CycloneNTT: An NTT/FFT Architecture Using Quasi-Streaming of Large Datasets on DDR- and HBM-based FPGA Platforms

Kaveh Aasaraai, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, Javier Varela, Kevin Bowers
{kaasaraai,jvarela,kbowers}@jumptrading.com
{ecesena,rmaganti,nicolas}@jumpcrypto.com

ABSTRACT

Number-Theoretic-Transform (NTT) is a variation of Fast-Fourier-Transform (FFT) on finite fields. NTT is being increasingly used in blockchain and zero-knowledge proof applications. Although FFT and NTT are widely studied for FPGA implementation, we believe CycloneNTT is the first to solve this problem for large data sets ($\geq 2^{24}$, 64-bit numbers) that would not fit in the on-chip RAM. CycloneNTT uses a state-of-the-art butterfly network and maps the dataflow to hybrid FIFOs composed of on-chip SRAM and external memory. This manifests into a quasi-streaming data access pattern minimizing external memory access latency and maximizing throughput. We implement two variants of CycloneNTT optimized for DDR and HBM external memories. Although historically this problem has been shown to be memory-bound, CycloneNTT's quasi-streaming access pattern is optimized to the point that when using HBM (Xilinx C1100), the architecture becomes compute-bound. On the DDR-based platform (AWS F1), the latency of the application is equal to the streaming of the entire dataset $\log N$ times to/from external memory. Moreover, exploiting HBM's larger number of channels, and following a series of additional optimizations, CycloneNTT only requires $\frac{1}{6} \log N$ passes.

ACM Reference Format:
Kaveh Aasaraai, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, Javier Varela, Kevin Bowers. . CycloneNTT: An NTT/FFT Architecture Using Quasi-Streaming of Large Datasets on DDR- and HBM-based FPGA Platforms. In *Proceedings of (Jump Trading / Jump Crypto).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

The Number Theoretic Transform (NTT) is a generalization of the Fast Fourier Transform (FFT). The FFT, first presented in [CT65] and credited back to Gauss, remains unparalleled in its impact on modern computer science, having largely been responsible for the birth of digital signal processing. As a result, numerous attempts have been made over the last half century to optimize and improve both the asymptotic and concrete efficiency of the FFT.

Classically, the FFT is computed over the field of complex numbers, which has an interpretation as transforming periodic functions from the time to the frequency domain. When the computation is done over finite fields, the FFT is usually referred to as NTT. It has an interpretation as transforming polynomials over the finite field from their coefficients to their value representations, and it's used for example for large-polynomial multiplication, or for interpolating polynomials at given values.

Jump Trading / Jump Crypto, 2022, Chicago, IL
© 2022

Modern applications in cryptography, ranging from homomorphic encryption to lattice-based and other post-quantum primitives, to zero-knowledge (ZK) proofs, motivated a recent review of classical FFT algorithms and research in efficient implementations over finite fields. And, as richer applications are built or proposed, the need to compute NTT over larger datasets grows.

Our work is motivated by ZK applications, particularly in the context of blockchain technology. ZK proofs are a building block to build both scalability solutions (e.g., ZK-rollups) and privacy-enhanced applications for blockchains. In many concrete schemes, generating a ZK proof requires computing a NTT, typically on a fairly large dataset.

Contribution. In this paper we present CycloneNTT, a FPGA solution for computing NTT on large datasets ($\geq 2^{24}$, 64-bit elements), that thus require external memory.

By leveraging the algorithm introduced in [BLDS10] to reduce memory accesses, and applying a series of algorithmic- and implementation-level optimizations, CycloneNTT achieves a quasi streaming data access pattern that maximizes throughput.

The architecture is configurable, and we apply it to DDR- and HBM-based platforms. Moreover, we show how the designer can trade off power for delay, depending on the target system and environmental conditions.

Organization. In Sec. 2 we present related work, and in Sec. 3 we define terminology and notation. We introduce CycloneNTT in Sec. 4 describing requirements, architecture and high-level interaction between host and FPGA. In Sec. 5 and 6 we present two instances of CycloneNTT: a single-layer streaming implementation that is memory bounded, and a multi-layer streaming one that can overcome memory bandwidth constrains and become compute-bound. We show experimental results in Sec. 7 and we conclude in Sec. 8 highlighting future works.

2 RELATED WORK

Following the growing interest in ZK proofs, recent efforts have been made to map them to specialized hardware architectures. Of particular relevance are PipeZk for ASIC [ZWZ+21], and NTTGen for FPGA [YKKP22].

PipeZk is a pipelined accelerator for zkSNARKs targeting an ASIC architecture, and composed of two subsystems: one for polynomial computation (including NTT) and one for multi-scalar multiplication. The former performs a recursive decomposition of large NTT kernels into smaller tiles, which allows them to fit into on-chip compute resources while complying with off-chip bandwidth limitations. In PipeZk, multiple modules can run in parallel to optimize data utilization. A module is composed of NTT cores, each

performing the butterfly operations, and employing FIFOs of different depth to match the required stride access. Because of the data-access pattern, the architecture needs to block data in on-chip SRAM (performing a matrix transpose) before it is able to write it back to off-chip memory. Although it is an interesting architecture, it targets a limited range of input sizes from 2^{14} to 2^{20} elements. And being an ASIC design means that it loses the deployment flexibility and time-to-market offered by FPGAs.

NTTGen is a hardware generation framework targeting FPGAs, optimized for homomorphic encryption. The inputs to the framework are application parameters (such as latency, polynomial degree, and a list of prime moduli), and hardware resources constraints (e.g. DSP, BRAM and I/O bandwidth), while the output is synthesizable Verilog code. It exploits data-, pipeline- and batch-parallelism, and offers two flavours of cores: general purpose, and low-latency customized for generalized Mersenne primes. Input and output polynomials are stored in off-chip memory, and results are shown for polynomial degrees ranging from 2^{10} to 2^{14}. At the heart of the design is the use of Streaming Permutation Networks (SPN) [CP15], which allows for arbitrary permutation strides, while reducing the interconnect complexity and avoiding expensive crossbars. A SPN consists of three sub-networks: two of them for spatial permutations (in the same cycle) and one for temporal-permutation (across cycles). NTTGen actually extends the original SPN to support runtime-control of the underlying routing tables and address generations. In comparison, CycloneNTT takes a different approach: we avoid permutations entirely by using a multi-FIFO architecture backed by off-chip memory.

Worth mentioning as well is HEAX [RLPD20], a highly parallelizable hardware architecture targeting fully homomorphic encryption, with pipelined NTT cores and a word-size of 54 bits to maximize resources utilization (DSPs). It requires on-chip multiplexers to distribute data and twiddle factors to the cores, and is mainly optimized for on-chip memory usage up to 2^{13} elements. Beyond that, it employs off-chip memory but only to a very limited extent due to the latency overhead.

Previous relevant art targeting FPGAs also include [SRTJ+19], [KLC+20] for homomorphic encryption, [DM22] for lattice-based cryptography, [MK20] for homomorphically encrypted deep neural network inference, as well as [MKO+20] for post-quantum digital signature scheme. Nevertheless, these designs make primarily use of on-chip BRAM and do not directly provision the use of off-chip memory. All these work could benefit from CycloneNTT approach to scale to larger NTTs and possibly allow for richer applications.

Other interesting references, although not directly applicable to CycloneNTT, include: in-memory computation such as CryptoPIM [NGI+20] and MeNTT [LPY22]; optimized GPU-based implementations such as [DCH+21]; as well as RISC-V architecture extensions for NTT such as [PS22] and [KA20].

3 BACKGROUND

3.1 Number Theoretic Transform (NTT)

Let N be a positive integer and $\mathbb{F}$ a field containing a primitive N-root of unity, denoted as w_N. That is, $w^N = 1$ in the field, but $w^k \neq 1$, for $0 < k < N$.

The Discrete Fourier Transform (DFT) is a linear transformation $F_N : \mathbb{F}^N \to \mathbb{F}^N$ defined for $\mathbf{y} = F_N(\mathbf{x})$ by:

$$y_i = \sum_{j=0}^{N-1} \omega_N^{ij} x_j$$

When $\mathbb{F}$ is a finite field of characteristic $p > 0$, the DFT is usually called Number Theoretic Transform (NTT). Note that in practical implementation, a DFT over $\mathbb{C}$ will be an approximation, whereas in the case of NTT, we are interested in exact solutions.

A naive implementation of DFT has complexity $O(N^2)$. An efficient algorithm to compute DFT with complexity $O(N \log N)$ or better is referred to as Fast Fourier Transform (FFT). For practical applications, we typically set $N = 2^n$ (radix-2 FFT/NTT), extending the x_i with zeros.

Since the w^{ij} are N^2 coefficients, but the w^k are only N distinct numbers, there is periodicity in the matrix coefficients. This led to a number of "factorization lemmas", of which the Cooley-Tukey framework [CT65] is the most well-known and fundamental.

When N factors in $n_1 n_2$, the Cooley-Tukey framework decomposes a given order N DFT into n_2 smaller order n_1 DFTs, and n_1 smaller order n_2 DFTs, combined with so-called *twiddles*, which are proper powers of ω_N. It directly follows that the complexity reduces from N^2 to $n_1 n_2^2 + n_1^2 n_2 < N^2$.

By repeatedly factoring N, one gets to smaller DFTs of prime order. These "minimal" FFTs (combined with their twiddle multiplications) are called *butterflies*. In particular, if N is a power of 2, factoring all the way leads to repeated order 2 DFTs, for a total complexity of $N \log N$.

Two common ways to compute a FFT are *decimation in time* (DIT) and *decimation in frequency* (DIF). These represent the two cases of factoring N "from the left" or "from the right": DIF when n_1 is a small radix, DIT when n_2 is a radix. We stress that DIT and DIF differ both on 1) how they "loop" over the input values, and 2) the butterfly function. We add that it is not necessary and, as we will see, not always efficient to factor "all the way down" - repeated higher-order (also called higher *radix*) factorsr educe the number of accesses to memory.

3.2 Goldilocks Field

While our design is generic and can be adapted to any field, including complex numbers, our current implementation focuses on the field $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1 =$ 0xFFFF_FFFF_0000_0001, called Goldilocks field [Gou21, Tea22].

Elements $a,\ b \in \mathbb{F}_p$ can be represented as 64-bit (unsigned) integers. Common field operations are extremely efficient on 64-bit CPUs, including:

- $a \pm b \mod p$, as a single addition/subtraction, optionally followed by subtracting/adding $\varepsilon = 2^{32} - 1 =$ 0xFFFF_FFFF in case of overflow/underflow.
- $a \cdot b \mod p$, as a single $64 \times 64 \to 128$-bit multiplication followed by reduction mod p, that can be done efficiently (a handful of assembly instructions), again using the sparse representation of p.

Algorithm 1 Radix-2 gNTT

```
1:  function gNTT(values, logN, twiddles)
2:      values are expected in bit-reversed order
3:      N ← 2^logN
4:      for i = 0 to logN do                          ▷ DIT loop
5:          m ← 2^(i+1)
6:          for k = 0 to N step m do
7:              ω ← twiddles[k/m]      ▷ one twiddle in inner loop
8:              for j = 0 to m/2 do
9:                  e ← values[k + j]
10:                 o ← values[k + j + m/2]
11:                 values[k + j] ← e + o             ▷ DIF butterfly
12:                 values[k + j + m/2] ← ω(e − o)
13:             end for
14:         end for
15:     end for
16:     return values
17: end function
```

3.3 gNTT Algorithm

In [BLDS10], Bowers *et al.* introduce the gFFT algorithm to compute DFT (corresponding to a factorization they call G). Their key observation is that it is possible to re-arrange the computation and drastically reduce the number of memory accesses to the twiddle factors, asymptotically from $O(N \log N)$ in the classical Cooley-Tukey down to $O(N)$.

Surprisingly, the resulting algorithm looks exactly like a Cooley-Tukey DIT, but with butterflies that resemble those of a DIF butterfly. The derivation is independent from the base field, therefore it also applies to NTT.

We call gNTT the analogous algorithm for NTT, and we present a radix-2, in-place, iterative version in Alg. 1. This is at the core of CycloneNTT. Note the DIT loop starting at line 4 (therefore this implicitly requires the input values to be sorted in bit-reverse order), and the DIF butterfly in line 11-12. The key feature of gNTT is the twiddle ω at line 7, which is constant throughout the inner loop.

As we will see in more detail, NTT is generally a memory-bound computation and CycloneNTT, in some instances, results in a compute-bound one. Reducing the number of twiddles accesses via gNTT is the first step towards achieving this goal.

We also highlight that gNTT is often more efficient than Cooley-Tukey even for software implementations. For example, as a corollary of this work, we implemented gNTT in the open source library `plonky2`[1], resulting in a 10-15% performance improvement.

4 CYCLONENTT

In this section we present CycloneNTT architecture and its unique properties. It performs in-place operations on the input values. For the sake of this study, input data is assumed to be stored in off-chip memory, although streaming data to the FPGA from the host can be easily accommodated.

The following lists key design goals for CycloneNTT:

- All computation performed entirely within the FPGA and DRAM directly connected to the FPGA.
- Support large datasets that do not fit in on-chip SRAM.
- Avoid random-access to off-chip memory.
- Provide configuration to the designer to trade-off power for delay.

[1] https://github.com/mir-protocol/plonky2/

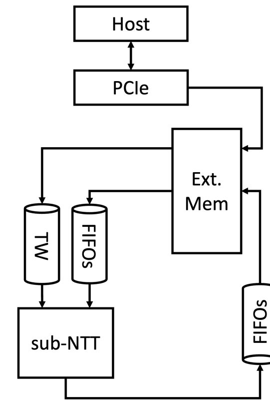

Figure 1: CycloneNTT System Architecture and Dataflow

4.1 Architecture

The overall system architecture is shown in Fig 1. The system is composed of the following main components:

- External memory, which is assumed to be DRAM based (DDR, HBM).
- Multiple large FIFOs for streaming the dataset in and out, backed by on-chip and off-chip memory.
- Sub-NTT module which can be either a single array of butterfly units, or a multi-layered network of butterfly units connected just like a regular NTT network.
- Some connectivity to a host, either through PCIe or network, to load twiddle factors and input data into the external memory.
- A host processor to initialize the system, with no interaction during the computation.

In Sec. 5 and 6 we present two instances of this architecture that primarily differ on the Sub-NTT module and how data is streamed from/to external memory.

The first instance, Single-Layer Streaming, computes NTT by moving values in and out of external memory $\log N$ times (e.g., 24 times). The second instance, Multi-Layer Streaming, improves on that by computing C layers at a time (e.g., $C = 3$, or $C = 6$), thus reducing the number of passes over the values to $\frac{1}{C} \log N$ (resp., 8 or 4, compared to the 24 for Single-Layer Streaming).

4.2 High-Level Protocol

Host and FPGA implement the following protocol:

ntt_init(N) Initialize the FPGA and twiddles.

1. Host pre-computes all N twiddles, sorted in bit-reverse order.
2. Host sends twiddles over PCIe.
3. FPGA receives the twiddles and stores them in the external memory.

ntt_send(values) Send input values into FPGA.

(1) Host sends the input values over PCIe, in bit-reverse order[2]
(2) FPGA receives the input values and stores them in the external memory.

ntt_compute() Compute NTT.
(1) Host sends `compute` command over PCIe.
(2) FPGA repeats multiple times:
  (a) Load twiddles and values into input FIFOs (from external memory into SRAM).
  (b) Process twiddles and values via Sub-NTT.
  (c) Send output values into output FIFOs (from SRAM to external memory).

Details depend on the architecture and are explained in Sec. 5, 6.

ntt_recv() → values Receive output values from FPGA.
(1) Host sends `recv` command over PCIe.
(2) FPGA returns output values from external memory over PCIe.

We would like to note that optimizing data transfer between the FPGA and the host is an orthogonal problem that depends on the actual platform, and the type of communication used (PCIe/Ethernet), hence we consider it out of the scope of this work.

4.3 Butterfly Unit

We design and implement a fully-pipelined DIF butterfly unit for the Goldilocks field $\mathbb{F}_p$, where elements are represented as 64-bit integers. Fig 2 shows the 8-stage pipeline design with an initiation interval of 1. The DIF butterfly computes:

$$e' = e + o$$
$$o' = \omega(e - o) ,$$

where $e,\ o$ are two input values, ω is the twiddle factor, and all operations are modulo q.

We employ multiple techniques as outlined in [Gou21] to reduce the complexity of the computations involved, including modular multiplication:

- Multiplication by the field's $\epsilon = 2^{32} - 1$ is simplified to a single subtraction as the constant is in the form of $2^n - 1$.
- Due to the unique properties of the Goldilocks field and its prime, the modular multiplication times ω can be reduced to a single 64-bit unsigned multiplication along with a series of 64-bit additions and subtractions.

5 SINGLE-LAYER STREAMING

In this version of the architecture, the NTT network is processed one layer at a time. For each layer, the entire dataset is read from off-chip memory, processed through an array of butterfly units, and the results are sent back to the off-chip memory, forming a *cyclone* of data streaming.

Along with the dataset, twiddle factors are also streamed in to be used by the butterfly array. This architecture streams in and out the entire dataset as many as the layers of the NTT network. For N numbers, the network has $\log N$ layers, hence the time complexity of this architecture is determined by the bandwidth of the off-chip memory to stream in and out the entire dataset $\log N$ times.

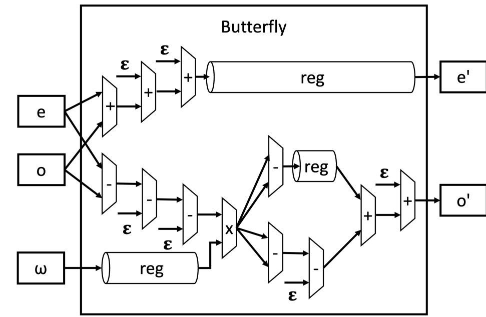

Figure 2: CycloneNTT butterfly unit optimized for Goldilocks field. This 8-stage pipeline outputs $e' = e + o$, $o' = \omega(e - o)$. e and o are "even" and "odd" inputs to the butterfly, ω is the twiddle factor, $\epsilon = 2^{32} - 1$ is used for modular reduction, and reg(s) are pipeline registers to align data in time.

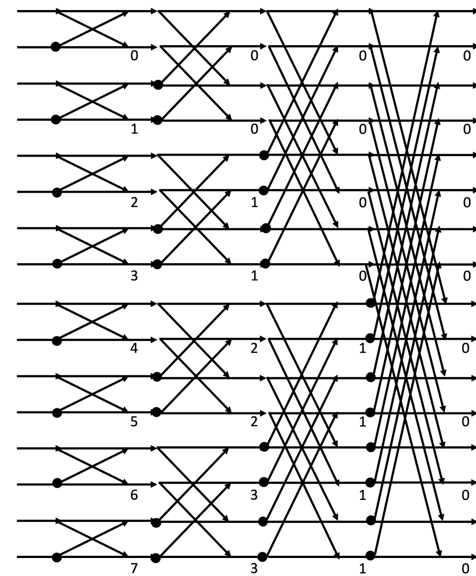

Figure 3: An 8-input butterfly network (starting from left), showing twiddle indexes used in each layer. All right-inputs are highlighted with a bubble.

5.1 Streaming Twiddles

Since we use [BLDS10]'s network, within a layer, twiddle factors are used in sequential order. As we progress through the layers, the number of twiddle factors needed is cut in half, as shown in Fig 3. However, the stream always starts from index-0. Therefore, for layer L, we simply stream in twiddles $[0 \dots \frac{N}{2^{L+1}})$ from off-chip memory.

We make an important observation here that as we progress through layers, the bandwidth required for streaming twiddles is

[2] Bit-reverse order indexes can be computed "on-the fly", and Host can access input values in RAM, in bit-reverse order. This seems more efficient than in-place reordering.

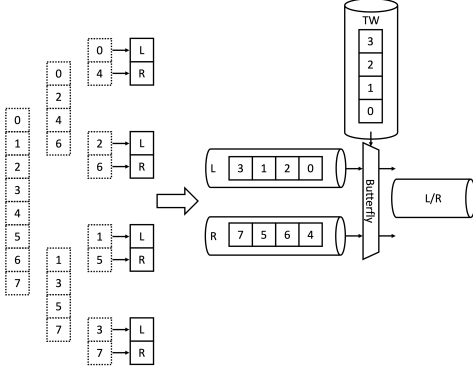

Figure 4: Input data is first bit-reverse reordered and placed in two FIFOs. The butterfly unit pops data from both FIFOs and twiddle FIFO, but pushes data only into one FIFO

cut in half. This is not only because half the twiddles are used, but also due to the fact that in layer L, 2^L neighboring butterflies use the same twiddle factor. This provides opportunity for twiddle reuse, given we process butterflies in a sequential order within the layer.

5.2 Parallel Butterflies

We note that the data dependency in the butterfly network is only inter-layer. Within a layer, we can execute in parallel as many butterflies as we can afford to feed input data to, and fit the circuit on the chip. In this single-layer CycloneNTT architecture, we employ an array of B fully-pipelined butterflies in parallel. In order to achieve maximum throughput, we need to feed the butterfly array every cycle to avoid bubbles. Therefore, we require $2 \times B$ numbers every cycle to be read from off-chip memory, regardless of the layer we are processing.

It should be noted that this architecture is memory bound by design: for every butterfly unit in the system, two numbers and a twiddle factor need to be read from off-chip memory.

5.3 *2*-FIFO Architecture

As demonstrated in Fig 3, there exists only inter-layer data dependency in the butterfly network. Furthermore, every output of a butterfly feeds only a single butterfly, meaning that a single butterfly feeds two butterflies in the next layer. However, we make a further important observation that a single butterfly, depending on its position in the layer, either feeds the left input or the right input of its target butterflies, as highlighted in Fig 3. We use this observation to propose a 2-FIFO architecture as shown in Fig 4.

Every butterfly can be seen as reading its inputs from two FIFOs, *left* and *right*. However, both its outputs will be pushed into either left, or right FIFO (Fig 4). In addition, as evidenced in Fig 3, the two outputs are 2^L rows apart, as their target butterflies in the next layer are 2^L apart. For a given butterfly in position (P) in layer (L), we have:

$$
\begin{aligned}
O - FIFO : left/right &= \lfloor \frac{P}{2^L} \rfloor \mod 2 \\
O - FIFO : positions &= 0, 2^L
\end{aligned} \tag{1}
$$

We start by placing every other input number into left and right FIFOs (Fig 4). Processing every layer of the network amounts to reading $\frac{N}{2}$ numbers from each FIFO (N total), running them through the array of butterflies in parallel, and writing the results back to their corresponding FIFO.

5.4 FIFO Memory

The two FIFOs employed in this CycloneNTT architecture are backed by both on-chip and off-chip memory. We construct the FIFOs as ring buffers in off-chip memory, and read from off-chip memory directly. However, when pushing into the FIFOs, we first store the data and its position in the FIFO in a small, on-chip buffer. When enough data has been buffered to fill an entire DRAM row, we flush the buffer to off-chip memory at the corresponding ring address. Flushing data at DRAM row granularity ensures the lowest overhead due to the internal structure of the DRAM, yielding the highest throughput possible.

6 MULTI-LAYER STREAMING

The single-layer architecture proposed in Section 5 requires a single butterfly unit per two input data as we arrange the units in a one-dimensional array. Therefore, if off-chip memory bandwidth allows for streaming in maximum of $2B$ elements per cycle, we can only utilize B butterfly units. However, if the chip's capacity allows for more butterfly units, we have a memory bound architecture.

To overcome the under-utilization of our computing resources, we can arrange the butterfly units into a sub-network of butterflies rather than a vector. This sub-network is similar in shape to the overall butterfly network shown in Fig 3, it is only smaller in size. A sub-network with B butterfly units *in each layer* (total of $\log(2B)$ layers by design) requires $2B$ inputs, and produces $2B$ outputs, hence its memory bandwidth requirements are the same as in the single-layer architecture.

Said in another way, if we have bandwidth to stream $2B$ elements per cycle, then we can construct a sub-network with B butterfly units per layer, that sequentially processes $\log(2B)$ layers, for a *total* of $B \log(2B)$ butterfly units. In return, we will only need to stream the dataset from/to external memory $\frac{1}{\log(2B)} \log N$ times.

Just to provide a concrete example, we can instantiate a 6-layer architecture with $B = 32$ butterflies per layer, or a total of $6 \times 32 = 192$ butterflies. This architecture will only need to stream the dataset from/to external memory $\frac{1}{6} \log N$ times.

6.1 Streaming Twiddles

In multi-layer processing, we process a sub-network of the overall network at a time. This means that for every sub-network execution, we require twiddle factors for all the $\log(2B)$ layers at the same time. Consequently, we require higher memory bandwidth to stream in twiddle factors. On a positive note, recall that the bandwidth requirement for twiddle factors halves every layer into the network. Therefore, for a sub-network width B, the overall bandwidth requirement for the twiddle factors is only $\Sigma_0^{B-1} \frac{1}{2^i}$, i.e. < 2 times larger compared to what is needed for single-layer processing, regardless of the value of B.

6.2 *2B*-FIFO Architecture

In the case of single-layer processing, we employ 2 FIFOs. As for multi-layer processing, we require larger number of FIFOs to be able to feed all the sub-network inputs at the same time. Fortunately, HBM-based FPGA platforms provide a large number of ports to off-chip memory banks (e.g. 32). Furthermore, each port is significantly wider (typically 512-bits) than Goldilocks field's numbers (64-bits). Since we stream in data from all FIFOs at the same time, we can combine multiple FIFOs (512/64=8) into the same memory bank to be able to access all FIFOs in parallel.

As with single-layer architecture, the sub-network is fed from all FIFOs in parallel, and all outputs of the sub-network are directed to a single FIFO, identified by the position (P), layer (L), and size of the sub-network (B):

$$\begin{aligned} O-FIFO: index &= \lfloor \frac{P}{2^L} \rfloor \mod 2B \\ O-FIFO: positions &= \{i \times (2B)^L; i \in [0, 2B)\} \end{aligned} \tag{2}$$

6.3 Output Buffer Analysis

As stated in eq. (2), egress FIFO index switches every $(2B)^L$ beats, yet the sub-network is fed from all FIFOs in parallel. Therefore, for every FIFO we need a buffer to absorb the rate disparity until the egress FIFO switches to the next index. However, as we advance through network layers (L), egress FIFO changes very infrequently up until the very last layer, where we would require a buffer as large as $B \times \frac{N}{2B} = \frac{N}{2}$ numbers. Since we target very large datasets, this is not a practical buffer size we can afford on-chip. In the following subsections we show how we solve this issue by employing a technique we call *quasi-streaming*.

6.4 Quasi-Streaming

The primary reason behind streaming, rather than random access of, the data to/from off-chip memory is the internal structure of the DRAM architectures. In a typical DRAM (DDR/HBM), accessing a row of the data is expensive, hence the memory is arranged in a very wide format. To minimize overhead, applications must strive to access and consume an entire row before moving on to the next. Therefore, streaming data within a row is very beneficial. However, we make a key observation that if an application's access quantum is an entire DRAM row, for such application random access to different rows yields the same throughput as streaming rows sequentially.

In Fig 5 we show the throughput we achieve by accessing random and sequential addresses, while varying access size. As can be seen, streaming data reaches maximum throughput with access size of 256-bytes, however random-access only eventually reaches the same throughput with access size of 4096-bytes or higher. This demonstrates that when accessing a DRAM one-row-at-a-time, the application can view the DRAM as a truly random-access-memory, and no consideration needs to be made for sequential addressing.

Using this observation, we propose exploiting quasi-streaming in CycloneNTT to set the output buffer capacity of FIFOs to a few multiples of a DRAM row, irrespective of B. Since we can random access ingress FIFOs at DRAM-row granularity, we strategically choose a sequence of addresses to read and feed the sub-network

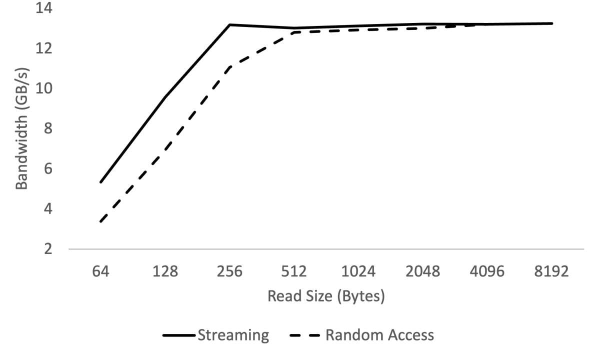

Figure 5: Bandwidth comparison between streaming and random-access of various number of bytes. This data was captured on C1100 platform using HBM channel-0.

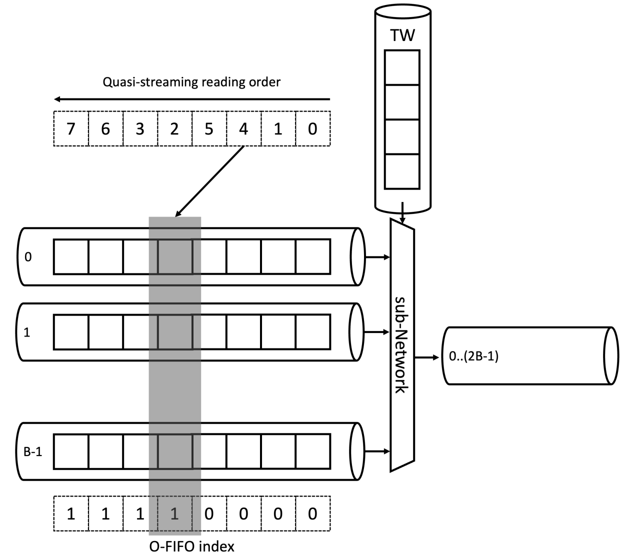

Figure 6: Quasi-Streaming read order to change O-FIFO index every two beats. In this example we highlight reading index-4 out of order leads to output FIFO index-1.

that produces frequently-changing egress FIFO indexes. Fig 6 shows an example of such sequence for quasi-streaming of data, that enables switching egress FIFO index quickly. Alternatively, for a large enough B (≥ 8), which leads to reading one or more DRAM rows at a time, one can conveniently choose B to be the quantum of FIFO reads, simplifying the read sequence further. It should be noted that in this architecture, the elements stored in the FIFOs are not popped sequentially, but rather with pre-defined read addresses.

6.5 Power vs Delay

CycloneNTT is a configurable architecture. The primary parameter to tune is the number of inputs to the sub-network (B), which also determines the number of FIFOs ($2B$). As we grow B, fewer passes

	Layer-1	Layer-2	Last Layer ($\log N$)
Twiddles	1	$\frac{1}{2}$	$\frac{1}{\log N}$
FIFO-0	1	1	1
FIFO-1	1	1	1

Table 1: Relative bandwidth requirement for each port

of the dataset is required ($\frac{\log N}{\log 2B}$), lowering execution time. However, sub-network size grows proportional to $B \log 2B$, resulting in increased power dissipation. Depending on the application and environmental properties, one can choose the right B to create the best fitting solution.

Another major design decision to consider is clock frequency. Depending on the platform, after a certain clock speed the memory interface will be saturated, and increasing the clock speed returns no discernible gains. On the other hand, increases in clock speed result in more power dissipation. For example, on the C1100 HBM-based platform [AX21, AX22], each port is rated at the theoretical maximum speed of 14.2GB/s, excluding DRAM timing overheads. Considering the 512-bit interface to the memory, this translates to a maximum clock speed of 222MHz, after which the memory interface is saturated.

7 MEMORY INTERFACE

In this section we discuss the architecture and complexity of the memory interface needed for reading and writing to off-chip DRAM storage. CycloneNTT uses DRAM to store twiddle factors, input vectors, and the back-end for the 2B FIFOs employed.

7.1 Single-Layer

In this section we discuss the relatively simpler memory interface needed for the single-layer CycloneNTT. As we only require two FIFOs, we can simply allocate one memory port per FIFO as shown in Fig 7. In addition, we require a third memory port to access twiddle factors. Table 1 shows the relative bandwidth requirement per port in each layer. For example, when processing the first layer of the entire network, at every beat for every butterfly, we require one number from each FIFO, along with one twiddle factor. Therefore, while processing the first layer, the bandwidth requirement is the same for all ports.

7.1.1 Twiddles. As described in Sec 3.3 our butterfly network uses twiddles in a streaming fashion, in bit-reverse order. Therefore, at initialization time we store all twiddle factors into the corresponding memory bank, in bit-reverse order. During the computation of layer l we simply stream in twiddles $[0 \dots \frac{N}{2^l})$ as required by the network.

7.1.2 Reads. At least in the case of Goldilocks field and our targeted platforms, the memory interfaces are wider (512-bits) than the individual numbers (64-bits), and we expect this will be the case for most applications. Consequently, every read from the memory provides multiple numbers. This provides the opportunity to process multiple numbers at the same time using a vector of butterflies, as explained in Section 5.

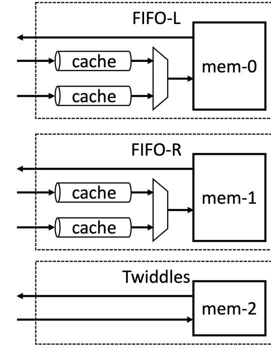

Figure 7: Memory interfaces used in single-layer streaming connecting left/right FIFOs and providing twiddle access.

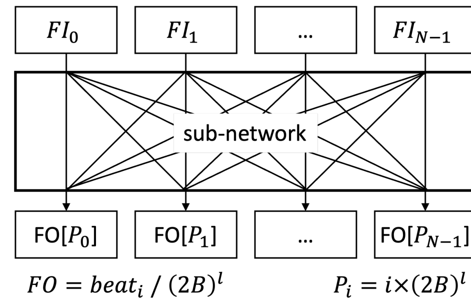

Figure 8: Output FIFO index and position in the FIFO based on the beat-index in each layer $l \in [0 \dots \log N)$.

7.1.3 Writes. As shown in Fig 3 and Eq 1 both outputs of the butterfly unit are directed to the same FIFO, either left or right. In addition, the two outputs are not placed sequentially, and are separated by Eq 1. Consequently, for each write port, we require two (one for each butterfly output) small SRAM-based caches to hold each butterfly output as they become available. This caching is necessary as writing individual numbers to DRAM yields very low bandwidth utilization.

7.2 Multi-Layer

The multi-layer configuration of CycloneNTT improves memory bandwidth utilization while presenting new challenges with regards to data write backs.

Similar to single-layer, memory ports are assumed to be wider than individual numbers. However, unlike single-layer, in the same cycle we cannot consume multiple numbers from the same FIFO as that would require creating a vector of sub-networks which would be prohibitively expensive. Instead, we map multiple FIFOs into the same memory port, which results in one number per FIFO per read-cycle throughput as we require. Considering our targeted field and platforms, each port can accommodate $\frac{512}{64} = 8$ FIFOs.

7.2.1 Reads. Each memory port is independently fetching numbers from its allotted eight FIFOs (or input vector for the first round) in a quasi-streaming fashion. To minimize read overhead, read requests are batched into D beats, yielding all numbers required for D

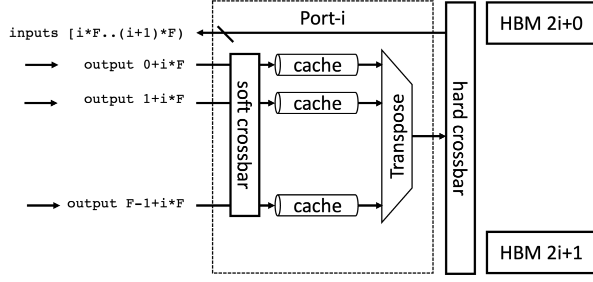

Figure 9: Logical and physical memory ports in multi-layer architecture. Each cache is implemented with a single blockram.

beats of the sub-network. D is chosen in a way that $512 \times D \geq$ DRAM-row. Next, read addresses follow a FIFO-first scheme to provide a fast switching output FIFO as per Eq 2 and Fig 8. For the first and last layers, output FIFO either switches with every beat or does not switch at all, therefore read addresses end up being sequential. However, starting from the second layer, each port requests D beats, then jumps the gap to the next output FIFO. In practice, this is done with carefully decomposing the read address into limbs and cycling through them out of order. This is very similar to DRAM row/bank/bank-group interleaving, with the added complexity that the bit width allocated for each part changes as we progress through the layers. Nevertheless, this yields a compute-heavy, yet relatively simple, circuit to determine read addresses.

7.2.2 Writes. Writing data back to DRAM is substantially more complicated compared to reads. We have to overcome two challenges: a) Multiple FIFOs need to be combined into the same port, b) As evident in Fig 8, the output data needs to be transposed.

In order to combine multiple outputs into the same port, we need an SRAM-based cache to store the results as they become available. The cache is as wide as the sub-network, and requires $F \times D$ rows where $F = 8$ in this case. It is only after every $F \times D$ beats that we have the data for all FIFOs allotted to each port, at which time we can write them to the memory after transposing them.

7.2.3 Twiddles. Similar to the single-layer architecture, twiddles have the same bandwidth requirement as numbers in the first layer. We opt to allocate the same number of ports for twiddles as for numbers, which is $\frac{2B}{F=8}$.

7.3 Data Transposition

Row-Column data transposition is required starting from the second layer. This is shown in Fig 8 as sequential numbers for each FIFO are spread across beats. The challenge here is each write port cache needs to support writing the entire beat (2B numbers) every cycle as they come out of the sub-network, yet it needs to be able to transpose $F = 8$ beats into a write word for DRAM, also every cycle. The first operation requires a wide memory arrangement, while the second operation requires multiple read ports.

Fortunately, individual numbers being written (64-bits) are about the same width as typical blockrams found in FPGAs (36/72 bits). We propose the architecture shown in Fig **??**. Each column is stored into a separate blockram with acceptable storage waste. To provide multiple read ports for data transpose, we carefully rotate the data through blockram columns as they are being written. In this way, no two numbers belonging to the same column are ever written to the same blockram, hence we can read them in parallel from each ram. This architecture requires $2B$, $2B \times 1$ multiplexers inside the soft-crossbar, and F, $2B \times 1$ multiplexers for data transpositions. For $B > 4$ this can yield to long critical paths, hence we opt to use a 2-cycle multiplexer design.

	Number of Ports	Utilization
Single Layer - F1	3	$\frac{3}{4} = 75\%$
Single Layer - C1100	3	$\frac{3}{32} = 9\%$
3-Layer - C1100	3	$\frac{3}{32} = 9\%$
4-Layer - C1100	6	$\frac{6}{32} = 18\%$
6-Layer - C1100	24	$\frac{24}{32} = 75\%$

Table 2: Memory port usage and overall memory bandwidth utilization of the platform.

7.4 Simultaneous Reads and Writes

At any given time, we are reading from all FIFOs to feed the sub-network, and writing back its outputs to FIFOs. In order to avoid the read/write clash into the same memory bank, we allocate two HBM banks for each logical memory port of the architecture, as shown in Fig 9. Across layers, we ping-pong between the underlying layers for reads and writes, hence no HBM port is ever used for reads and writes at the same time. This comes at the expense of extra memory ports used.

Accessing adjacent HBM memory ports comes with almost no overhead in HBM architectures. HBM provides a hardened, low-latency crossbar that provides a high speed all-to-all configuration only for adjacent ports. Therefore, ping-ponging between two ports for each logical port amounts to changing the MSB of the address bit, and using the same port for reads and writes with no multiplexers required in the fabric.

7.5 Memory Interface Utilization

Following the memory interfacing explained in the previous sections, table 2 reports the number of memory ports used and overall utilization of the platform memory interfaces, under various CycloneNTT configurations. Note that the reported number includes ports needed for accessing numbers and twiddle factors. Overall, in the case of multi-layer architecture, $3 \times \frac{2B}{F=8}$ ports are needed, making the 6-layer architecture the largest a 32-port HBM platform can support.

8 EVALUATION

In this section we evaluate CycloneNTT on two different platforms with various design parameters. We implement all networks with Goldilocks butterfly units, however any number system can be substituted. For example if the butterfly unit is replaced by one that handles complex numbers, the entire system would calculate the FFT of input data.

Platform	Memory Type	Single/Multi-Layer
AWS-F1	DDR	Single
C1100	HBM	Single 3 (B=4) 4 (B=8) 5 (B=16) 6 (B=32)

Table 3: All configurations used for evaluation.

Platform	Layers	Total Power (W)	Design Power (W)
AWS-F1	Single	35.243	0.906
C1100	Single	28.45	1.727
	3	27.766	1.265
	4	35.071	3.919
	5	48.800	14.739
	6	52.752	23.499

Table 4: Power dissipation as estimated by Vivado.

8.1 Methodology

Table 3 lists all CycloneNTT configurations used in this work. The focus of this work is on the C1100 platform as is the more suitable platform for such application. However, to demonstrate the portability of our design, we showcase a single-layer configuration on AWS F1 platform as well.

We implement the entire system in SystemVerilog RTL, and have made the code publicly available on GitHub, however for anonymity we omit the link until after publication. We value the reproducibility of all the results presented here, and chose platforms that are accessible to most designers.

We report five metrics for each configuration of CycloneNTT:

- Resource Utilization: We report LUTs, registers, BRAMs, URAMs (if any), and DSPs used in each configuration.
- Power: We rely on Vivado's power report to get an estimate of the power dissipation of the design. It should be noted that these are estimates by the tool, and actual energy consumption can vary at runtime due to input data and environmental properties, and would require live measurements.
- Clock Speed: We report the clock speed achieved for each design. As mentioned in Section 6.5, pushing the clock speed beyond a certain point provides no discernible gains and will only result in higher power dissipation.
- Latency: We report the time it takes to process the entire NTT network. We exclude data transfer times between the FPGA and the host, as we find data transfer optimization to be orthogonal to this work, and can vary depending on the platform, e.g. in a host-less, network-attached appliance the interface speed can vary.
- Throughput: To facilitate comparison across platforms and configurations, and with prior art, we report the throughput of the design in terms of millions-of-numbers-per-second.

8.1.1 AWS-F1. This cloud platform provides access to cards with AMD-Xilinx VU9P FPGAs [AWS]. The card has four independent DDR channels, each with 16GB capacity and a theoretical throughput of 16GB/s. The interface exposed to the design is 512-bits wide, and would be saturated at 250MHz. On this platform, we rely on Amazon provided shell for all communications with the host and DDR. We find the platform and shell easy to use, taking about 20% of the VU9P chip's resources. The use of this platform in this study is to demonstrate CycloneNTT's performance on a DDR-based platform where only a few number of memory ports are available.

Platform	Layers	LUTs	Regs	BRAM	DSP
AWS-F1	Single	11255 (10%)	15325 (10%)	54 (10%)	160 (10%)
C1100	Single	20234 (2.57%)	23061 (1.46%)	320 (5.66%)	54 (4.54%)
	3	16771 (1.9%)	16998 (1.0%)	144 (2.4%)	29 (2.2%)
	4	53026 (6.7%)	46026 (2.9%)	384 (6.8%)	74 (6.2%)
	5	170099 (20%)	121075 (6.9%)	960 (16%)	212 (16%)
	6	563677 (65%)	319104 (18%)	2304 (39%)	691 (51%)

Table 5: Resource utilization count and the percentage of the platform used for various CycloneNTT configurations.

8.1.2 C1100. This is a PCIe card provided by AMD-Xilinx directly, and is curated towards cryptocurrency mining [AX21]. This platform is equipped with a VU55P HBM-based FPGA. This FPGA has 32-HBM channels, each with 2Gbits of capacity, and 14GB/s of theoretical bandwidth. HBM channels are almost independent [AX22]. The interface exposed to the design is 512-bits wide, and would saturate at 222MHz. On this platform we use Vitis/XRT environment for development and deployment. However, we use pure RTL kernels, and only rely on the platform for communication to the host and HBM.

8.2 Results

In this subsection we discuss our findings regarding CycloneNTT architecture over four metrics.

8.2.1 Power. Table 4 reports estimated power dissipation for various CycloneNTT configurations on both platforms. Since we compare power profile on two different platforms, we include power report for the entire chip and the design. The single-layer architecture consumes more energy compared to the 3-layer design. This can be explained by its inefficient use of resources as shown in the next section. The 6-layer architecture demonstrates a super-linear power increase with respect to B. As reported in the next subsection, this design has a significantly higher resource utilization, hence the significant increase in its power dissipation.

8.2.2 Resource. In Fig 5 we report the resource utilization of various CycloneNTT configurations. The single-layer architecture demonstrates a clear inefficiency in using resources, as it consumes more than a 3-layer architecture. However a super-linear growth is

Platform	Single/Multi-Layer	Clock Speed (MHz)
AWS-F1	Single	250
C1100	Single	300
	3	300
	4	300
	5	176
	6	161

Table 6: Maximum clock speed attainable for each configuration.

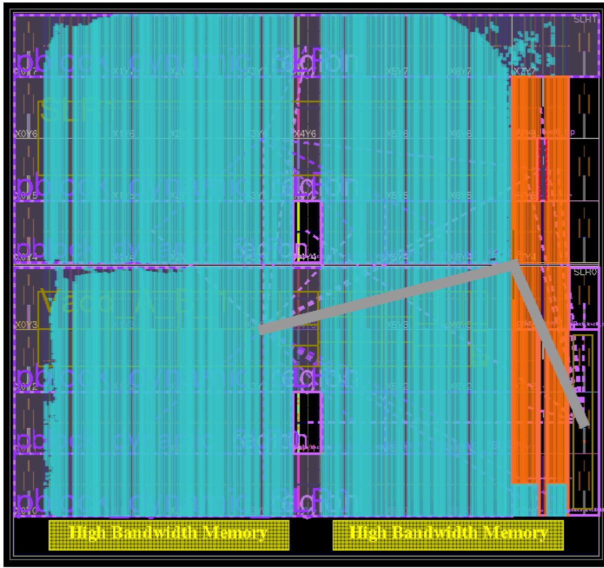

Figure 10: The final placement of a 6-layer CycloneNTT on C1100 platform.

visible with increasing the number of layers in the case of multi-layer architectures.

As discussed in Section 6.3, in the case of multi-layer architectures, buffers as wide as $2B$ and as high as DRAM row are required to absorb the rate disparity between ingress and egress of the FIFOs. However, as discussed in Section 7 a separate buffer per memory write port is also required, which also scales with respect to B. As a result, and as evidenced in these results, BRAM usage grows exponentially with respect to the number of layers.

The only DSP usage in the design is inside the butterfly units. The single-layer design requires B butterfly units multiplied by the butterfly vector size (=8 in our case), whereas the multi-layer design requires $B \log 2B$ units. Considering DSPs only, the largest CycloneNTT architecture to fit in C1100 platform would be 8-layers. However, due to excessive wires required inside the sub-network of butterfly units, and to connect the soft crossbar required inside write caches, as described in Section 7, a 6-layer ($B = 32$) is the largest configuration that fits on the C1100 platform. In Fig 10 shows the final placement of a 6-layer CycloneNTT on C1100.

Layers	2^{18}	2^{20}	2^{21}	2^{24}
3	655-1310	-	6116-12233	55924-111848
4	-	1092-2184	-	20971-41943
5	-	732-1464	-	-
6	64-129	-	-	5518-1137

Table 7: Execution latency lower and upper bound, in microseconds, for various architectures with applicable input sizes.

8.2.3 Clock Frequency. Fig 6 reports the maximum clock speed achieved for each design configuration. Given the relatively flat architecture of CycloneNTT, the clock speed is not significantly affected by the number of layers used in the sub-network. However, the routability of the design is affected, as the chip simply runs out of wires to connect all the butterfly units for 7-layers or larger designs. Although having a higher clock frequency is generally desirable, however due to memory bandwidth constraints, the overall system performance gain is negligible after a certain point. In addition, when power dissipation is considered, one can choose to lower the clock speed to lower power profile of the design, at minimal expense of performance.

8.2.4 Estimated Latency. Given the clock frequency of each configuration, we can estimate the latency of a multi-layer design for a given input size. In an L layer design, with input size of N, all numbers are read from the external memory and written back $\frac{N}{L}$ times. Given we employ $2B = 2^L$ FIFOs in parallel, the lower bound for the number of cycles it would take to cycle through the numbers is $\frac{N}{L} * \frac{N}{2^L} * period$. However, due to write caching required for data transposition, the entire process can be delayed. In the worst case scenario, all reads and all writes happen sequentially, which gives us the upper bound for execution time of two times the lower bound. In Table 7 we report the range for execution latency of various configurations with various input sizes.

8.2.5 Latency. Table 8 compares the execution time of different designs for different dataset sizes. As expected, the single-layer architecture is the slowest, as it spends $\log N$ passes over the entire dataset. In comparison, multi-layer architectures significantly cut down the latency by processing multiple layers at the same time. Of course as evidenced before, this speed gain comes at the relatively significant cost of power and resources. Comparing Table 7 with Table 8 we see that all obtained numbers fall between the estimated range.

8.2.6 Throughput. Fig 11 shows the improvement in throughput as we add layers to CycloneNTT. The exponential growth with respect to the number of layers is evident. Here we also compare CycloneNTT with prior works. [KMG+19] demonstrates a throughput of 233 million per second for one million points, as that is the largest that can fit in their platform's on-chip memory. [YKKP22] is an NTT generator platform, but does not accommodate a million point NTT. We use their largest sample size (2^14) and assume a linear scaling to calculate their equivalent million/sec throughput. We should note that CycloneNTT utilizes external memory to accommodate large data sets while providing superior throughput.

Layers	2^{18}	2^{20}	2^{21}	2^{24}
Single	3536	16806	36388	358937
3	1137	-	10859	101589
4	-	2117	-	42570
5	-	1036	-	-
6	101	-	-	8083

Table 8: Execution latency, in microseconds, for various architectures with applicable input sizes.

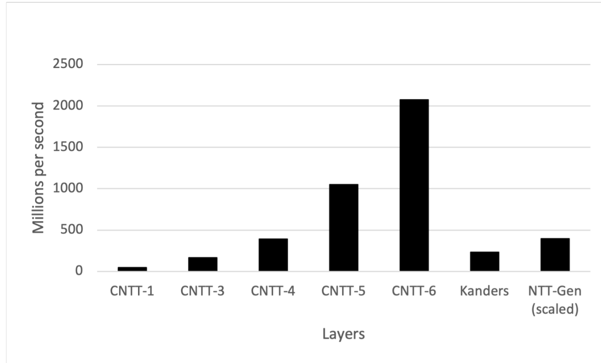

Figure 11: Throughput in millions of numbers per second processed by each architecture and prior work.

9 CONCLUSION AND FUTURE WORK

CycloneNTT is a hardware solution for computing NTT on large datasets ($\geq 2^{24}$, 64-bit numbers) that require external memory. By applying a series of algorithmic- and implementation-level optimizations, CycloneNTT achieves a quasi-streaming data access pattern that maximizes throughput. The architecture is configurable and it has been applied to DDR- and HBM-based platforms. Moreover, the designer can trade off power for delay, depending on the target system and environmental conditions. To the best of our knowledge, CycloneNTT is the first architecture to tackle this problem for such large datasets in an efficient manner.

Future work relates to the current limitations on external memory, and it is two-fold. First, as datasets grow larger, even off-chip memory capacity becomes a limiting factor, which would require the deployment of multiple FPGAs. Efficient inter-FPGA communication through various channels (PCIe/Ethernet/etc.) will be the key points in achieving the same level of throughput as in a single FPGA solution. Second, DRAM technology suffers, in general, from high-latency, non-deterministic access latency, temperature and reliability issues (for a detailed analysis, refer to [JWW21]). An alternative here is to explore off-chip SRAM, which offers lower and deterministic access latency at the cost of lower capacity and bandwidth. Historically, off-chip SRAM has been limited to an order of magnitude lower capacity compared to DRAM. However recent process technology advancements have enabled higher capacity chips. The use of smaller, yet-faster, off-chip memory coupled with the deployment of multiple FPGAs, could provide higher system-level performance and energy efficiency.

Alternative platforms that may also be considered in a future work include, but are not limited to, AMD-Xilinx Versal AI Series (making use of their AI Engines) as well as Versal HBM series.

REFERENCES

[AWS] AWS. Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed: 2022-09-23.

[AX21] AMD-Xilinx. Varium C1100 compute adaptor data sheet, DS1003 (v1.0). https://docs.xilinx.com/v/u/en-US/ds1003-varium-c1100, September 2021. Accessed: 2022-09-23.

[AX22] AMD-Xilinx. Vitis unified software platform documentation: Application acceleration development (UG1393): HBM configuration and use. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use, May 2022. Accessed: 2022-09-23.

[BLDS10] Kevin J. Bowers, Ross A. Lippert, Ron O. Dror, and David E. Shaw. Improved twiddle access for fast fourier transforms. *IEEE Transactions on Signal Processing*, 58(3):1122–1130, 2010.

[CP15] Ren Chen and Viktor K. Prasanna. Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2015.

[CT65] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DCH+21] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–355, 2021.

[DM22] Kemal Derya, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication. *Microprocessors and Microsystems*, 89:104451, 2022.

[Gou21] A. P. Goucher. An efficient prime for number-theoretic transforms. https://cp4space.hatsya.com/2021/09/01/an-efficient-prime-for-number-theoretic-transforms, September 2021. Accessed: 2022-09-23.

[JWW21] Matthias Jung, Christian Weis, and Norbert Wehn. *The Dynamic Random Access Memory Challenge in Embedded Computing Systems*, pages 19–36. Springer International Publishing, Cham, 2021.

[KA20] Emre Karabulut and Aydin Aysu. RANTT: A RISC-V architecture extension for the number theoretic transform. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 26–32, 2020.

[KLC+20] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A. Rutenbar. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 56–64, 2020.

[KMG+19] Hans Kanders, Tobias Mellqvist, Mario Garrido, Kent Palmkvist, and Oscar Gustafsson. A 1 million-point FFT on a single FPGA. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(10):3863–3873, 2019.

[LPY22] Dai Li, Akhil Pakala, and Kaiyuan Yang. MeNTT: A compact and efficient processing-in-memory number theoretic transform (NTT) accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(5):579–588, 2022.

[MKO+20] Ahmet Can Mert, Emre Karabulut, Erdinc Ozturk, Erkay Savas, and Aydin Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, pages 1–1, 2020.

[MK20] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, Michela Becchi, and Aydin Aysu. A flexible and scalable NTT hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 346–351, 2020.

[NGI+20] Hamid Nejatollahi, Saransh Gupta, Mohsen Imani, Tajana Simunic Rosing, Rosario Cammarota, and Nikil Dutt. CryptoPIM: In-memory acceleration for lattice-based cryptographic hardware. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[PS22] Rogério Paludo and Leonel Sousa. NTT architecture for a linux-ready RISC-V fully-homomorphic encryption accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2669–2682, 2022.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1295–1309, New York, NY, USA, 2020. Association for Computing Machinery.

[SRTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. FPGA-based high-performance parallel architecture

for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.

[Tea22] Polygon Zero Team. Plonky2: Fast recursive arguments with PLONK and FRI. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf, September 2022. Accessed: 2022-09-23.

[YKKP22] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. NTTGen: A framework for generating low latency NTT implementations on FPGA. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 30–39, New York, NY, USA, 2022. Association for Computing Machinery.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. PipeZK: Accelerating zero-knowledge proof with a pipelined architecture. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 416–428. IEEE Press, 2021.