precision or 104 TFLOP/s using Tensor cores with FP16 precision. The Groq TSP has a chip area size of $725mm^2$, built on a 14nm process node, and is equipped with 220MB on-chip SRAM. The Groq TSP is advertised to achieve 188 TFLOP/s with FP16.

The comparison run uses the standard Bonito HAC model and a basic POD5 dataset of around 150MB. As shown in Figure 4.8, the Groq TSP cannot handle a batch size of 512. Additionally, the chunk size is reduced to decrease the cost of storing execution instructions on SRAM. Hence, the Groq implementation uses a smaller batch size of 32 and a smaller chunk size of 1248. The GPU implementation uses the default batch size = 512 and chunk size = 10000.

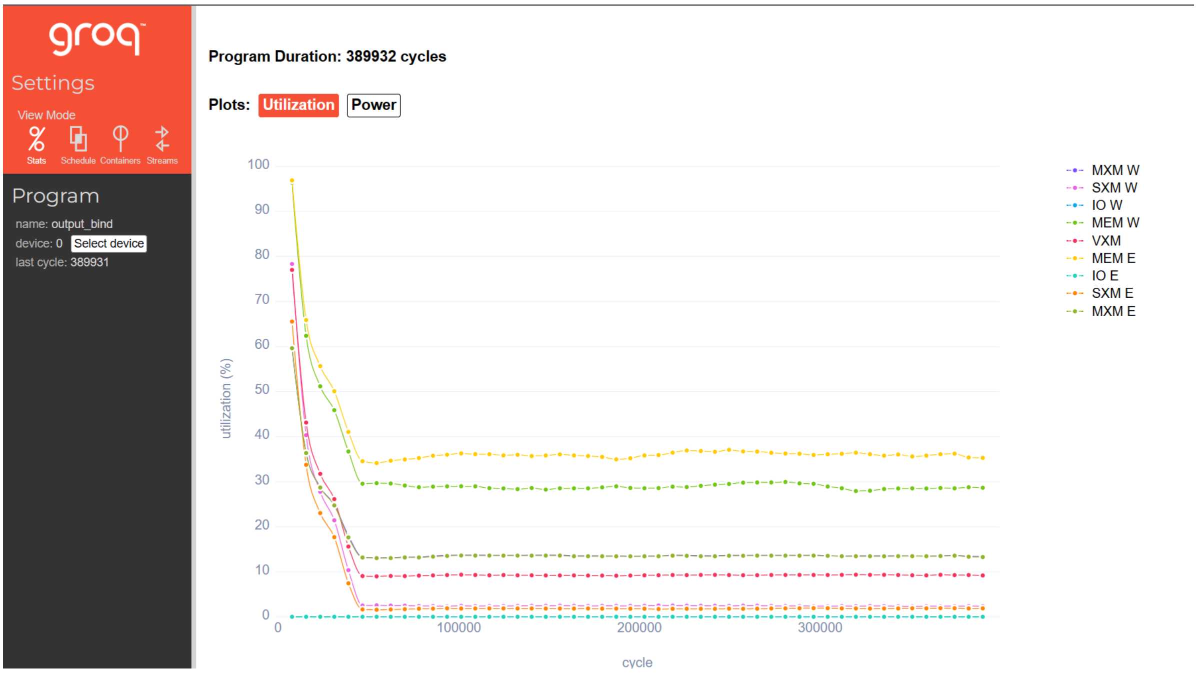
The comparison run resulted in Nvidia taking 10.53 seconds to process the dataset, whereas Groq took 14.40 seconds. These initial results showed noncompetitive numbers from Groq, compared to the baseline RTX 2080 Ti. Figure 4.8a showed that the Groq TSP should achieve 22.4 TFLOP/s, which is higher than the RTX 2080 Ti with an achieved peak of 11.5 TFLOP/s. But this was not the case for the Bonito comparison. A possible reason is that small chunks mean more data transfer instances between the host memory and the Groq TSP memory. If one 10,000 element chunk is transferred once, 8 small chunks of 1248 elements need to be transferred separately instead. That transfer time possibly plays a role in the longer Groq runtime.

Although disappointing, such results are expected to be seen, as the first iteration of the Groq implementation was not optimized. At that point in time, the knowledge, and understanding of the Groq TSP was not deep enough. Therefore, further analyses were performed, starting with the decomposition of the Bonito model into smaller pieces. Groqview is the specialized profiler made for the Groq TSP. The full Bonito model with 3 Conv and 5 LSTM layers proved too much for Groqview to handle, leading to immediate crashes. Hence, decomposing was done to enable Groqview to handle the large number of operations. The upcoming sections focus solely on the LSTM behavior and use a 1 LSTM layer setup.

Resource utilization

In Figure 4.9 and Figure 4.10, a profile is shown from 1 LSTM layer, using the same parameters as in Bonito's Fast model. Here, the LSTM has model size = 96, batch size = 1, and chunk size = 6144. From the low utilization numbers in Figure 4.9, it becomes clear that the current LSTM configuration does not utilize all available resources. In larger chunk sizes, this behavior remains, indicating that the sequential nature of LSTM limits Groq in achieving faster performance.

In Figure 4.11, the dots represent activity on the Groq TSP, where the time-axis starts from top to bottom. Initially, a dense segment of blue dots appears at the top, indicating high activity. Following the high-density segment are sparse segments. These segments indicate the LSTM execution, where



Settings

View Mode

Program

name: output_bind

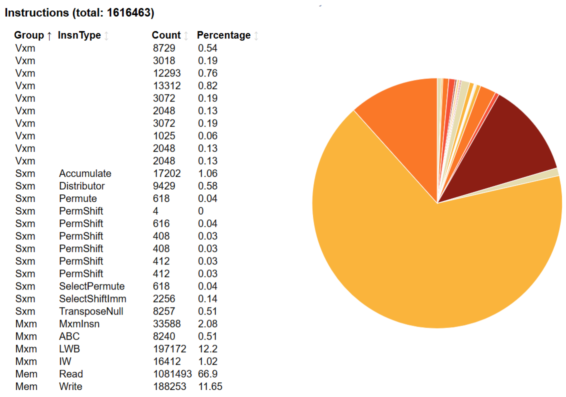
device: 0 Select device

last cycle: 389931

Program Duration: 389932 cycles

Plots: Utilization Power

Figure 4.9: Bonito Fast model 1LSTM, 6144 chunk size, 1 batch size



Instructions (total: 1616463)

Group ↑	InsnType	Count	Percentage
Vxm		8729	0.54
Vxm		3018	0.19
Vxm		12293	0.76
Vxm		13312	0.82
Vxm		3072	0.19
Vxm		2048	0.13
Vxm		3072	0.19
Vxm		1025	0.06
Vxm		2048	0.13
Vxm		2048	0.13
Sxm	Accumulate	17202	1.06
Sxm	Distributor	9429	0.58
Sxm	Permute	618	0.04
Sxm	PermShift	4	0
Sxm	PermShift	616	0.04
Sxm	PermShift	408	0.03
Sxm	PermShift	408	0.03
Sxm	PermShift	412	0.03
Sxm	PermShift	412	0.03
Sxm	SelectPermute	618	0.04
Sxm	SelectShiftImm	2256	0.14
Sxm	TransposeNull	8257	0.51
Mxm	MxmInsn	33588	2.08
Mxm	ABC	8240	0.51
Mxm	LWB	197172	12.2
Mxm	IW	16412	1.02
Mem	Read	1081493	66.9
Mem	Write	188253	11.65

Figure 4.10: Instruction distribution for 1LSTM, 6144 chunk size, 1 batch size

each spare segment represents a LSTM cell execution. In Figure 4.11, four LSTM cells are executed. This result confirms that the current LSTM implementation on Groq is not utilizing all its capabilities, as there are gaps between activities, which indicate low occupancy.

When testing out the High Accuracy model (HAC), similar low utilization behavior remains, as with the Fast model. Figure 4.12 shows an increased utilization number for all units, such as MEM, MXM or VXM, but it still does not reach close to 100%. Instead of 15% with the Fast model, the HAC model reaches an utilization of 51% for the MXM unit. With this observation, it was not clear why exactly this low utilization happened and why the HAC model showed higher utilization, compared to the Fast model. In order to find the answers, a thorough understanding of the execution behavior of Groq's TSP is required, to achieve an optimal execution.

Custom LSTM implementation

To understand the execution behavior of LSTMs on Groq, a better understanding of what exactly happens per cycle is the key to identifying problems or bottlenecks. The Groq compiler converts the existing PyTorch neural network model into an ONNX model, before the Groq compiler generates a Groq executable. Because PyTorch's default LSTM class was used initially, the ONNX export produced one wrapped LSTM block that gets interpreted by the Groq compiler as a single operation. This meant

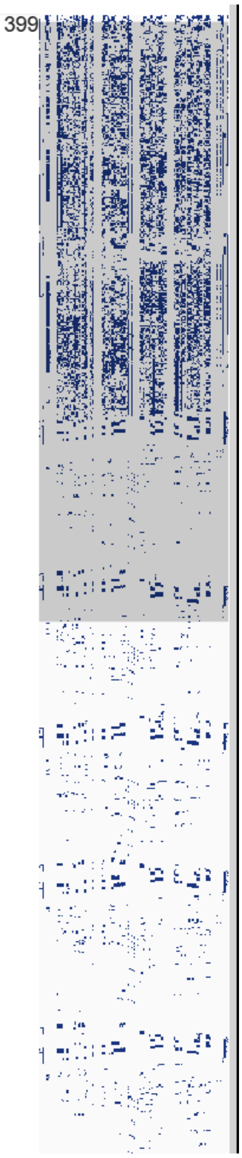


Figure 4.11: Scheduling overview with GroqView

individual operations, such as multiplications, sigmoid and such, were not visible. Therefore, a custom PyTorch LSTM class was implemented to enable ONNX to generate a graph showing each individual operation, following the equations from Figure 3. This resulted in a larger ONNX graph, but enabled the GroqViewer to identify individual operations.

4.3. Analysis of Bonito inference on Groq

From the GroqView tool, it became apparent that the MXM units are underutilized. This low utilization severely affects the LSTM performance throughput. In Figure 4.9, around 15% of the MXM unit is being utilized for the Fast model and around 51% for the HAC model in Figure 4.12. Diving deeper into why it behaves like this, we need to understand how data is processed on the Groq TSP. In this section, the physical tensor shape will first be explained, followed by the MXM architecture. Then, a per-cycle analysis is given on utilization. Lastly, INT8 quantization is discussed as a potential performance improvement. The remaining analysis focuses solely on Bonito's LSTM layers and will use the optimized tensor shape of hidden size = 320 for the test results.

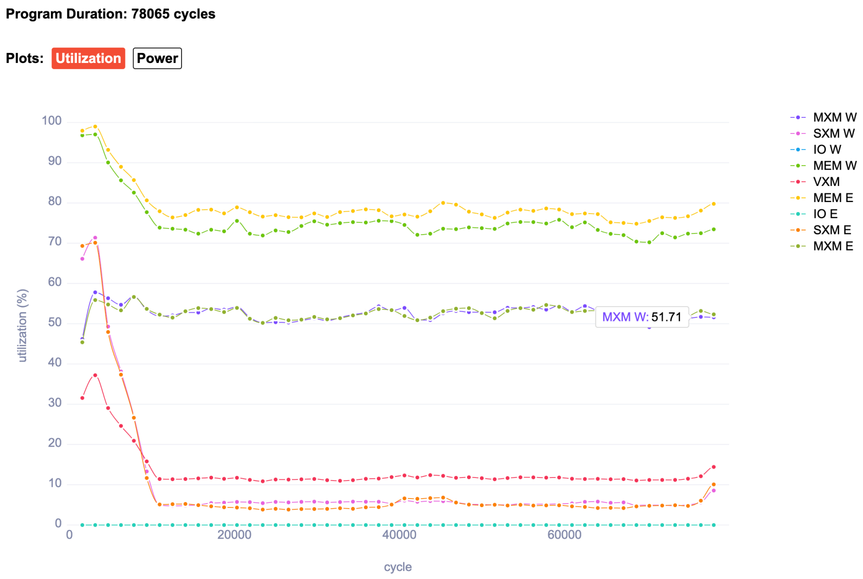

Program Duration: 78065 cycles

Figure 4.12: Bonito HAC utilization graph 1 LSTM

4.3.1. Physical tensor shape

At its core, the Groq TSP processes data and tensors with 320 element vectors. When tensors' inner dimension are larger than 320 elements, Groq uses slices to split the original ”logical shape” into smaller ”physical shapes”. In practice, it would change the example tensor in Figure 4.13 from [10, 10, 1024] into [4, 100, 4, (256, 256, 256, 256)]. Here they respectively represent the properties *splits*, *vectors*, *bytes*, and a list of inner-dimension for each split vector. A large tensor gets represented by vectorizing all dimensions, except the inner dimension. The inner dimension instead is split into $ceil(\frac{inner\_dimension}{320})$ slices. Lastly, based on the data type, *bytes* will differ from 1 byte to 4 bytes. This *bytes* represents the data type, with 1 byte being an INT8 data type, 2 bytes being an FP16 data type, and 4 bytes being an FP32 data type.

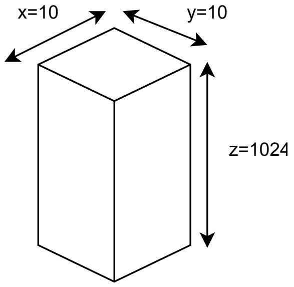
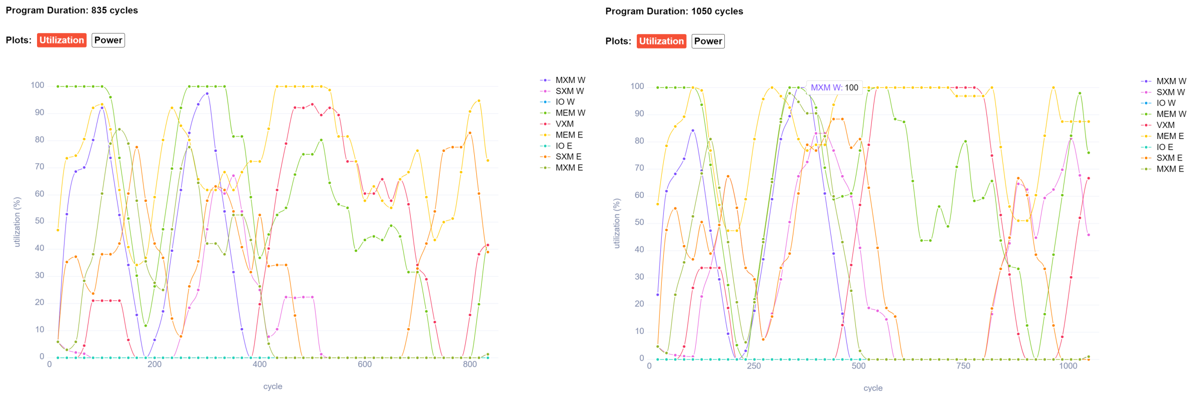


Figure 4.13: Visualization of an example tensor

4.3.2. Optimizing MXM usage

Next, the two MXM units on the Groq TSP are capable of storing up to 320x320 INT8 weights or 160x320 FP16 weights per MXM unit. Important to note, each MXM unit contains two slices, where two slices can individually store 2 times [320, 320] INT8 weights, or two slices work together to store 1 time [160, 320] FP16 weights. Knowing this, we see that with either the Fast or HAC model, the weight matrices of [96, 96] or [384, 384] do not perfectly fit the MXM module. The Fast model would only fill up utilization $U_{MXM} = \frac{96}{320} = 30\%$, and the HAC model would be split in $ceil(\frac{384}{320}) = 2$ slices and only fill $U_{MXM} = \frac{192}{320} = 60\%$. This results in weight matrices of shape [1, 96] or [1, 192] being loaded onto the MXM plane, for Fast and HAC models respectively. Since the MXM plane supports up to [160, 320] for FP16, the full potential of the MXM plane is not being used. Achieving higher plane utilization can be done in two ways. By increasing the batch size, or by adjusting the hidden size.

Adjusting the batch size allows more of the MXM plane to be used. [27] also confirm this suggestion in their comparative evaluation of novel AI accelerators. In their RNN tests, their performance with an increased batch size also yields higher performance. Similarly, our own LSTM test confirm this as well. In Figure 4.25, batch=16 and batch=32 are compared. With an increase of roughly 20% more cycles, we obtain 100% increased processing power. However, increasing the batch size does occupy more on-chip memory, which is one of the limiting factors regarding scaling batch size.



(a) Groqview batch = 16

(b) Groqview batch = 32

Figure 4.14: Comparing impact of batch size with hidden size = 320

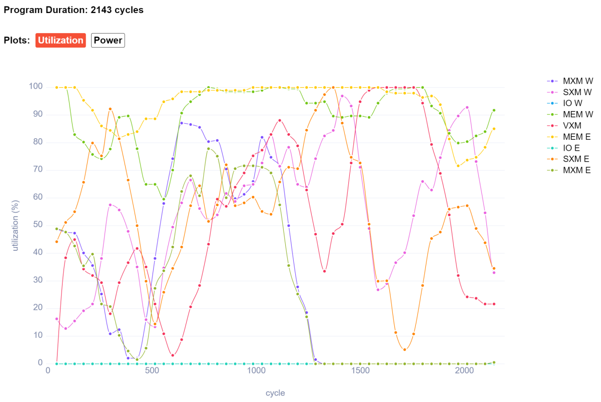
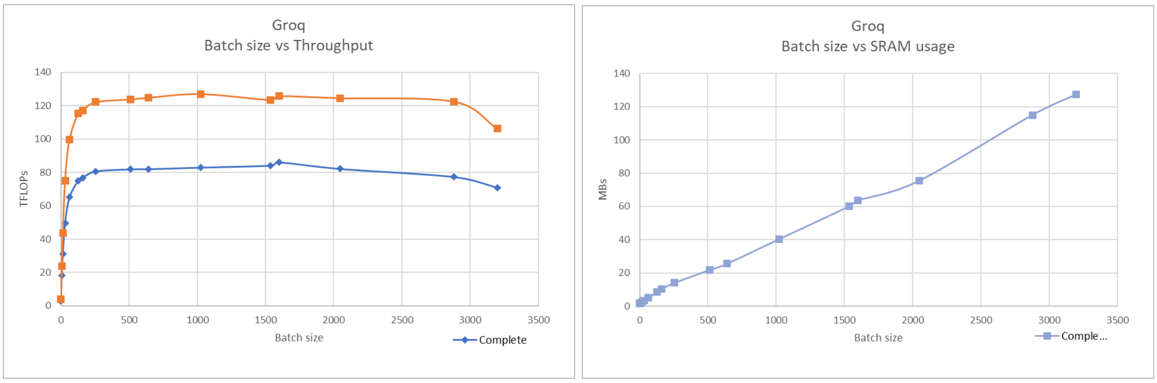


Figure 4.15: Original HAC hidden size = 384 and batch = 32

Another approach to optimize the MXM plane usage is by adjusting the hidden size, such that it occupies most of the [160, 320] available weight registers. From the previous paragraph, the Fast and HAC models only use a weight matrix of size [1, 96] or [1, 192] respectively. By increasing the hidden size to be ideally a factor of 320, the MXM plane can instead load a weight matrix [1, 320]. Thus, up to 320 features can be multiplied with the input matrix. To relate it more with formulas, $W_{i\_input}x_t$ can be done in 1 matrix multiplication, when we reduce the hidden size from 384 down to 320. Of course, the impact on accuracy could differ, but the concept of optimally aligning the tensor shape with Groq's TSP is an important factor to keep in mind. By reducing the hidden size from 384 down to 320, we achieve a significant performance boost. Comparing Figure 4.14b with Figure 4.15, the cycle count decreases from 2143 to 1050 cycles respectively, showing the importance of optimizing the tensor shape. Optimizing the tensor shape results in a new performance chart across different batch sizes, as shown in Figure 4.16a. It becomes clear how performance is lost when the LSTM hidden size does not optimally match the MXM size, based on the MXM plane size and the $ceil()$ property.



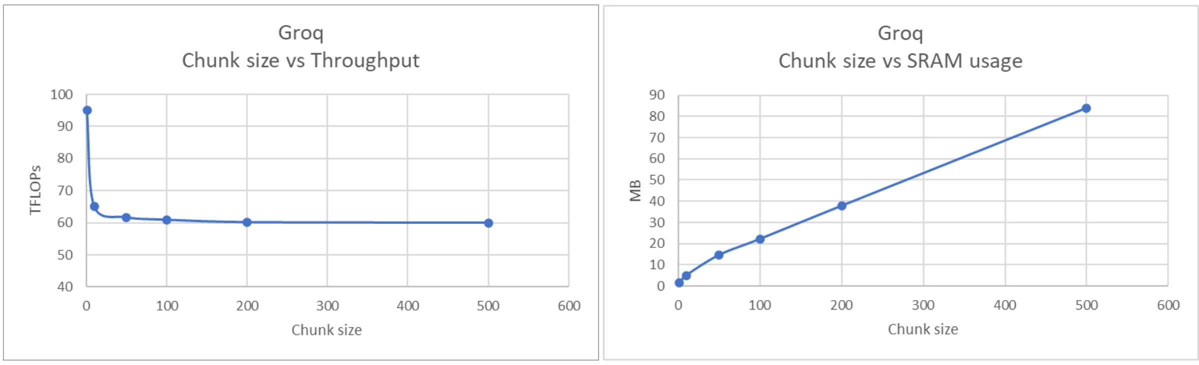
(a) This graph represents the throughput of LSTMs on Groq, with different batch sizes, using hidden size = 320 and chunk size = 10.

(b) Impact of batch size on SRAM usage with hidden size = 320 and chunk size = 10.

Figure 4.16: Comparing impact of batch size with hidden size = 320 and chunk size = 10.

MEM usage

As mentioned in the initial Groq results, the Groq compiler returned many ”does not fit on-chip” errors, where the program was too large to fit on the SRAM. In Figure 4.17, the results show that longer chunk size do not improve overall throughput. Furthermore, the impact of batch size slightly affects the overall throughput. Figure 4.16b shows the added cost of SRAM usage, whereas Figure 4.16a shows the slight increase in throughput for batch size = 1600. At a higher batch size, throughput seems to fall off.



(a) Impact of chunk size on throughput.

(b) Impact of chunk size on SRAM usage.

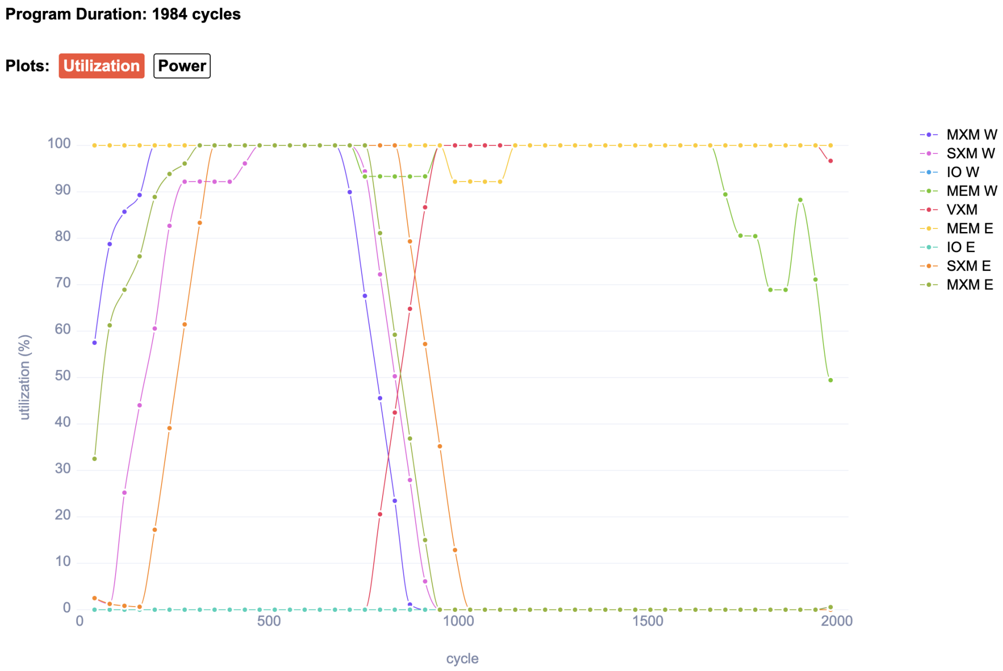
Figure 4.17: Comparing impact of chunk size with hidden size = 320 and batch size = 64.

4.3.3. 1 LSTM cell execution

In Figure 4.18, the cycle utilization is shown for a complete LSTM cell execution. In Figure 4.19, the matrix multiply-accumulate and element-wise addition is shown. Both figures use a hidden size = 320, batch size = 160, and chunk size = 1. The hidden and batch sizes are specifically chosen, such that it fits the physical tensor perfectly.

What stands out is the difference in number of cycles. As it seems, the matrix multiply-accumulate and addition costs roughly as much as the special function + element-wise computation at the later stages in an LSTM cell. It indicates that the operations Sigmoid, Tanh, addition and element-wise multiplication are relatively slow, compared to the matrix multiply-accumulate operation. This is partly because most operations are executed in the VXM module, whereas the matrix multiply-accumulate occurs in separate MXM modules. According to the Groq API manual, the VXM is rated at 20 TOPS, which converts to 10 TFLOPs. This is a significantly lower performance number than the MXM modules, which have a combined number of 188 TFLOPS of performance. When 50% of the execution cycles are spent on the small VXM unit, the available 188 TFLOPs in the MXM get utilized for only half of the time. This indicates that the VXM operations are bottlenecking the overall LSTM performance.

The next point is the change in utilization numbers with longer chunk sizes. Figure 4.20 shows a



Program Duration: 1984 cycles

Plots: Utilization Power

Figure 4.18: Cycle overview for the full LSTM cell, with batch = 160, hidden = 320 and chunk = 1.

cycle overview of the full LSTM cell, with chunk size = 10. The overall utilization seems lower, with the MXM utilization being part of the bottom wave and the VXM being part of the upper graph. Figure 4.20 confirms that the VXM module is forming a bottleneck, peaking at 80% utilization. It indicates that the VXM is much more utilized than the MXM. Additionally, with longer chunk sizes, the overall utilization seems lower. This can be explained because the LSTM cell is optimizing the pipelining process between cells $cell_t$ and $cell_{t+1}$. While $cell_t$ calculates output h_t in VXM, $cell_{t+1}$ can prepare to calculate $x_{t+1} \times W$ in MXM, since this is not dependent on the previous result. Hence, this reduces peak utilization, by starting earlier with the matrix multiply-accumulate. However, by starting earlier, the data movement for MXM LW and VXM operations occur at the same clock cycles. This means the compiler must schedule and balance the data transfers on the streaming channels accordingly.

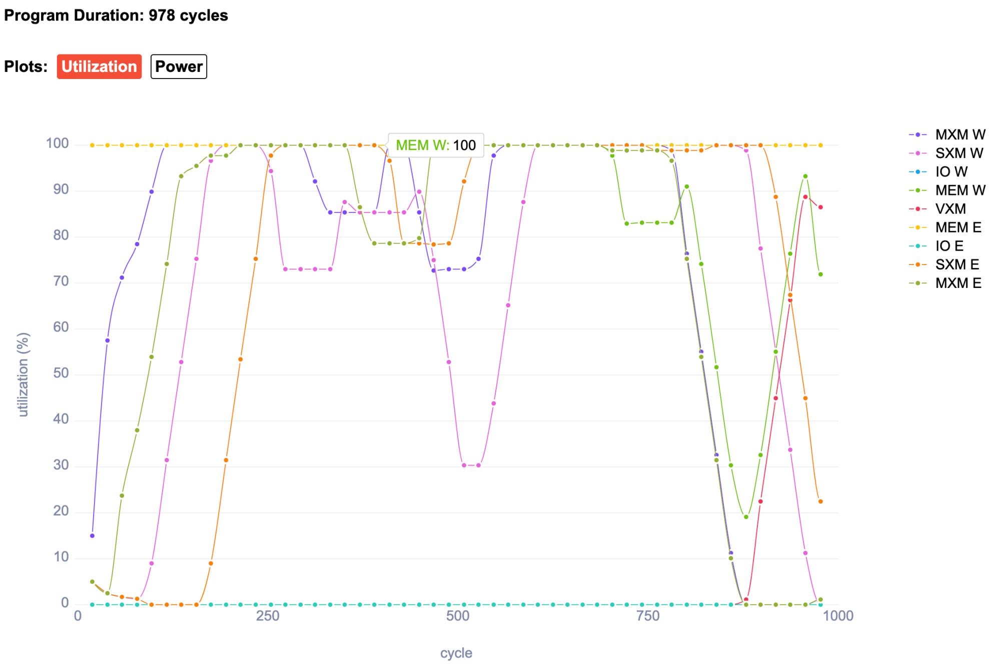
4.3.4. Per cycle analysis

After optimizing the tensor shape and attempting to improve MXM utilization, the most utilized component in the Groq chip is the MEM unit with high utilization numbers throughout the whole LSTM cycle. This section aims to decompose the cycles, in order to uncover the memory bandwidth usage.

MXM data movement

Looking from a data movement standpoint, there are in total 8 matrix multiplication per LSTM time-step, as seen from the LSTM equations in Figure 3. The previous section on optimized MXM usage showed that large input tensor and weight tensor are sliced to match the physical tensor shape and to fit the computational units. Thus, Bonito's 384-features input are sliced in two slices. This results in every LSTM cell requiring multiple MXM Load Weight (LW) operations to be performed, to process a large weight matrix multiplication. As noted before, performing MXM LW costs 40 cycles and accumulating costs roughly 40 cycles [28]. Reflecting to Figure 4.18, an input matrix of A[160, 320] is multiplied by weight matrix of W [320, 320]. This multiplication happens in total 8 times per LSTM cell. Because the MXM weight buffer supports 320 bytes by 320 bytes, using FP16 precision as weights means the MXM weight buffer can fit only 160 by 320 elements. Thus, each weight multiplication requires 2 slices, creating in total 16 smaller multiplications and MXM LW operations. Furthermore, there are two MXM on each hemisphere, meaning two multiplications can occur in parallel.

Hence, performing all 8 matrix multiplications will take $80 \cdot \frac{16}{2} = 640$ cycles. Accumulating the two results from each sliced weight matrix W is done in the SXM. Accumulating is done by loading



Program Duration: 978 cycles

Figure 4.19: Cycle overview for only the matrix multply-accumulate and addition, with batch = 160, hidden = 320 and chunk = 1.

part *a* and part *b* into the SXM module, requiring ideally less than 4 clock cycles. This is based on how far the partial result in MEM is stored from the SXM. In between each complete multiplication, the accumulated result from SXM needs to be written back to memory again, before being used in the VXM addition operation. This will take around 24 cycles, which is the cycles required for data to travel from MXM to VXM in the worst-case distance. Figure 4.21

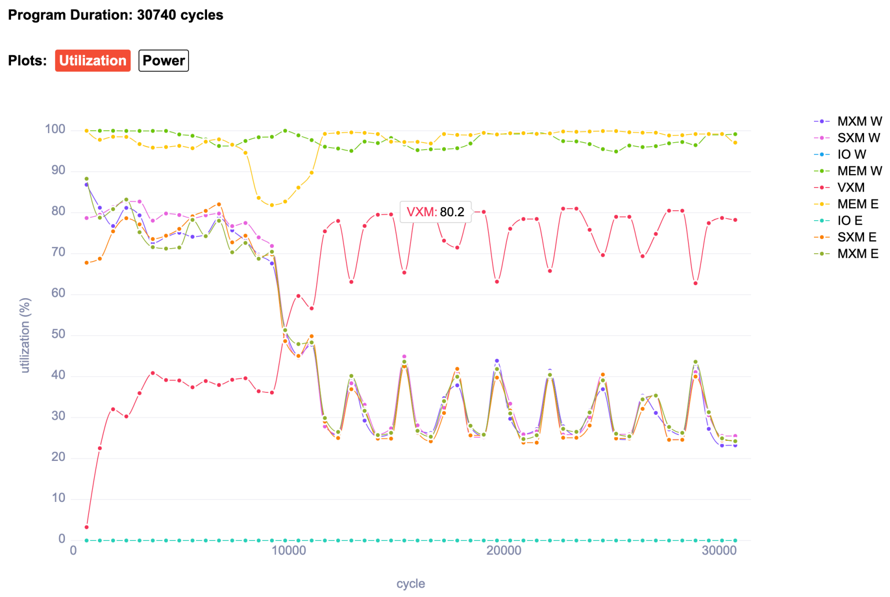
Streaming channels

Groq states that the streaming channels combined have a bandwidth of 20 TiB/s [22]. Furthermore, there are in total 32 eastwards channels and 32 westwards channels. Hence, every direction can support 10 TiB/s of combined input and output to SRAM, or 5 TiB/s in the eastwards direction, feeding data to MXM, as an example. This means that per cycle, 5KB can be transferred to MXM in each direction. One MXM LW operation would transfer $320 \cdot 160 \cdot 2 = 102$KB. This would require around 21 cycles, when fully utilizing the streaming channel in 1 direction and assuming that the weights are stored in SRAM directly next to the MXM module. Streaming matrix A is illustrated in Figure 4.22, showing how data moves per clock cycle. Worst case, data travels from far, which can pass through up to 44 streaming registers on the streaming channel, as shown in Figure 4.21. Thus, the constant reloading of the MXM weight buffer strains the streaming channel and increases the MEM utilization. In other words, because 1 larger matrix multiplication needs several MXM LW operations, it breaks the streaming-like behavior of multiplying $W \times A$ and causes stalls, where these MXM LW operations occupy the streaming channels.

Groq's architecture is optimized for when weights W do not need to be reloaded and activation matrix A is ”large enough” with size [N, 320] to enable a continuous result accumulation. By enablling streaming activation matrix A to stream for more cycles, the MXM can perform multiplications for more cycles without MXM LW interruptions, and keep accumulating the results to achieve a better pipeline. In essence, the ideal Groq program would use a low number of MXM Load Weight operations and a large Activation matrix A, such that the MXM remains constantly performing multiplication cycles.

VXM operations

The results in Figure 4.18 showed that operations in the VXM account for 50% of the total cycles in an LSTM cell. To understand what happens exactly, we analyze the cycle cost per type of operation. Inside the VXM are 5120 ALUs present. These ALUs are responsible for the addition, element-wise



Program Duration: 30740 cycles

Figure 4.20: Cycle overview for the full LSTM cell, with batch = 160, hidden = 320 and chunk = 10.

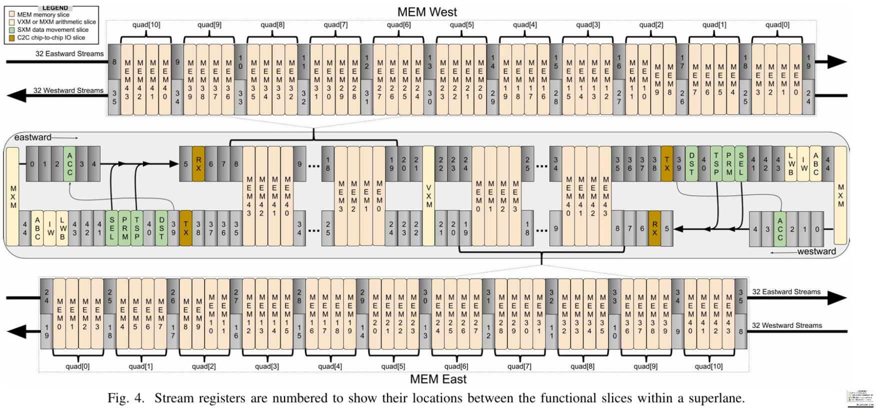


Fig. 4. Stream registers are numbered to show their locations between the functional slices within a superlane.

Figure 4.21: Stream registers are numbered to show their locations between the functional slices [22]

multiplication, Sigmoid, and Tanh. According to the Groq API programming guide, a Tanh costs 1 ALU and 1 cycle per point-wise computation, and a Sigmoid costs 2 ALU and 1 cycle per point-wise computation. The addition and element-wise multiplication also cost 1 ALU and 1 cycle per point-wise computation.

$$f_t => kn + 2kn = 3kn \tag{4.1}$$

$$i_t => 3kn \tag{4.2}$$

$$g_t => 2kn \tag{4.3}$$

$$o_t => 3kn \tag{4.4}$$

$$c_t => 3kn \tag{4.5}$$

$$h_t => 2kn \tag{4.6}$$

Figure 4.23: FLOP cost per formula, with the matrix multiplication removed

Figure 4.23 shows the element-wise operations required in the LSTM with the matrix multiplication removed in the formula, because this paragraph focuses on the computations occurring in the VXM. Furthermore, f_t, i_t, o_t have an additional kn elements, because the Sigmoid activation requires 2 ALUs,

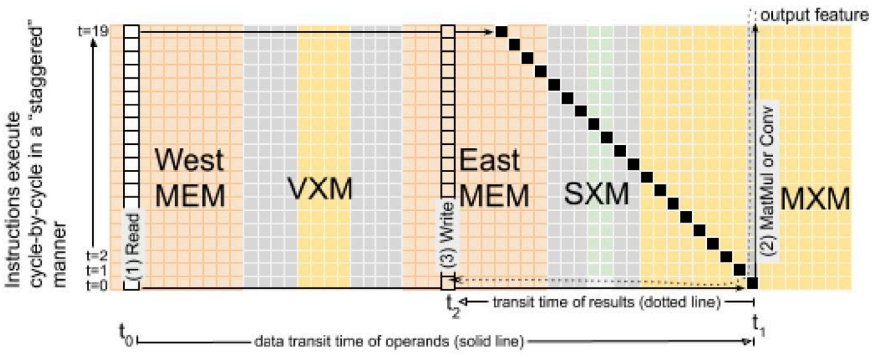

output feature
Instructions execute cycle-by-cycle in a "staggered" manner
t=19
t=2
t=1
t=0
(1) Read
West MEM
VXM
(3) Write
East MEM
SXM
(2) MatMul or Conv
MXM
t_2
transit time of results (dotted line)
t_0
data transit time of operands (solid line)
t_1

Figure 4.22: Illustration of how data moves in steps across the streaming channels [22].

instead of 1. Hence, for an input size of [160, 320], the cycle cost for each operation is shown in Table 4.1.

	Addition	Element-wise Mult	Sigmoid	Tanh
Cycles	10	10	20	10

Table 4.1: Cycle costs per operation in VXM, based on an input matrix [160, 320].

Thus, having 5 additions, 3 Sigmoids, 2 Tanh and 3 element-wise multiplications will cost in total $5 \cdot 10 + 3 \cdot 20 + 2 \cdot 10 + 3 \cdot 10 = 160$ cycles, excluding memory transfers between calculations. Figure 4.24 is a perfect example of how data needs to travel for many cycles in certain cases, showcasing that the result in MEM needs additional cycles to travel back to the VXM in the future. Hence, this this all together leads to a high cycle cost for the operations in VXM.

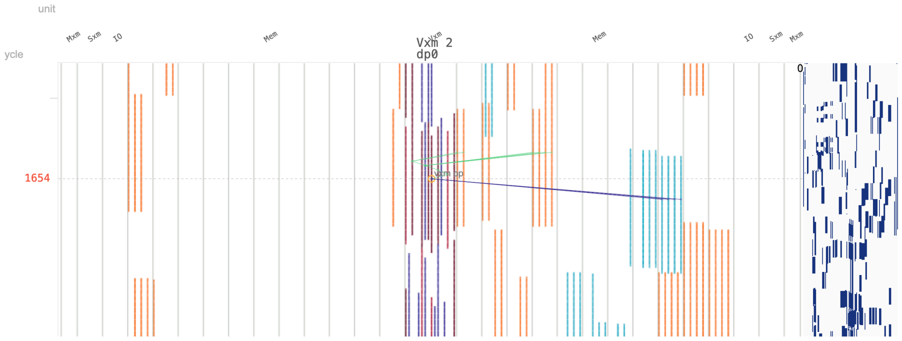


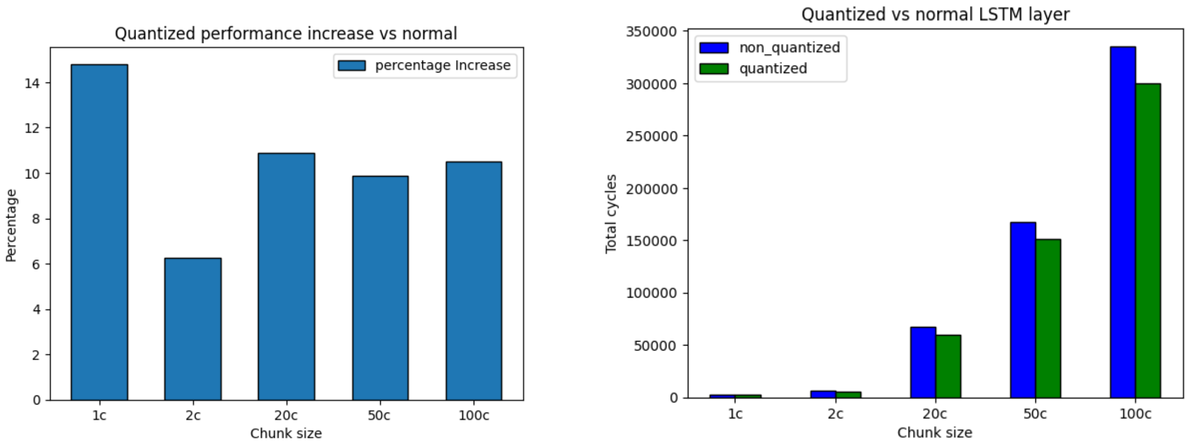
Figure 4.24: This diagram shows the data movement between each VXM operation

4.3.5. INT8 quantization

One of the common optimization strategies is to reduce the precision of the model. Prior tests in this section are run on FP16 precision. By running on FP16, the maximum FLOP/s gives 188 TFLOP/. However, running on INT8, Groq claims performance up to 750 TOP/s. This indicates that Groq's architecture is more optimized for INT8, than FP16. Using the knowledge from the MXM plane architecture, INT8 allows for [320, 320] weights and using four slices, instead of [160, 320] weights and two slices for FP16. Hence, allowing four times as many weights to be processed per cycle. However, there are challenges with quantizing a model. First, precision is lost, for which the model could lose accuracy. Second, Groq does not support all quantized PyTorch and ONNX functions. Unfortunately, the LSTM class is not natively quantizable.

For the first challenge, Oxford Nanopore Technologies also pursued INT8 quantization and obtained positive results. Their flagship basecaller Dorado in C++ makes use of INT8 datatypes [29]. This means quantization should yield an acceptable loss of accuracy, for increased performance. However, their development on INT8 quantization is only implemented in their C++ basecaller. There is no method to convert Dorado directly into ONNX, for the Groq compiler to process further. Instead, Groq proposed to use ONNX quantization methods to quantize the model. Unfortunately, ONNX does not support ”static quantization” for LSTM. The other method, called ”dynamic quantization”, is not supported by the Groq compiler, leaving us in a bind.

After creating the custom LSTM implementation, the Groq compiler managed to quantize the LSTM model and show a performance increase. This is made possible due to the ONNX graph that generates individual node operations for the custom LSTM, such as MatMul, Tanh and Add. However, this quantization method is not natively supported by Groq.
In Figure 4.25a and Figure 4.25b, a comparison is given between the normal and quantized LSTM model. From these bar charts, the improvements average around 10%. The jump from FP16 to INT8 would ideally result in a 300% in performance, while the current improvement only yields 10%. This means that the quantization strategy is currently limited by software and not limited by architectural design choices.



(a) Percentage increase per chunk size

(b) Quantization impact on different chunk sizes

Figure 4.25: Comparing impact of batch size with hidden size = 320

4.4. Overview of analysis

This chapter has covered the analysis of LSTM performance and behavior on a GPU and a Groq TSP, in the context of Bonito’s HAC model at FP16 precision. The used accelerators are presented in Table 4.2

	Process node	Chip area	Peak TFLOP/s
Nvidia RTX 2080 Ti	12nm	$754mm^2$	107.60
Groq TSP	14nm	$725mm^2$	188

Table 4.2: Overview of the different accelerators properties and their corresponding compute performance.

With the Bonito HAC model with hidden size = 384, an initial basecalling comparison between the GPU and Groq was performed, with results shown in Table 4.3. The runtime is based on the complete Bonito HAC model processing a 150MB test dataset, while the achieved TFLOP/s are based on pure LSTM layer performance. The GPU achieved 10.5% of the peak 107.6 TFLOP/s, whereas the Groq TSP achieved 11.7% of the peak 188 TFLOP/s. The Groq TSP achieved a higher peak TFLOP/s than the GPU, in context of LSTM performance. However, in terms of runtime, the GPU finished the test dataset in 10.53 seconds and Groq finished in 14.40 seconds. The possible reason for the slow Groq runtime was unoptimized data transfers between host memory and Groq SRAM.

For the GPU, we saw that the main limiting factor is the low compute utilization due to Bonito’s small problem size. There are too many cores available, while Bonito can only utilize part of the available cores.

For the Groq TSP, we saw a mismatch between Bonito’s input tensor size and Groq’s physical tensor

	Achieved TFLOP/s	Peak TFLOP/s	% TFLOP/s	Runtime (s)
Nvidia RTX 2080 Ti	11.5	107.6	10.6	10.53
Groq TSP	22.4	188	11.8	14.40

Table 4.3: Results of comparing GPU to Groq, when running Bonito HAC model.

shape. Furthermore, the in-depth cycle analysis showed how the Groq TSP is limited by the VXM and the corresponding element-wise operations present in LSTMs.

By optimizing Bonito for Groq's physical tensor shape, an experiment with batch size = 1600 and hidden size = 320 was done. Table 4.4 shows that this change increased the Groq achieved TFLOP/s from 22.4 to 86.4 TFLOP/s for LSTM layers. This shows that Groq has potential to surpass GPU performance with Bonito. However, to test the hidden size = 320 with Bonito's full model, it is required to reprogram the Bonito program in such a way, that the weights contain 320 hidden features instead of 384 hidden features. However, this endeavor was deemed outside the scope of this thesis. In chapter 6, further discussion on the limitations and potential improvements is presented, with more details on the estimated performance gains.

	Achieved TFLOP/s	Peak TFLOP/s	% TFLOP/s
Nvidia RTX 2080 Ti	11.5	108	10.6%
Groq TSP (original HAC model)	22.4	188	11.8
Groq TSP (Optimized batch and tensor shape)	86.4	188	45.9%

Table 4.4: Overview of the different accelerators properties and their corresponding performance on Bonito's LSTM layers.

5

ASIC estimations

The analysis on existing accelerator architectures revealed new insights on LSTM performance and what design choices affect overall throughput. Therefore, the next step is to investigate whether a custom Application-Specific Integrated Circuit (ASIC) can prove viable for LSTMs, while competing with existing accelerators. In this section, the workflow of designing and researching an ASIC implementation is discussed. Then, the synthesis step is further elaborated upon, with the area and performance estimations presented. Afterward, an analysis on mapping the synthesis design to an existing Groq chip layout is performed to accurately estimate the final ASIC design performance.

5.1. ASIC workflow

In terms of ASIC design workflow, there are 11 steps in total; from chip specification on paper to tape-out where a foundry will produce the actual silicon chips in ASICs. Figure 5.1, shows an overview that describes the required steps to take. For this thesis, steps 1 and 2 have been completed by Jasper Haenen [30], a student at TU Delft. In these steps, he has designed the baseline architecture for an LSTM Processing Engine (PE) in step 1 and verified its functionality in step 2. For the verification, both a test bench and the Alveo U280 FPGA (Field-Programmable Gate Array) were used to verify its behavior. Unlike an FPGA, an ASIC implements the logic behavior directly onto the chip. Changes in the chip design are impossible after the foundry has produced the chip. Therefore, it is important to guarantee the functional correctness during the design process by following the ASIC workflow.

For this thesis, the focus is on the RTL synthesis step, mapping the RTL (Register-Transfer Level) code to an ASIC. Performing an ASIC design requires highly specialized and NDA-protected programs. The Synopsys toolchain and TSMC Process Design Kit (PDK) were fortunately available through the TU Delft software portal. For this purpose, the Synopsys Design Compiler program is used. Together with the TSMC 40nm PDK, Design Compiler synthesizes a netlist with gates and routes to create a gate-level logic circuit using the standard cells from the PDK.

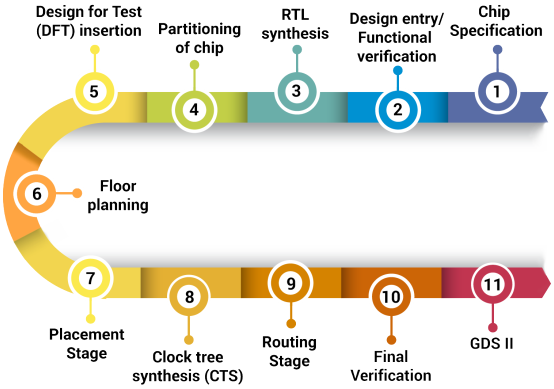


Figure 5.1: Diagram showing the design steps involved in an ASIC design [31]

5.2. FPGA architecture

The LSTM architecture design made by Haenen [30] uses Processing Engines (PEs) to process 1 FP16 input per PE. In Figure 5.2, the architecture inside each PE closely follows the diagram of an LSTM cell, as shown in Figure 3.2. In terms of hardware, the design consists of Multiply-Accumulate components (MACs), adder components and default multipliers. Furthermore, the Sigmoid and Tanh components use a point-wise linear approximation instead of calculating with exponentiation. This approximation is faster and requires less area cost [30].

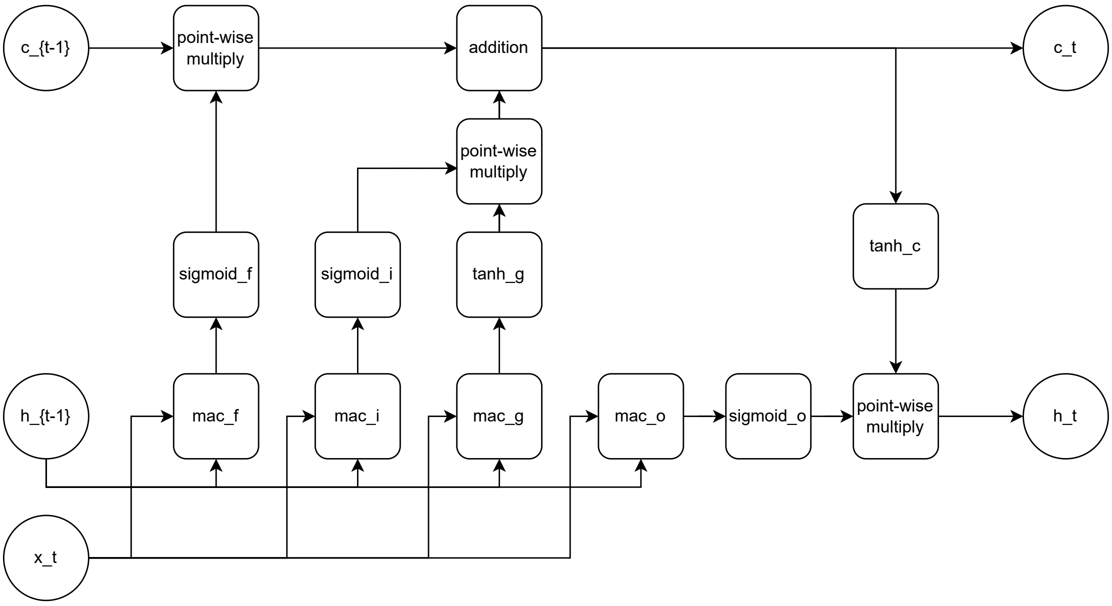


Figure 5.2: Overview on architecture components used in 1 PE.

The overall architecture follows a 384-cycle matrix multiply-accumulate stage, followed by a 9-cycle post-processing stage. Important to note, the matrix multiply-accumulate stage can vary in required cycles, depending on the matrix size of the input tensor. In total, a full LSTM cell for Bonito's HAC model would require 393 cycles to be processed. Figure 5.3 shows the different stages in the PE.

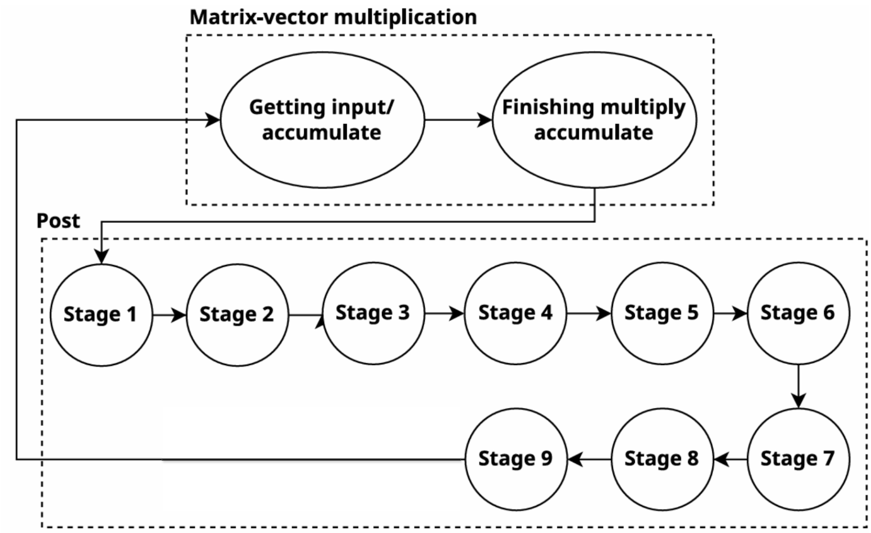

Matrix-vector multiplication
Getting input/ accumulate
Finishing multiply accumulate
Post
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Stage 6
Stage 7
Stage 8
Stage 9

Figure 5.3: Stages of execution in the Process Engine

Scaling of the LSTM design is done by increasing the number of PEs on the final design. For instance, placing 384 PEs would allow Bonito's HAC model to process 384 inputs in parallel per clock cycle. Further scaling, based on batch size or for the SUP model, is possible with additional PE placements. However, to handle larger batch sizes, it requires that blocks of 384 PEs are not interconnected with each other. Thus, with a batch size of 2, two blocks of 384 PEs are introduced, such that the matrix accumulates only accumulates for the 384-sized input vector per batch. Similarly to GPUs, the FPGA and ASIC design can have many PE blocks to enable more parallel performance. However, this will cost more area, more cache, and more memory bandwidth. Thus, a balance must be found between PE blocks and resource allocations.

Contrary to the Groq heterogeneous architecture, this custom LSTM architecture uses a similar approach to GPUs, with each PE responsible for a small task. The more individual PEs on the design, the more reads it can process in parallel. Groq's design is rather focused on processing larger chunks of data and placing it in a pipeline in between the separate VXM and MXM compute blocks. Another difference is that a PE block executes the LSTM cell in 393 clock cycles, with the majority of executions spent on matrix multiply-accumulate. Conceptually, reducing the clock cycles required to process 1 LSTM cell results in a large performance improvement. For instance, Groq's MXM multiplier would take roughly 20 cycles to compute a matrix multiplication of size [160, 320]. Adding the VXM unit for addition and special functions, Groq would require at least 25 cycles to process 1 LSTM cell, not accounting for data transfer times. However, Groq requires the input tensor to be sliced into smaller physical tensors. This means an ASIC matrix multiply-accumulate is performed in 1 stage of 393 cycles, while on Groq, multiple smaller MXM operations are performed. Compared to the GPU, the GPU requires multiple clock cycles as well, as the majority of hardware is allocated to matrix calculations and a separate special function unit is required to process the remaining Sigmoid and Tanh calculations. However, the GPU requires substantially more clock cycles than the ASIC or Groq design. In return, the GPU has the ability to process a large batch size.

The ASIC design is optimized to execute one LSTM cell computation in 393 clock cycles. In contrast, the Groq or GPU accelerator requires roughly 2000 or 103,027 cycles respectively, to compute one LSTM cell. However, on an architectural level, the ASIC design is custom made for only LSTM applications. This wastes no area on other unused compute logic normally present on more general-purpose accelerators, such as Groq or GPUs.

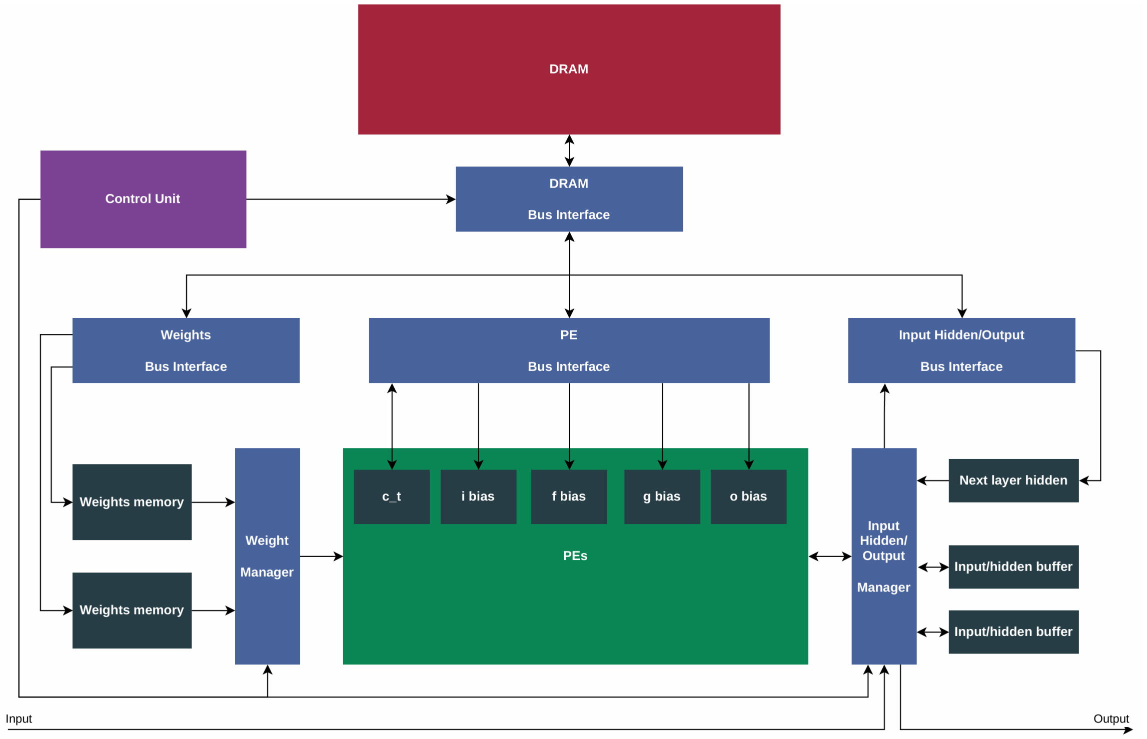


Figure 5.4: Final FPGA architecture diagram including data control [30]

Lastly, due to time constraints, the design used for the ASIC estimation does not take into account cache and memory control. Haenen did theorize about the memory control architecture, as shown in Figure 5.4. However, this portion of the architecture was not implemented in this thesis. Furthermore, the ASIC design is based on a fixed-point implementation, due to Xilinx-locked IP components used in the floating-point design and its tight integration with Xilinx's IP control blocks. Hence, an analysis is presented in the next sections, estimating the fixed-point to floating-point difference and correcting for those in the results. Therefore, this ASIC design section covers only the compute units and their performance.

5.3. Performance evaluation

5.3.1. ASIC synthesis estimations

Using Synopsys Design Compiler, a synthesis implementation was created, based on the same FPGA design used in the FPGA synthesis by Haenen [30]. In Table 5.1, the results are shown from the Design Compiler tool. The ASIC synthesis achieved an area cost of 4.58mm^2 and a clock speed of 500MHz, using a 40nm process node. This result led to the question of why the ASIC clock speed is so low. To answer this question, a component analysis is performed to investigate the critical path.

	384 PE	2 PE
Clock Period (ns)	2.0	2.0
Clock Frequency (MHz)	500	500
Area (μm^2)	4582473	24242

Table 5.1: Synthesis result of the LSTM design, containing 384 PEs

	DW_add	DW_mult	Sigmoid	Remaining logic
Clock Period (ns)	0.09	0.85	0.58	0.19
Area (μm^2)	47.9808	1218.3948	371.8512	undefined

Table 5.2: Critical path breakdown for the ASIC implementation

First, the timing analysis returned a critical path from sigmoid_i to c_t_reg. In Figure 5.2, the path from sigmoid_i to c_t contains a point-wise multiply and an addition. This indicates that the current fixed_point design does not use intermediate buffers to create a pipeline. Table 5.2 shows the components and its corresponding latency in the critical path. The first observation shows that the default multiply unit DW_mult is the slowest in the critical path, followed by the Sigmoid operation. Likewise, the Tanh operation has a similar latency as the Sigmoid. One way to solve this is by introducing a buffer inside the Sigmoid and Tanh, such that the critical path is split into two stages. In doing so, the design runs at 38% higher clock speed, at the cost of 12% increased area. In Table 5.3, the synthesis result is shown for only 2 PEs, as the synthesis process for 384 PEs is very time-consuming. The critical path will be the same for both the 2 PE and 384 PE synthesis, while the area estimates error is less than 1% for the full 384 PEs synthesis.

	2 PE
Clock Period (ns)	1.45
Clock Frequency (MHz)	689
Area (μm^2)	27162

Table 5.3: Synthesis result when using a buffer-equipped Sigmoid and Tanh

This led to a new critical path, inside the MAC component. Currently, the MAC is using a default multiply and adder path. Synopsys has an IP MAC component, which is optimized for latency. By replacing the generic multiply-accumulate logic with the IP component, the new clock frequency is 74% faster than the baseline of 500MHz, at the cost of 20% more area, as shown in Table 5.4. Table 5.5 shows the area cost for each component inside a PE. Here, the MAC is the largest contributor in area cost. The remaining critical path is the stage where sigmoid_o and tanh_c are multiplied together, and resized to 16 bits. However, improving this is not pursued further due to time constraints. Hence, the final ASIC design uses the synthesis results from Table 5.4 for the future sections.

	2 PE	384 PE
Clock Period (ns)	1.15	1.15
Clock Frequency (MHz)	869	869
Area (μm^2)	29088	5379134

Table 5.4: Synthesis result when using a buffer-equipped Sigmoid and Tanh, and replace the MAC with an Synopsys IP component

	MAC	Add	Sigmoid	Tanh
Area (μm^2)	1731	280	485	690

Table 5.5: Area cost per component in one PE.

5.3.2. Final design

With these optimizations implemented to reduce the critical path, the line was drawn here to mark the final design used in the comparison between the ASIC design, and existing GPUs and Groq accelerators. By implementing these buffers, the current design uses 7 stages, instead of the 9 stages in the floating-point design by Haenen [30]. The difference in stages is due to the removal of the fused multiply-add stage inside the special functions Sigmoid and Tanh. The Sigmoid directly calculates the result in 1 stage, instead of two stages as proposed by Haenen. The Tanh uses a LUT to find the result in 1 cycle instead of 2 cycles. In Figure 5.5, the current sequence of operations is presented.

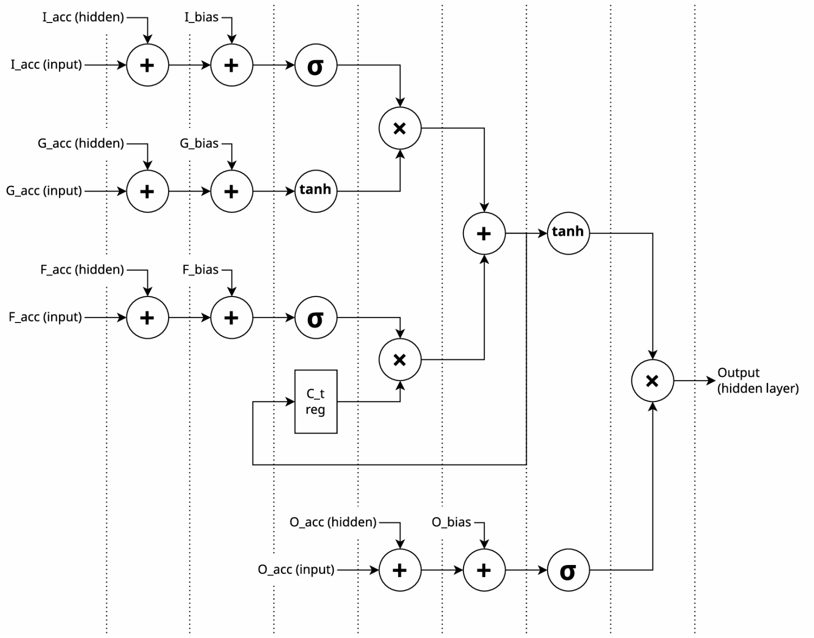


Figure 5.5: Timeline of the 7-staged sequence performed in the PE [30]

With a final design rated at 869MHz and an area cost of approximately $5.38mm^2$ per 1 block of 384 PEs, it shows that the final design is an improvement compared to the initial synthesis result. The next section will discuss the conversion from fixed-point to floating-point, such that the comparison between the ASIC design and existing floating-point accelerators is done on the same precision.

5.3.3. Fixed-point to floating-point comparison

Due to the IP-locked components in Xilinx Vivado, the floating-point design of Haenen [30] could not be utilized in the ASIC estimation. Hence, this thesis shifted its focus to the fixed-point design. However, comparing a fixed-point implementation to a floating-point baseline in GPUs or Groq is an unfair comparison, due to the added hardware cost in handling floating-point. Hence, this section covers an analysis on the difference between fixed-point and floating-point in ASICs.

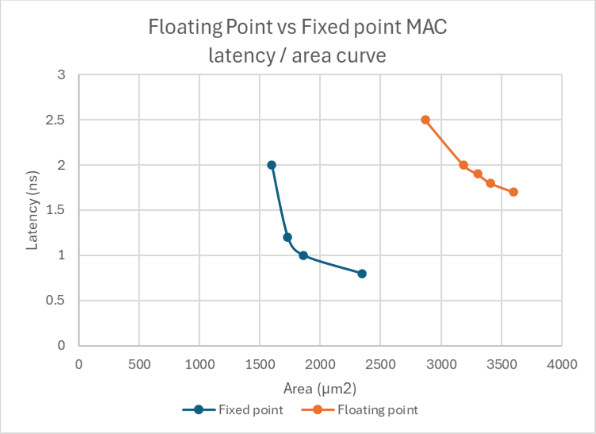


Figure 5.6: Fixed-point vs floating-point: area / latency curve for MAC component

The comparison was made using two Synopsys IP MAC blocks. One designed for fixed-point and the other designed for floating-point. From Figure 5.6, it is clear that the fixed-point MAC component is both faster and costs less area than the floating-point MAC component. However, the area/latency curve for the fixed-point shows that the fixed-point becomes much less efficient when optimizing for latency. Compared to the floating-point MAC component, the fixed-point curve follows an inverse exponential curve, compared to the less curving floating-point graph.

When optimizing for latency, the fixed-point is x2 faster and is x1.5 more area efficient. Hence, for a specific fixed-point design, conversion of the synthesis results should be done by making the fixed-point design x2 slower and increasing the area cost by x1.5. From the ASIC synthesis results, we saw that a fixed-point design of 384 PEs costs $5.38mm^2$ and had a latency of 1.15ns. To convert to a floating-point design, this same block of 384 PEs costs $8.07mm^2$ and has a latency of 2.3ns, or a clock speed of 434 MHz, as shown in Table 5.6.

	2 PE	384 PE
Clock Period (ns)	2.3	2.3
Clock Frequency (MHz)	434	434
Area (μm^2)	43632	8068701

Table 5.6: Area cost and latency estimates for fixed-to-floating point design.

5.3.4. Converting the ASIC design to Groq chip layout

While the synthesis results only returned estimates of the compute performance, it does not give a direct number that is comparable to the Groq or GPU design in terms of area and performance. One way to create a fair comparison is by theorizing what the performance would be if we were to replace the current compute units on the Groq TSP by the ASIC PEs. In this way, we can estimate how much more efficient the ASIC design is using the existing Groq chip die configuration.

Groq has released a die photo in Figure 2.10, which enables us to make an estimation on the area occupied by the compute units VXM and MXM. Based on the picture, the compute units VXM and MXM are in total 324 by 756 pixels, giving a total area of 244944 pixels. The entire die photo is 799 by 902 pixels, giving a die photo area of 720698 pixels. Thus, the percentage area occupied by the computing units is $\frac{244944}{720698} \cdot 100 = 34\%$. Next, Groq announced that their die is $725mm^2$ [28]. Thus, the compute units occupy $725 \cdot 0.34 = 246mm^2$ of the die.

With the available compute area calculated, an estimation can be made on how the PEs will perform on the Groq chip layout. The Groq compute area can fit $\frac{246}{8.07} = 30$ blocks of 384 PEs. In theory, placing 30 blocks would enable the PE-equipped Groq chip to process a batch size of 30 at a clock frequency of 434MHz. With this clock frequency and 30 blocks of 384 PEs, the theoretical peak performance can be estimated as follows. First, each LSTM cell execution requires 391 cycles. Executing 1 LSTM cycle at hidden = 384 and batch = 30 costs $16 \cdot 384^2 \cdot 30 + 5 \cdot 384 \cdot 30 = 70.83$ MFLOP. Running at 434MHz results in $\frac{434 \cdot 10^6}{391} = 1109974$ LSTM cell executions per second. This gives a peak performance of $1109974 \cdot 71.52 \cdot 10^6 = 7.86 \cdot 10^{13}$ FLOP per second, or 78.6 TFLOP/s.

The synthesis step did not take SRAM control into consideration due to time constraints. However, a quick analysis of the required memory bandwidth can be valuable in a later stage to determine the SRAM requirements.
Supplementing the peak performance of 78.6 TFLOPs requires a memory bandwidth that can supplement 4x 16-bit weight input, 1x 16-bit data input per cycle per PE, and 2x 16-bit output that is only transferring data once every 391 cycles.

The inputs generate $384 \cdot 2 \cdot 5 = 3840$ Bytes per cycle for 1 block of 384 PE, or $3840 \cdot 30 = 115200$ Bytes per cycle for 30 blocks of 384 PEs. Converting this gives a required memory bandwidth of $115200 \cdot 434 \cdot 10^6 = 4.999 \cdot 10^{13}$ Bytes/s or 50 TB/s. Per 1 block of 384 PEs, it would require a bandwidth of $3840 \cdot 434 \cdot 10^6 = 1.666 \cdot 10^{13}$ Bytes/s or 1.6 TB/s

Process node correction

On a 40nm process node, the achieved peak performance is rated at 78.6 TFLOP/s. Groq is produced on a 14nm process node, which means that more transistors can fit on the same silicon area. This leads to a direct increase in peak performance, as more blocks of 384 PEs can be placed. Furthermore, the clock speed can be increased due to smaller transistors.

TSMC has shown transistor density across different nodes for each generation. For the 14nm process node, the density is estimated at 33.8M TR/mm^2. However, published numbers on transistor density date back to only the 16nm process node by TSMC and is estimated to have a transistor density of 28.88 MTr/mm^2 [32]. Despite the lack of published numbers for the 40nm process node, an estimation can be made. Rieger [33] has researched the transistor scaling across different generations of lithography advancements. Figure 5.7 shows a graph, displaying an average x2 density increase per generation. TSMC released the 40nm process node in 2008. Based on the chart, the 40nm process node would sit around the 5M TR/mm^2. Additionally, De Vries [34] analyzed the 40nm AMD Bobcat vs the 45nm Intel Atom processor and reported that the 40nm Bobcat processor is much more efficient due to the lower routing density. This is reflected in Bobcat's smaller core size of 4.6mm^2, versus Atom's core size of 9.7mm^2. He estimates that the routing density is x1.78 more efficient for the 40nm process node, than the 45nm process node.

Intel has released numbers on their 45nm process node and has an estimated 3.33 MTr/mm^2 density [35]. Hence, the 40nm process node transistor density is estimated around $3.33 \cdot 1.78 = 5.972$M TR/mm^2, which is roughly in line of the diagram in Figure 5.7. Therefore, the gap between the 14nm process node and the 40nm process node is a difference between 33.8M TR/mm^2 and 5.9M TR/mm^2. The 14nm process node is x5.7 more dense than the 40nm process node. Thus, x5.7 more blocks of 384 PEs can fit on the Groq chip, increasing the new peak performance from 78.6 TFLOP/s to 448 TFLOP/s. The estimated total blocks of 384 PEs on the 14nm process node would grow from 30 to 171 blocks.

5.4. Overview of the ASIC design

This chapter covered the ASIC design process for an LSTM Processing Engine (PE). Using Synopsys Design Compiler and TSMC 40nm PDK, a synthesis was performed on the PE. This design is based on the fixed-point implementation by Haenen [30] due to Xillinx IP-locked components.

For Bonito's HAC model, a block of 384 PEs can process 1 LSTM cell with hidden size = 384 and batch size = 30 in 391 clock cycles. For this block of 384 PEs, the initial fixed-point synthesis resulted in an ASIC design clocked at 500MHz and costs $4.58mm^2$ in area. By shortening the critical path, the final synthesis design reached an clock speed of 869MHz and an area cost of $5.38mm^2$. This is a 74% higher clock speed, at the cost of 20% more area. This was done by introducing small registers that act as pipeline buffer.

With the final synthesis design, a fixed-to-floating-point conversion was done to account for the additional hardware cost and latency, when using floating point. Comparing a fixed-point Multiply-Accumulate unit to a floating-point variant showed that a floating-point design is around x2 slower and

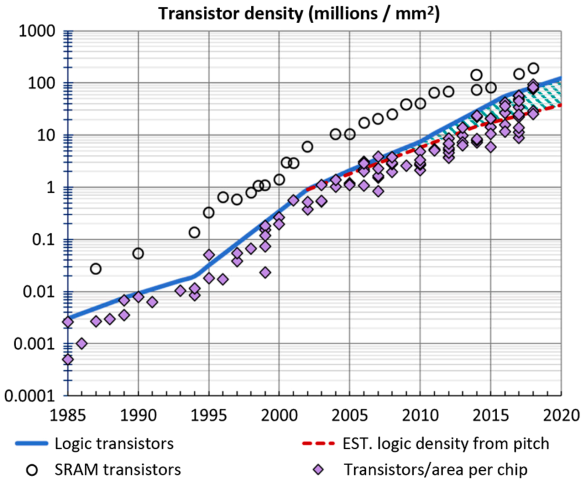


Figure 5.7: Density of transistors has steadily advanced by x2 per generation. This diagram is presented on a logarithmic scale. [33]

costs x1.5 more area. Using this result, the floating-point variant of the final ASIC design is estimated in Table 5.7, with the floating-point design reaching a clock speed of 434MHz and costing $8.07mm^2$ area.

	Fixed-point	Floating-point
Clock Period (ns)	1.15	2.3
Clock Frequency (MHz)	869	434
Area (μm^2)	5379134	8068701

Table 5.7: Difference in latency and area cost, between fixed-point and floating-point for 1 block of 384 PEs.

Next, an estimation of the PE performance is made, using an existing chip layout. This chip layout is based on the Groq die-photo in Figure 2.10, where the Groq compute units are replaced with the ASIC PEs. The Groq chip has $246mm^2$ allocated to compute units. Allocating this area with ASIC PEs resulted in an estimated performance of 78.6 TFLOP/s on the 40nm process node. Then an estimation on process node conversion was made. Table 5.8 shows the final result of 448 TFLOP/s, when using a 14nm process node and applying the ASIC PEs onto the Groq chip layout.

	TFLOP/s	Number of PE blocks	Chip Area(mm^2)	Compute Area(mm^2)
ASIC on Groq (40nm)	78.6	30	725	246
ASIC on Groq (14nm)	448	171	725	246

Table 5.8: Performance estimation when using Groq-based chip layout, for both the 40nm and 14nm process node.

6 Architectural limitations and potential improvements

The results presented in the previous chapters have shown the benefits and limitations of running LSTM layers, used in the Bonito application, on various architectures. This raises the question; what makes an optimized architecture that is specifically tailored to LSTMs?

This chapter highlights which architectural design decisions should be used and which aspects of existing accelerators should be altered in order to optimize for LSTMs. Afterward, the ASIC design is evaluated, and potential improvements are suggested for the current design. Then, the performance of the GPU is compared to the Groq TSP. Additionally, a comparison is made between the original Groq design and the improved Groq design, based on the suggested improvements for Groq. Lastly, the performance comparison between the GPU, Groq TSP, and ASIC design is presented.

6.1. GPU for LSTMs

In the previous chapters, the inference behavior of Bonito on GPUs was covered. There are benefits to using GPUs for LSTMs, but there are also a few drawbacks in the GPU architecture that limit LSTM performance.

The major benefit for GPUs is the potential to spread a computation across many GPU cores. Additionally, supplying the many GPU cores with enough data is possible due to the large VRAM capacity and its ability to enable large batch sizes to be executed in parallel. The limiting factor of LSTMs is its sequential loop dependency, and by enabling large batch sizes, the individual core's low performance is significantly negated by its parallelism. The large VRAM capacity allows enough data to be available to the many GPU cores, enabling a high level of parallelism. Thus, resulting in a higher throughput and a higher computational occupancy on the GPU.

However, the GPU cores are not efficient in executing sequential tasks. Individual core performance is low on GPUs as observed by matrix multiplications and a large number of clock cycles. Results in the previous chapters have shown that for LSTMs, the majority of compute cycles reside in matrix multiplication, up to 70% of the total cycles per LSTM cell. However, the LSTM cells achieve only 12% of the maximum TFLOP/s performance on a GPU.

Furthermore, Bonito's LSTM layers do not occupy all available compute cores on the GPU, leaving resources underutilized at times. Matrix multiplications on GPUs become efficient only when large enough problem sizes are provided, such that it can fully utilize its vast number of compute cores. Therefore, the combination of LSTM's slow sequential processing and Bonito's small problem size led to Bonito's LSTM layers not fully utilizing the available resources.

Additionally, the small L2 cache on-chip forms another factor in the inefficiency of LSTMs on GPUs. Due to the data-intensive nature of LSTMs, many large data blocks are being transferred between VRAM and L2 cache during each LSTM cycle. Especially with larger batch sizes and hidden sizes, the L2 cache is too small to fit all the data. The low data locality requires multiple cache refreshes and forms a memory bandwidth bottleneck.

Potential improvements to LSTM performance

From the perspective of a GPU architecture, an improvement to consider is increasing the individual compute core performance. This can be done by adjusting the matrix multiplication hardware to achieve lower latency at the cost of fewer parallel cores. Results have shown that matrix multiplications are responsible for over 70% of the total clock cycles per LSTM cell. Halving the number of CUDA cores to introduce larger compute cores would decrease the overall clock-cycle cost of matrix multiplication, as it became clear not all available SMs are utilized to their potential. By reducing the number of SMs and improving compute latency, a trade-off is made to balance around throughput and Bonito's problem size. By halving the SM unit count and increasing the compute hardware area x2 for the remaining SM units, the SM unit latency should decrease by half. This effectively doubles the achieved TFLOP/s, while retaining the original area cost, but reducing the maximum TFLOP/s.

Another improvement could be to cache the required data for the current and also the next LSTM cell, at the cost of more L2 cache size. Currently, with a batch of 512, only 1 LSTM cycle can be stored on the cache. In case of increasing the batch size, more data transfers between VRAM and cache are required. Improving data locality could improve the memory bandwidth bottleneck, as it reduces the waiting time between the completion of the LSTM cell $Cell_t$ and the start of the LSTM cell $Cell_{t+1}$.

Theoretically, improving the data locality by pre-caching the next LSTM cell input would result in no idle time by eliminating cache misses. First, the matrix multiplication costs around $12,500$ clock cycles without cache reloading. The baseline RTX 2080 Ti GPU has a memory bandwidth of 616 GB/s, or 616000 MB/s. Transferring 0.4MB from VRAM to L2 cache takes roughly $\frac{0.4}{616000} \cdot 1000000 = 0.65\mu s$ or $650ns$. The GPU clock speed runs at 1545 MHz, giving a clock period of $\frac{1}{1545000000} \cdot 10^9 = 0.65ns$. Thus, once cell $Cell_t$ finishes, new data must come from VRAM, effectively costing $\frac{650}{0.65} = 1000$ clock cycles, before cell $Cell_{t+1}$ can start its execution. Currently, the L2 cache on the RTX 2080 Ti accounts for approximately 5.6% of the chip area. Hence, doubling the L2 cache increases the area cost by 5.6%, and brings an improvement of $\frac{12,500}{13,500} = 7.5\%$ in cycle reduction.

In summary, Bonito's LSTM layers use matrices that are too small to utilize all available compute cores, while the individual compute cores are relatively slow. With limited L2 caches, storing the inputs for $cell_t$ will fill the L2 cache. This causes a L2 cache miss when $cell_{t+1}$ is executed. A potential improvement for GPUs is to reduce the number of SM units, while investing more area for the remaining SM units, such that the compute latency decreases. Additionally, increasing the L2 cache size allows for preloading data for future LSTM cells. Reducing the SM units by half for bigger and lower latency multipliers could improve the achieved TFLOP/s by x2 for no added area cost. Introducing additional cache costs 5.6% additional area for 7.5% increased performance.

6.2. Groq for LSTMs

Groq's architecture is fundamentally different from a GPU. However, despite its impressive specifications, the architecture does not enable LSTMs to be effective in optimizing the available resources.

The properties on the Groq TSP that benefit LSTMs are the fast memory bandwidth for data movement and the fast specialized MXM and VXM units. They enable a calculation to be performed in only a few clock cycles, where the Groq TSP can achieve up to 188 TFLOPs for FP16 and up to 800 TOPs for INT8.

Despite the capabilities to perform large matrix multiplications very fast, the LSTM layers in Bonito utilize an even bigger weight and input matrix for its calculation. This causes Groq to slice the logical tensor into smaller physical tensor slices, at a maximum of 320-elements long and 160-elements wide. Mismatching the LSTM input tensors leads to partial slices that cover only a portion of the 320-elements long physical tensor shape. This leads to inefficient utilization of the computational resources. In case of the Bonito HAC model with 384 features at FP16, Groq would slice the input tensor into 2 slices. Effectively, splitting the multiplication matrix into 2 smaller matrices which utilize only 192 out of 320 available compute units. Furthermore, the weight tensor of [384, 384] would be sliced into 6x [128, 192], as the weight buffer can only hold up to 160-elements wide and up to 320-elements long, at FP16. Compared to a weight tensor of [320, 320], it would be sliced into 2x [160, 320]. This is x3 more efficient.

These slices lead to the following problem of constant reloading of the MXM weight buffer. As the LSTM multiplication has both a Weight matrix W and Activation matrix A that gets slices by Groq, matrix

W is required to be reloaded 2 times, every LSTM cell cycle. Reloading the MXM weight buffer and streaming matrix A into the MXM multiplication happens over the same streaming channel. Thus, only after MXM weight buffers are loaded in 40 cycles, can matrix A start streaming its input data, but before the next slice of matrix W gets reloaded into the weight buffer. This introduces stalls during MXM weight reloading.

Despite the fast on-chip SRAM, the memory and streaming register channels are also a limiting factor for LSTMs on Groq. These channels accommodate both weight and activation matrices. This means that the streaming channel is used first to reload the Weight matrix W onto the MXM weight buffer, before it streams the Activation matrix A. While streaming matrix A, the next slice of the matrix W cannot be preloaded. Thus, it must wait until matrix A finishes. This creates a gap between multiplications and reduces the overall computational performance.

Data movement between components occur in stages. A 320-element vector and a 1-element vector require the same clock cycles to travel across the streaming channels. As shown in the previous chapters, there is no direct data path between components, but data flows across stream registers eastward or westward on the streaming channel, traversing one register per clock cycle. Traversing from one side to the other takes 44 clock cycles in the worst-case data placement in SRAM. This architecture fits calculations that can be pipelined, but LSTMs cannot make full use of the resources, as the matrix addition can only be done after the matrix multiplication finishes. This causes the Groq compiler to accumulate the full matrix multiplication result in MEM, before executing the addition and special functions in the VXM module. Additionally, inside the VXM, each type of operation needs to store its result on MEM as well, before it can be used in future operations.

Results from the previous chapters have shown that the VXM is severely bottlenecking the overall performance. Around 50% of the total computation is spent in the VXM, where addition, element-wise multiplication, Sigmoid and Tanh are performed. This indicates that the MXM units are unused 50% of the time. Especially the Sigmoid is expensive to calculate, as it is twice as expensive as the Tanh, addition, and element-wise multiplication calculation.

Large chunk sizes lead to long sequences and this is not possible on Groq due to Groq's limited SRAM capacity. Similarly, large batch sizes increase the SRAM utilization as well. In the scenario of Bonito, long sequences and large batch sizes are used. The default chunk size is 10000 and batch size is 512. Due to the deterministic behavior of the Groq Compiler, each instruction is stored on SRAM, meaning that longer sequences require more instruction ops to be executed. Hence, increasing the on-chip utilization, while adding no benefit in overall throughput. On the other hand, increasing batch size does improve overall throughput, albeit at the cost of more SRAM. As the test results have shown, the ideal batch size is 1600, reaching peak throughput of 86.40 TFLOPs. However, in context of Bonito, long sequences are desired such that the model can utilize its ”memory state” to more accurate perform basecalling. With short sequences, the basecalling accuracy is significantly affected, indicating that the small SRAM is quite a downside.

Potential improvements to LSTM performance

In terms of architectural improvements, having more than 320 data lanes would benefit LSTMs to execute larger models. By adapting the architecture to the requirements for Bonito's LSTM layer, a higher computational load can be achieved. For example, increasing the width of the lane from 320 to 384 lanes would save 1 additional slice that would be created, improving the performance by 50%. Instead of having $ceil(\frac{384}{320}) = 2$ slices with 320 lanes, $ceil(\frac{384}{384}) = 1$ slice is required when using 384 lanes. Likewise, adjusting the Bonito LSTM parameters to fit the current 320 lanes on Groq would result in the same performance improvement. However, this requires a software-based change in the Bonito model, rather than adjusting the hardware architecture. Thus, changing from 320 to 384 lanes could potentially result in x1.44 higher performance. However, this brings an added chip area cost, to support up to 384-element long matrices. The added cost can be estimated as adding 4 additional superlanes in the design, which contain the MXM, SXM, MEM and VXM portion as well. Each superlane in Figure 2.10 is occupies around 4.21% of the total chip die. This brings an additional area cost of $4 \cdot 4.21 = 16.84\%$, for a 44% throughput increase. Thus, adding more lanes increases both the maximum TFLOP/s and the achieved TFLOP/s. But the achieved TFLOP/s increases more than the maximum TFLOP/s, as the resource utilization goes up when saving 1 tensor slice.

The memory bandwidth bottleneck is a problem that is mainly caused by the MXM weight reloading, caused by the physical tensor shape slicing. On Groq, MXM LW operations cost many clock cycles, relative to the actual matrix multiplication clock cycle cost. Reducing these MXM LW operations can be done by increasing the MXM buffer size or by introducing extra SRAM that specifically store the weight tensor data. This way, the ”up to 40” cycles of MXM LW operations can be reduced to 20 cycles, when the extra SRAM is directly next to the MXM. Meanwhile, the streaming channels can pre-fetch the activation matrix A to immediately start the matrix multiplication when MXM LW is finished. Reducing the initial 40 cycles to 20 cycles for MXM LW means that the MXM can do twice the amount of work, compared to the stalls introduced by the original design. This would result in the MXM requiring only half the cycles, compared to before. That means the overall cycle count would be reduced by 25%, assuming that MXM and VXM account for 50% of cycles each. The cost would be a small area increase, as the weight tensor required to store is of the size 8x [320, 320]. That gives an SRAM size of 1.6MB required to store only the weights. This is less than 1% of the available SRAM, meaning that the additional area cost is less than 1%.

Another more straightfoward improvement could be by simply increasing the size of the VXM. As the VXM accounts for 50% of the LSTM cycles, it is an quick approach to reduce the overall LSTM cycles. Ideally, the VXM is balanced around the fact that it can finish the addition, element-wise multiplication, Sigmoid, and Tanh, at the same time it takes the MXM to load the new data between $cell_t$ and $cell_{t+1}$. In other words, increasing the VXM performance will directly increase the throughput, as it decreases the idle time where the MXM is not used. For example, doubling the VXM unit will result in a 25% decrease in overall LSTM cycles. The VXM contributes for around 5.9% of the chip area. Doubling the VXM would mean an additional area cost of 5.9%.

To solve the chunk size limitation, it is possible to introduce special buffer registers that only store the newest outputs h_t and c_t for each LSTM cell cycle. By doing so, it is possible to use a smaller chunk size as a model input for the Groq compiler, while theoretically remembering the outputs h_t and c_t in between chunks. Doing so should reduce the number of execution instructions stored on SRAM and enable a longer sequence to be analyzed by the LSTM model. According to the Groq API, it is possible to write custom low-level code, which can direct the data movement to a specific SRAM area and utilize it for the next Groq execution. The concept is called “persistent data”, but this could not be implemented due to limited possibilities as external Groq user. Hence, introducing special state-buffers achieves the same result.

In summary, the Groq TSP is currently mismatched with Bonito’s hidden size, with the Groq TSP optimized for 320 hidden features and Bonito using 384 hidden features. The slicing concept to fit Groq’s physical tensor shape also introduced a bandwidth bottleneck on the MEM and streaming channels, by constantly requiring weight reloading in MXM. Furthermore, the VXM unit is forming a compute bottleneck in the LSTM cycle. Lastly, Groq has a limited SRAM capacity of 220MB and no concept of VRAM on the TSP. Long chunk sizes are not possible, as longer chunks introduce more execution instructions and these instructions are stored on the SRAM as well.

A potential improvement for the Groq TSP is introducing more compute hardware, such that Groq can support up to 384 elements per cycle and optimize for Bonito’s HAC model. Another improvement is to allocate specialized MXM buffers for the weight data, such that the streaming channels bandwidth is not consumed by the MXM LW operation. Furthermore, increasing the VXM compute performance will directly reduce the clock cycle cost of the LSTM execution. Lastly, enabling longer chunk sizes can be done by using introducing specialized *state buffers* that store only the hidden and memory outputs in between cells. All these changes bring a certain estimated throughput improvement at a certain area cost. These numbers are presented in Table 6.1.

	Performance increase	Area increase
Increasing from 320 to 384 elements	44.0%	16.8%
Introducing special MXM buffers	25.0%	<1%
Doubling VXM compute units	25.0%	5.91%
State-buffers	Increases only maximum chunk size	<1%

Table 6.1: Estimated performance improvements, when applying said design changes on the Groq TSP.

6.3. ASIC for LSTMs

From an architectural perspective, there are some similarities between the initial ASIC design and the Groq design. In terms of compute capabilities, the ASIC matrix multiply-accumulate stage is designed to be executed in 384 clock cycles, for hidden size = 384, whereas Groq requires roughly 80 cycles to accumulate each partial [160, 320] matrix. Compared to a GPU that requires over 12,500 clock cycles to execute 1 LSTM multiplication for Bonito's HAC model, Groq and the ASIC design optimize around low latency per 1 LSTM cell calculation.

However, the difference between the ASIC design and Groq is how Groq uses a heterogeneous design with different compute units, whereas the ASIC uses a similar approach to GPUs. The ASIC design consists of specialized PEs and increases performance by scaling the number of PEs to, depending on the available chip area and requirements. This is similar to the increase in GPU SM units for larger GPUs.

When designing the ASIC, the initial LSTM design showed potential, with the ability to perform a single LSTM cycle in only 393 clock cycles. However, the current matrix multiply-accumulate stage inside the LSTM PE forms a significant bottleneck, costing 384 out of 391 cycles. Chapter 4 showed how fast each Groq MXM can perform a multiply-accumulate stage, in roughly 80 cycles for matrices of size [160, 320]. However, with multiple blocks of 384 PEs, the multiply-accumulate stage remains at 384 cycles, as computing the batch size happens in parallel, while Groq needs to slice its large matrix in smaller partial multiplication matrices. To put this in perspective, we use the example of batch = 160 and hidden = 320 and chunk = 1. Groq would perform 8x [160, 320] X [320, 320] multiplications, which get sliced into 16x [160, 320] X [160, 320] multiplications, and then split across 2 MXM modules, for a total of 640 cycles.

Potential improvements in current ASIC design

The ASIC design can currently fit only 30 blocks of PEs on the 40nm process node. Estimating the conversion from 40nm to Groq's 14nm process node, the number of blocks increase to 171 blocks. This means that the 14nm ASIC design can support up to batch size = 171, while only requiring 384 cycles. This shows how the ASIC design performs better with larger batch sizes, compared to the Groq design, which is more optimized around minimal MXM weight reloading.

Therefore, an improvement in the current ASIC design could be to change the multiply-accumulate stage to reduce clock cycles. It is not clear exactly how Groq implemented its MXM multiply-accumulate architecture. However, it is possible to reduce the cycles, as Groq has proved, by using a different approach in multiply-accumulate.

Another improvement is to double the MAC components per PE, such that MAC component can alternate between two stages. Instead of one multiply-accumulate for 384 cycles, a second multiply-accumulate starts in the extra MAC components, such that the post-processing step has a lower downtime. This leads to the post-processing step to wait for only 193 cycles, instead of 384 cycles. However, this change is rather expensive in terms of area, as the MAC components are the most expensive in terms of area cost. A single PE in the final fixed-point design costs 14,550 μm^2, with a single MAC component costing around 1730 μm^2. Doubling 4 MACs to 8 MACs per PE increases the total area cost to at least 21,420 μm^2, an increase of x1.47. This would lead to a cycle reduction of up to 50% less cycles, leading to a x2 improvement in throughput.

In the critical path, the multiplication and addition after point-wise multiplication forms the next bottleneck, as the multiplication result from $sigmoid_o \times tanh_c$ produces a 32-bit number, which needs to be resized to 16 bits. Splitting this critical path in two sections can lead to a higher clock frequency, which leads to a faster multiply-accumulate stage. The multiply-accumulate stage takes 384 out of 391 cycles, meaning that a higher clock speed will benefit the multiply-accumulate stage the most, at the cost of one or two additional cycles in the 7-stage post-processing step.

In summary, the multiply-accumulate stage in the ASIC is a bottleneck, consuming 384 out of 391 cycles per LSTM cell. There are two suggestions to improve this. First, Groq has shown that a fast multiply-accumulate is possible. Hence, it should be possible to design the current multiply-accumulate stage to cost less cycles. Second, by doubling the number of MAC units in each PE, the post-processing stage of 7 cycles is used more often. This means that the post-processing stage is idle only every 192 cycles, instead of every 384 cycles. For the other bottleneck, the current critical path limits the overall ASIC clock speed. Introducing an additional stage between multiplying $sigmoid_o \times tanh_c$ and resizing the multiply result to 16 bits should reduce the critical path latency, at the cost of 1 additional cycle. This increases the current 391 cycles to 392 cycles.

6.4. Groq vs GPU vs ASIC

The following section assumes an FP16 precision for Bonito's HAC model and TFLOP/s performance. From the initial Bonito execution comparison, Groq did not outperform the baseline GPU RTX 2080 Ti in terms of runtime. However, with optimized input tensors and matching batch sizes, the achieved LSTM performance on Groq is much higher than the RTX 2080 Ti. As the GPU reached around 13 TFLOP/s at batch = 1600, Groq performs x6.46 better at 86 TFLOP/s.

The results show that, on Groq, the highest-performing LSTM batch size is 1600, rated at 86.15 TFLOP/s. This batch size is explained by the fact that Groq supports MXM weight matrices of up to 160 elements wide. The batch size = 1600 is a perfect even multiple. However, the achieved TFLOP/s is much lower than the peak 188 TFLOP/s claimed by Groq. Around 45% of the potential throughput is reached. If the suggested improvements for Groq could be applied, it would be interesting to see the potential combined speedup estimates.

First, increasing lane width from 320 to 384 would increase the overall TFLOP/s throughput by x1.47. Second, introducing special SRAM buffers for MXM weights can cut the MXM LW operation cycles by half. As an example, there are 16 partial matrix multiplications scheduled, each costing 40 MXM LW and 40 accumulate cycles, and there are 2 MXM modules in parallel. In total, the normal Groq design would require $80 \cdot \frac{16}{2} = 640$ cycles. Now, the Groq design with special MXM weight buffers reduce the total cycles to $60 \cdot \frac{16}{2} = 480$. This is an 25% improvement in MXM cycles. That leads to 12.5% less cycles required in the whole LSTM cell. Third, doubling the VXM should reduce the time spent in VXM by half. That gives an overall reduction of 25% of cycles required in the whole LSTM cell.

So, assuming the example parameters of batch = 160, hidden = 320 and chunk = 1, the total cycles was 1984 for 1 LSTM cell, with 50% accounting for either MXM or VXM calculations. By applying all improvements, the total cycles would be $988 \cdot 0.75 + 986 \cdot 0.5 = 1234$ cycles. This results in a 37% reduction in cycles. The best-performing batch size = 1600 reached 86.15 TFLOPs. Reducing the required cycles by 37% results in a new throughput of 136.75 TFLOPs. This is 72% of the peak performance of 188 TFLOPs. Adding the x1.47 increase in overall TFLOP/s with increasing lane width to 384, the final throughput would reach 201 TFLOP/s. This is a x2.33 performance increase from the 86.15 TFLOP/s. The added area cost will be $5.9 + 0.8 + 16.84 = 23.54\%$, or x1.23

Performance comparison

The suggested improvements in the Groq architecture are promising to achieve higher LSTM performance. In terms of comparing Groq to the baseline RTX 2080 Ti, the improved Groq design could potentially be x15.4 times faster.

The final ASIC design reaches an estimated peak performance of 448 TFLOP/s, with the adjustment to the correct 14nm process node and accounting for the fixed-to-floating-point conversion. Compared to the original Groq design with optimized tensors and batches, the ASIC performs x5.26 faster. Compared to the improved Groq design, the ASIC remains x2.25 faster. The numbers show that the ASIC has potential to be competitive with existing accelerators for LSTM applications. Table 6.2 summarizes the performance numbers for different accelerators used, together with the corresponding area costs.

	Achieved TFLOP/s	Max TFLOP/s	Utilized TFLOP/s	Area cost (mm^2)	Area increase	TFLOP/s improvement from original
Nvidia RTX 2080 Ti 12nm (original)	11.5	107	10.6%	754	0%	0%
Groq TSP 14nm (original)	22	188	11.8%	725	0%	0%
Optimizing MXM usage	86	188	45.7%	725	0%	284%
Introducing special MXM buffers	98	188	52.1%	726	<1%	339%
Doubling VXM compute units	136	188	72.3%	768	6.8%	510%
Increasing from 320 to 384 elements	196	226	86.7%	897	23.7%	797%
ASIC on Groq layout No optimizations (40nm)	79	79	100%	725	0%	0%
ASIC on Groq layout No optimizations (14nm)	448	448	100%	725	0%	470%

Table 6.2: Overview of the different accelerators used in this thesis, with the achieved performance and area cost. Additionally, the Groq improvement estimations are included and the different improvement numbers are compounded.

7 Conclusion and recommendations

This thesis researched the performance of existing hardware accelerators and custom ASIC accelerators when performing genome sequencing and basecalling the DNA reads with Oxford Nanopore Technologies' Bonito program. Specifically, the performance of LSTM layers in Bonito's neural network model was further studied and analyzed. Real-time basecalling brings computational challenges, as large datasets produced by the nanopores require immense computing power to perform the basecalling process. Hence, an analysis is performed on existing GPU and Groq accelerators. Afterward, an ASIC analysis is performed to determine the possibilities and viability of a custom LSTM accelerator.

First, the hardware architecture of GPUs and Groq accelerators has been presented. This enabled the in-depth LSTM performance analysis to understand the limitations of LSTMs in general, and what the limitations of each architecture are, in the context of LSTMs. The in-depth analysis started with the GPU analysis. This led to the finding that the problem size of Bonito is not large enough to utilize all available compute cores fully. Scaling the batch size to the point where the VRAM was fully used, the GPU achieved 13 TFLOP/s out of the maximum 107.4 TFLOP/s available. This translates to 12.1% of the total TFLOP/s available on the GPU. When using Bonito's default batch size = 512, the GPU achieved 11.5 TFLOP/s and utilizes 10.5% of the total TFLOP/s

After the GPU, the Groq accelerator was analyzed. The Bonito High-Accuracy (HAC) model performed worse than the GPU, when no optimization was applied. However, an in-depth behavior analysis revealed major optimization opportunities. The most prominent finding is the mismatch between input tensor and Groq's internal physical tensor shape. Groq uses 320-element wide streaming channels, while Bonito's HAC model uses 384 elements for the hidden features. The input from Bonito would be sliced in two parts and require double operation cycles. Hence, matching the physical tensor resulted in a x2 increased throughput performance, as fewer slices were needed.

The second finding originated from a 1-cell execution analysis. We observed the MXM and VXM account both for 50% of the execution cycles per LSTM cell. Further investigation revealed that the Vector-Execution-Module (VXM) was severely bottlenecking the overall LSTM performance. In terms of FLOP cost, the post-processing step in the VXM accounts for only 0.26% of the total FLOP cost per LSTM cell, as the majority of calculations are matrix multiplications. Hence, allocating 50% of the cycles for 0.26% of FLOPs is not efficient.

The third finding is how the memory organization does not benefit LSTMs. MXM LW operations handle the weight buffer reloading, which requires data to travel from Memory-Execution-Module (MEM) to the MXM units. During this travel, the MXM cannot perform any calculations, as the streaming channels, responsible for data transfers, are occupied with in-transit data. This introduces stalls in the execution sequence, where only MEM is active. Applying these improvements on the Groq TSP led to an estimated x15.4 performance improvement over the baseline GPU RTX 2080 Ti, estimating Groq's improved performance at 196 TFLOP/s. This comes at an added area cost of 23.7% of the original Groq chip area.

The GPU and Groq analysis served as a bridge to investigate the potential of an ASIC for LSTMs in Bonito. The ASIC synthesis revealed new insights on the original FPGA design by Haenen. Small improvements were introduced in the final fixed-point ASIC design to obtain a clock speed of 869MHz and an area cost of $5.38mm^2$ per 1 block of 384 PEs. The fixed-to-floating-point analysis was done to correct the difference between fixed-point and floating-point in hardware cost. This showed that the floating-point ASIC design is estimated to have a clock speed of 434MHz and an area cost of $8.07mm^2$ per block. This thesis used the TSMC 40nm process node, where the performance is estimated at 79.3 TFLOP/s. Converting to Groq's 14nm process node resulted in an estimated performance of 448 TFLOP/s, based on the $725mm^2$ Groq chip architecture layout.

With the in-depth analysis of GPUs, Groq accelerators and custom ASIC accelerators, the results show that pursuing a custom ASIC for Bonito's LSTM layers is viable for further development. The comprehensive analysis of existing accelerator architectures identified the computational limitations present in LSTMs, and uncovered the architecture-specific limiting factors present in each accelerator. These limitations led to a collection of suggested improvement avenues that were taken into consideration in the ASIC design. The ASIC analysis then identified limitations and possibilities to further improve the current design, demonstrating that the ASIC development remains an opportunity for future endeavors.

Recommendations

The discoveries from this thesis lead to new research opportunities, recommending the following topics for future work:

- Further research on data and memory control is advised to gain a more precise understanding of the ASIC chip-die layout and clock speed in synthesis and placement.
- Research possibilities to reduce the MAC cycle cost in the Processing Engine.
- Explore the ASIC workflow in a modern process node to obtain accurate numbers for comparing state-of-the-art Nvidia H100 GPUs to the ASIC performance.

Bibliography

[1] *The Human Genome Project — genome.gov*, `https://www.genome.gov/human-genome-project`, [Accessed 01-09-2024].

[2] P. H. F. Ran, *Genome engineering using the CRISPR-Cas9 system | Nature Protocols — rdcu.be*, `https://rdcu.be/dSEqo`, [Accessed 01-09-2024].

[3] P. Zheng, C. Zhou, Y. Ding, *et al.*, "Nanopore sequencing technology and its applications," *MedComm*, vol. 4, no. 4, Jul. 2023, ISSN: 2688-2663. DOI: `10.1002/mco2.316`. [Online]. Available: `http://dx.doi.org/10.1002/mco2.316`.

[4] M. MacKenzie and C. Argyropoulos, "An introduction to nanopore sequencing: Past, present, and future considerations," *Micromachines*, vol. 14, no. 2, p. 459, Feb. 2023, ISSN: 2072-666X. DOI: `10.3390/mi14020459`. [Online]. Available: `http://dx.doi.org/10.3390/mi14020459`.

[5] O. N. Technologies, *MinION portable nanopore sequencing device — nanoporetech.com*, `https://nanoporetech.com/products/sequence/minion`, [Accessed 03-09-2024].

[6] O. N. Technologies, *PromethION - Oxford Nanopore Technologies — nanoporetech.com*, `https://nanoporetech.com/products/sequence/promethion`, [Accessed 03-09-2024].

[7] A. J. Marian, "Medical dna sequencing," *Current Opinion in Cardiology*, vol. 26, no. 3, pp. 175–180, May 2011, ISSN: 0268-4705. DOI: `10.1097/hco.0b013e3283459857`. [Online]. Available: `http://dx.doi.org/10.1097/HCO.0b013e3283459857`.

[8] V. Higuera, *Genes: Function, makeup, Human Genome Project, and research — medicalnewstoday.com*, `https://www.medicalnewstoday.com/articles/120574`, [Accessed 11-09-2024].

[9] J. Quer, S. Colomer-Castell, C. Campos, *et al.*, "Next-generation sequencing for confronting virus pandemics," *Viruses*, vol. 14, no. 3, p. 600, Mar. 2022, ISSN: 1999-4915. DOI: `10.3390/v14030600`. [Online]. Available: `http://dx.doi.org/10.3390/v14030600`.

[10] S. Kovaka, S. Ou, K. M. Jenike, and M. C. Schatz, "Approaching complete genomes, transcriptomes and epi-omes with accurate long-read sequencing," *Nature Methods*, vol. 20, no. 1, pp. 12–16, Jan. 2023, ISSN: 1548-7105. DOI: `10.1038/s41592-022-01716-8`. [Online]. Available: `http://dx.doi.org/10.1038/s41592-022-01716-8`.

[11] Y. Wang, Y. Zhao, A. Bollas, Y. Wang, and K. F. Au, "Nanopore sequencing technology, bioinformatics and applications," *Nature Biotechnology*, vol. 39, no. 11, pp. 1348–1365, Nov. 2021, ISSN: 1546-1696. DOI: `10.1038/s41587-021-01108-x`. [Online]. Available: `http://dx.doi.org/10.1038/s41587-021-01108-x`.

[12] O. N. Technologies, *How nanopore sequencing works | Oxford Nanopore Technologies — nanoporetech.com*, `https://nanoporetech.com/platform/technology`, [Accessed 26-08-2024].

[13] O. N. Technologies, *GitHub - nanoporetech/dorado: Oxford Nanopore's Basecaller — github.com*, `https://github.com/nanoporetech/dorado`, [Accessed 26-01-2024].

[14] O. N. Technologies, *GitHub - nanoporetech/bonito: A PyTorch Basecaller for Oxford Nanopore Reads — github.com*, `https://github.com/nanoporetech/bonito`, [Accessed 26-08-2024].

[15] Z. A.-A. Mees Frensel and H. P. Hofstee, *Learning structured sparsity for efficient nanopore dna basecalling using delayed masking*, In review for the ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, [Accessed 20-08-2024], 2024.

[16] *Size of output of a Conv1D layer in Keras — stackoverflow.com*, `https://stackoverflow.com/questions/65006011/size-of-output-of-a-conv1d-layer-in-keras`, [Accessed 11-09-2024].

[17] G. Chevalier, *File:the lstm cell.svg*, 2018. [Online]. Available: https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg.

[18] Nviia, *NVIDIA Ampere Architecture — resources.nvidia.com*, https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper?xs=169656, [Accessed 26-08-2024].

[19] AWS, *GPU vs CPU - Difference Between Processing Units - AWS — aws.amazon.com*, https://aws.amazon.com/compare/the-difference-between-gpus-cpus/, [Accessed 22-03-2024].

[20] M. Hernández, G. D. Guerrero, J. M. Cecilia, *et al.*, “Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus,” *PLOS ONE*, vol. 8, no. 4, pp. 1–13, Apr. 2013. DOI: 10.1371/journal.pone.0061892. [Online]. Available: https://doi.org/10.1371/journal.pone.0061892.

[21] Nvidia, *NVIDIA Turing Architecture Whitepaper Available Now For Download — nvidia.com*, https://www.nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/, [Accessed 20-08-2024].

[22] D. Abts, J. Ross, J. Sparling, *et al.*, “Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain: IEEE, May 2020.

[23] D. Abts, J. Kim, G. Kimmell, *et al.*, “The groq software-defined scale-out tensor streaming multiprocessor : From chips-to-systems architectural overview,” in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–69. DOI: 10.1109/HCS55958.2022.9895630.

[24] R. Casado-Vara, Á. Rey, D. Pérez-Palau, L. de-la-Fuente-Valentín, and J. Corchado Rodríguez, “Web traffic time series forecasting using lstm neural networks with distributed asynchronous training,” *Mathematics*, vol. 9, p. 421, Feb. 2021. DOI: 10.3390/math9040421.

[25] OneFlow, *How to Choose the Grid Size and Block Size for a CUDA Kernel? — oneflow2020.medium.com*, https://oneflow2020.medium.com/how-to-choose-the-grid-size-and-block-size-for-a-cuda-kernel-d1ff1f0a7f92, [Accessed 28-08-2024].

[26] J. Alsop, M. D. Sinclair, S. Bharadwaj, *et al.*, “Optimizing gpu cache policies for mi workloads,” in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 243–248. DOI: 10.1109/IISWC47752.2019.9041977.

[27] M. Emani, Z. Xie, S. Raskar, *et al.*, “A comprehensive evaluation of novel ai accelerators for deep learning workloads,” in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, IEEE, Nov. 2022. DOI: 10.1109/pmbs56514.2022.00007. [Online]. Available: http://dx.doi.org/10.1109/PMBS56514.2022.00007.

[28] L. Gwennap, “Groq rocks neural networks,” *Microprocessor Report, Tech. Rep., jan*, 2020.

[29] J. Daw, C. Seymour, malton-ont, *et al.*, *Nanoporetech/dorado*, May 24, 2024. [Online]. Available: https://github.com/nanoporetech/dorado.

[30] J. Haenen, *Accelerating DNA basecalling of Nanopore reads on FPGAs | TU Delft Repository — repository.tudelft.nl*, https://repository.tudelft.nl/record/uuid:06115276-884c-4335-a7d8-024dcd59731f, [Accessed 15-01-2024], 2023.

[31] *Microprocessing - Sensing Control — sensingcontrol.com*, https://sensingcontrol.com/en/microprocessing/, [Accessed 20-08-2024], 2024.

[32] D. Schor, *TSMC Announces 6-Nanometer Process — fuse.wikichip.org*, https://fuse.wikichip.org/news/2261/tsmc-announces-6-nanometer-process/, [Accessed 26-08-2024].

[33] M. Rieger, “Retrospective on vlsi value scaling and lithography,” *Journal of Micro/Nanolithography, MEMS, and MOEMS*, vol. 18, Nov. 2019. DOI: 10.1117/1.JMM.18.4.040902.

[34] H. de Vries, *Chip Architect: AMD&apos;s 40nm Bobcat versus Intel&apos;s 45nm Atom — chip-architect.com*, http://www.chip-architect.com/news/2010_09_04_AMDs_Bobcat_versus_Intels_Atom.html, [Accessed 29-08-2024].

[35] I. Cutress, *Intel's 10nm Cannon Lake and Core i3-8121U Deep Dive Review — anandtech.com*, `https://www.anandtech.com/show/13405/intel-10nm-cannon-lake-and-core-i3-8121u-deep-dive-review/3`, [Accessed 26-08-2024].