L13: Projects and FPGA Applications

Cong (Callie) Hao
callie.hao@ece.gatech.edu

Assistant Professor
ECE, Georgia Institute of Technology

Sharc-lab @ Georgia Tech https://sharclab.ece.gatech.edu/
• Possible Projects

• FPGA Applications
  o FPGAs in autonomous system – Zishen Wan
  o FPGAs in encryption – Jianming Tong
Possible Projects – Standard ML Algorithms

• **Standard Machine Learning: CNN, RNN, LSTM**
  
  o CNN for classification: AlexNet, VGG16, ResNet18
  o CNN for detection: Tiny-Yolo
  o Millions of references (PyTorch, Tensorflow, Keras, ...)

• **Steps:**
  
  o 1. Find the original Python model, run it (inference only)
  o 2. Translate to simple C++: we call it golden model (as a starting point and testbench)
  o 3. Optimize the C++
Possible Projects – Standard ML Algorithms

• **Standard Machine Learning: CNN, RNN, LSTM**
  
  o CNN for classification: AlexNet, VGG16, ResNet18
  
  o CNN for detection: Tiny-Yolo
  
  o Millions of references (PyTorch, Tensorflow, Keras, …)

• **Difficulty:** ★★★★★

• **Significance:** ★★★★★

• **Amount of work:** ★★★★★★★
Possible Projects – GNNs

• **Graph Neural Networks (GNNs)**

• **Steps**

  o Get yourself familiar with GNNs (will be a guest lecture discussing it)
    - A brief and high-level introduction to GNN:
      [https://www.youtube.com/watch?v=fOctJB4kVlM&t=9s](https://www.youtube.com/watch?v=fOctJB4kVlM&t=9s)
  
  o Visit Open Graph Benchmark (OGB): [https://ogb.stanford.edu/docs/home/](https://ogb.stanford.edu/docs/home/)
    - Find your favorite GNN from the leaderboard: [https://ogb.stanford.edu/docs/dataset_overview/](https://ogb.stanford.edu/docs/dataset_overview/)
    - Understand it, translate it, and optimize it!
Possible Projects – GNNs

• Graph Neural Networks (GNNs)

• Steps
  • Difficulty: ★★★★★★★
  • Significance: ★★★★★★★
  • Amount of work: ★★★★★★★
Possible Projects – Systolic Array in HLS

• Try to implement your own Systolic Array in HLS!
  o Can be matrix multiplication, can be convolution
  o Different stationary
Possible Projects – Systolic Array in HLS

• Try to implement your own Systolic Array in HLS!
  
  o **Sparse matrix multiplication** using Systolic Array

![Sparse Matrix Multiplication Diagram](image-url)
Possible Projects – Systolic Array in HLS

- Try to implement your own Systolic Array in HLS!
  - Can be matrix multiplication, can be convolution
  - Different stationary
  - Sparse matrix multiplication using Systolic Array

- Difficulty: ★★★★★☆☆☆☆
- Significance: ★★★★★★☆☆☆
- Amount of work: ★★★★★☆☆☆☆
Robotic Computing on FPGAs

Zishen Wan
zishenwan@gatech.edu

PhD Student
ECE, Georgia Institute of Technology

ICSRL @ Georgia Tech https://zishenwan.github.io
Introduction

• **Presenter: Zishen Wan**
  - PhD Student in Georgia Tech (2020 Fall~Now)
  - MS in Harvard University (2018~2020)
  - BS in Harbin Institute of Technology (2014~2018)

  Email: zishenwan@gatech.edu
  Homepage: https://zishenwan.github.io

• **Research Interest**
  - General: Computer Architecture and VLSI
  - Domain Specific Accelerator
  - Reliability and Security
  - In-memory Computing
  - Neuromorphic Computing

• **Professional Activities**
  - 2022 CRNCH PhD Fellowship
  - 2021 DAC Young Fellow
  - 2020 DAC Best Paper Award
  - 2020 CAL Best Paper Award
Robotics (Autonomous Systems)

Requirements:
• High Performance
• Low Power
• Real-Time Processing
• Reliability and Safety
• Security
• Reconfigurable at Runtime
• SWaP (Size-Weight-and-Power) Constraints

Robotics is on the rise and everywhere!

Need Custom Hardware!
Robotic Computing Flow

Perception → Mapping & Localization → Planning → Control
Robotic Computing Flow

Perception → Mapping & Localization → Planning → Control
Perception

- **Perception**: Processing sensor data to understand the environments around robots
- **Feature Extraction**
- **Temporal & Spatial Matching**
- **Object Detection**
- **Scene Understanding**
Robotic Computing Flow

1. Perception
2. Mapping & Localization
3. Planning
4. Control
Mapping & Localization

- Mapping & Localization is the process of perception information to understand where a robot is in the world
- Mapping – Octomap Generation
- Mapping – Pointcloud Generation
- Simultaneously Mapping and Localization (SLAM)

[Image of a map with labeled rooms: Foyer, Living Room, Hall, Kitchen, Dining Room]
Robotic Computing Flow

- Perception
- Mapping & Localization
- Planning
- Control
Motion Planning

- **Motion Planning** is the process of computing an action plan for a robot on the previously computed map.

- **Sample-Based Motion Planning**

- **Trajectory Optimization**

Rapidly Exploring Random Trees (RRT)
Robotic Computing Flow

Perception → Mapping & Localization → Planning → Control
Control

• Control is the process of executing a plan in the real world

• Feedback Control

\[
\begin{align*}
\sum e(t) & \rightarrow P K_c e(t) \\
\sum I K_c \int e(\tau) d\tau & \rightarrow D K_c \frac{de(t)}{dt} \\
\sum u(t) & \rightarrow \text{Plant/Process} \\
\sum v(t) & \rightarrow r(t)
\end{align*}
\]

This is the canonical PID controller!

• Model Predictive Control

- Past Inputs and Outputs
- Future Inputs
- Future Errors
- Cost Function
- Constraints

Model

Optimizer

Predicted Outputs

Reference Trajectory

Model Predictive Control
Robotic Computing Flow

Perception → Mapping & Localization → Planning → Control
Robotic Computing Flow

Perception → Mapping & Localization → Planning → Control

or

End-to-End Learning
End-to-End Learning

- End-to-End Learning process input sensor information and directly generate output action using a neural network
- Reinforcement Learning
- Imitation Learning
End-to-End Learning

- End-to-End Learning process input sensor information and directly generate output action using a neural network

- Reinforcement Learning

- Imitation Learning
Robotic Computing Need HW Acceleration!

- Take motion planning as an example: collision detection for each connecting path can be very expensive...!

[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. MICRO’16]
• Take motion planning as an example: collision detection for each connecting path can be very expensive...!

And if the obstacles move, we have to recompute!

[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. MICRO’16]
Robotic Computing Need HW Acceleration!

- Take motion planning as an example: collision detection for each connecting path can be very expensive...!

![Graph showing collision checking latency vs. number of edges in roadmap for different methods: GPU Hashsets, CPU Hashsets, FPGA Accelerator, and FPGA Projected.](image)

But with custom hardware, this can be accelerated!

[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. MICRO’16]
• GPUs/CPUs’ power consumption is orders of magnitude higher than requirements of resource-constrained robotic computing scenarios.

• GPUs/CPUs’ general-purpose nature leads to time inefficiencies (real-time requirement)

• and more vulnerable to cybersecurity threats (safety requirement).

• ASICs typically have the highest energy-efficiency, but their limited configurability has difficulty adapting to new robotic scenarios, as the robotic computing algorithms are still evolving very fast.
Why FPGA for Robotic Computing?

• FPGAs have a unique advantage –
  
  **Compared to GPUs/CPUs:**
  
  o Higher energy-efficiency
  o Lower power
  o Higher performance
  o Higher reliability

  **Compared to ASICs:**
  
  o Higher reconfigurability
  o Higher adaptivity
  o Faster time-to-market
Project Ideas

- Lots of opportunity in accelerating robotic computing on FPGAs...

**Perception**
- ORB (Oriented FAST and Rotated BRIEF)
- ELAS (Efficient Large-Scale Stereo Matching)
- OpenVINS
- Yolo, Fast R-CNN
- Event-based perception, e.g., SNN

**Mapping & Localization**
- Octomap
- Pointcloud
- SLAM (Simultaneously Mapping and Localization)
- VIO (Visual Inertial Odometry)
- Registration

**Planning**
- RRT (Rapidly Exploring Random Trees)
- RRT*, RRT-C
- PRM (Probabilistic Roadmaps)
- Trajectory optimization e.g., DDP, SQP, Interior-Point Method, SNOPT, IPOPT, etc
Project Ideas

• Lots of opportunity in accelerating robotic computing on FPGAs...

- Tracking controllers (PID, LQR)
- MPC (Model Predictive Control)
- Neural Network

- Control
- Other Research Opportunities

End-to-End

You can apply many FPGA techniques learnt from the class to robotic computing:
• Sparsity
• Data precision
• Quantization
• Data reuse
• Memory optimization
• Loop optimization
...
References

**Book**

**Survey Paper**

Free-download under GT network
References

Perception


Mapping & Localization


[ICCAD’20] Han, Y., Yang, Y., Chen, X. and Lian, S., “DaDu series-fast and efficient robot accelerators”.

[DAI’19] Liu, R., Yang, J., Chen, Y. and Zhao, W., “eSLAM: An energy-efficient accelerator for real-time orb-slam on fpga platform”.

Motion Planning & Control


[MICRO’16] Murray, S., Floyd-Jones, W., Qi, Y., Konidaris, G. and Sorin, D.J., “The microarchitecture of a real-time robot motion planning accelerator”.

[https://sharclab.ece.gatech.edu/]
Question?

Email: zishenwan@gatech.edu
Homomorphic Work Summary
The general architecture to encompass different workload in FHE

• What’s Homophoric Encryption
• Introduce to Two dominant works: HEAX & PipeZK
• Workload Summary
  o NTT - Introduction
  o MUL - Introduction
• Operator Architecture
  o NTT in HEAX/PipeZK
  o MUL in HEAX/PipeZK
What is full homomorphic encryption

Homomorphic Encryption: Enable processing data in encrypted format without decryption
Homomorphic Operation: Implementation of computation on the ciphertext data.
High-level Challenges

• Slow: early FHE schemes were about $10^9$ times slower than performing computations on unencrypted data -> Acceleration Demanded
• No data-dependent operation allowed: Data are processed in encrypted format.
• Variant encryption schemes support: e.g. BGV, B/FV, GSW, and CKKS.
  o Underlying datatypes are same.
  o Vary in capabilities and performance.
• Complex Operation on long vector:
  o FHE encodes information using very large vectors, several thousand elements long.
  o modular arithmetic processing.
  o Esp two operations including Number-Theoretic Transform (NTT) and automorphisms
• Static Computation: All operations and their dependences are known ahead of time
• Data movement cost: Encrypting data increases its size (typically by at least 50×)
Flow of Homomorphic Encryption - BGV

• Encryption Processing & variables:
  o Plaintext Vector: Uncrypted text
  o Plaintext Polynomial \( m \in R_t \): Plaintext is usually encoded and stored in the coefficients of polynomial
    - \( a_0 + a_1 x + \cdots + a_{N-1} x^{N-1} \in R_t \)
  o Secret key \( \psi \in R_Q \): a polynomial under Modulo Q.
  o Error (Noise) \( \epsilon \in R_Q \): For ensuring the security.
  o Ciphertext \( ct \): Every plaintext polynomial \( m \in R_t \) is encrypted into 2 polynomial (\( \epsilon \)):
    - Every coefficient in encrypted polynomial is under modulo Q
    \[ ct = (a, b = as + te + m). \]
    - samples uniformly random \( a \in R_Q \).

• Decryption Processing:
  o Step 1:
  o Step \( \epsilon' = te + m = b - as \mod Q \)
  o Constraint \( m = \epsilon' \mod t \) as \( \epsilon' \) does not wrap around modulo Q -> decryption is correct -> error (noise) has ub
Supported Homomorphic Operations

- **Homomorphic Addition** = adding their corresponding polynomials in plaintext
  
  \[ ct_{\text{add}} = ct_0 + ct_1 = (a_0 + a_1, b_0 + b_1). \]

- **Homomorphic Multiplication** = multiply corresponding polynomials in plaintext
  
  \[ ct_X = (l_2, l_1, l_0) = (a_0a_1, a_0b_1 + a_1b_0, b_0b_1) \]

  \[ (u_1, u_0) = \text{KeySwitch}(l_2) \]

  \[ ct_{\text{mul}} = (l_1 + u_1, l_0 + u_0) \]

  - Multiplication will change the secret key -> requires KeySwitch to change back to original key
  - Key switch the +++ expensive operation.

- **Homomorphic Permutation** \(\neq\) same permutation on the underlying plaintext vector

  \[ ct_{\sigma} = (\sigma_k(a), \sigma_k(b)) \]

  \[ \sigma_k(a) : a_i \rightarrow (-1)^s a_{ik \mod N} \text{ for } i = 0, \ldots, N - 1, \]

  \[ (u_1, u_0) = \text{KeySwitch}(\sigma_k(a)) \]

  \[ ct_{\text{perm}} = (u_1, \sigma_k(b) + u_0) \]

  - Implemented through *automorphisms*
  - Usage e.g. a single automorphism + careful indexing -> rotate the vector of the N encrypted value
Noise Growth in Homomorphic Operation and Mitigation

• **noise is determined by multiplicative depth:**
  - I.E. the longest chain of homomorphic multiplications in the computation
  - reaons:
    - addition & permutations -> little growth in noise
    - multiplication -> significant growth in noise

• **Mitigation**
  - larger ciphertext modulus Q.
    - e.g. multiplicative depth of 16 requires Q to be about 512 bits.
    - the tolerable *multiplicative depth* grows linearly with logQ
    - increase computation cost since larger Q.
  - **Bootstrapping**, enables FHE computations of unbounded depth
    - Capability: it removes noise from a ciphertext without access to the secret key.
    - Usage: FHE programs with a large multiplicative depth can be divided into regions of limited depth, separated by bootstrapping operations
    - expensive = (typically tens to hundreds) homomorphic operations
  - **Modulus Switching**, rescale the Q into Q’ and noise gets rescaled correspondingly.
    - Usage: introduce modulus switching before homomorphic multiplication to reduce noise -> multiplication (less noise increase & less computation cost since smaller Q) -> noise increase a little -> rescale back.
HAEX: Accelerate keyswitch of the encryption computation of the processing.

Residue Number System Operation - Decomposition

- **Residue Number System (RNS) -> decompost Long-bitwidth data into short-bitwidth data**
  - modulo q:

- **Transform from Normal Number System to RNS**
  - Mapping: \([a]_p\) denotes the reduction of a modulo an integer p to \([0, p - 1] \cap \mathbb{Z}\):
  - Inverse Mapping: modulo decomposition -> different residue-based polynomial
    - E.g. \(q=105\)
      - Decomposition \((q_0, q_1, q_2) = (3, 5, 7)\)
        - \(q_0 = 3\) \((103 \mod 3)* x^2 + (87 \mod 3)* x + (79 \mod 3) = x^2 + 1\)
        - \(q_1 = 5\) \((103 \mod 5)* x^2 + (87 \mod 5)* x + (79 \mod 5) = 3* x^2 + 2* x + 4\)
        - \(q_2 = 7\) \((103 \mod 7)* x^2 + (103 \mod 7)* x + (79 \mod 7) = 5* x^2 + 3* x + 2\)
    - Benefits:
      - reduce the length of coefficient representation -> fit on-chip BRAM.
      - parallelism in multiple \(q_i\) -> multi-core MUL T module
NTT (Number Theoretic Transform) - Introduction

• A typical butterfly operator[1]

\[ y_t = \left( z_t + z_{t+\frac{N}{2}} \right) \]

\[ z_t = \left( z_t - z_{t+\frac{N}{2}} \right) \omega_N^t \]

Figure 3.2 The Gentleman-Sande butterfly.

• A typical 32-size NTT (right)
  • Irregular mem access
    • Input data not continuous at memory
    • Irregular changes every stage.
  • Vectorized Formula: \[ \hat{a} \equiv \text{NTT}(a) \]
  • Coefficient-wise Formula: \[ \hat{a}[i] = \sum_{j=0}^{N-1} a[j] \omega_N^{ij} \]

HEAX -- Keyswitch DSA Overview

- Polynomial size $2^{15}$.
- Target FHE scheme: CKKS
  - benefits over (BGV, BFV, TFHE) in approximate computation[1,2].
- Challenges:
  - store intermediate results on off-chip memory -> worse than navie-software design[3]
  - variant-number polynomial order ($n = 2^k$, $k = 12, 13, 14, 15$)
- Optimization
  - Multiplication Module:
    - Word size decomposition.
      - size reduced from 64 bit to mutli-modulus less than 52 bits (enable 54 bit multiplication using less DSPs)
    - Off-chip Memory access = # of components.
    - Residue-level Parallelism
  - NTT
    - NTT-level Parallelism (later)

HEAX - optimization Throughput Matching

- 1 INTT -> At most L times NTT & Dyadic Mod
- Sufficient Number of BRAM to hold all intermediate results -- avoid data

Algorithm 5 Key Switching | KeySwitch(ct, ksk)

Input: \( ct = (\tilde{C}_0, \tilde{C}_1) \in (\prod_{i=0}^{L} R_p)^2 \), and ksk = 
\[
\left( \left( \tilde{D}_{0,i} \right)_{0 \leq i \leq L+1} \right), \left( \tilde{D}_{1,i} \right)_{0 \leq i \leq L+1} \} : (p \prod_{i=0}^{L} R_p)^{L+2} \times 2
\]

Output: \( ct' = (\tilde{C}'_0, \tilde{C}'_1) \in (\prod_{i=0}^{L} R_p)^2 \)

1. for \( i = 0; i \leq L; i = i + 1 \) do
   2. \( \tilde{a} \leftarrow \text{INTT}_p(\tilde{C}_i) \) \( \Rightarrow \) INTT Module
   3. for \( j = 0; j \leq L; j = j + 1 \) do
      4. if \( i \neq j \) then
         5. \( \tilde{b} \leftarrow \text{Mod}(\tilde{a}, p) \)
         6. \( \tilde{b} \leftarrow \text{NTT}_p(\tilde{b}) \) \( \Rightarrow \) NTT Module
      7. else
         8. \( \tilde{b} \leftarrow \tilde{a} \)
         9. end if
      10. \( \tilde{c}'_i \leftarrow \tilde{c}'_{i,j} \mod p_j \)
      11. \( \tilde{c}'_i \leftarrow \tilde{c}'_{i,j} \mod p_j \) \( \Rightarrow \) Dyadic Mod.
   12. end for
   13. \( \tilde{b} \leftarrow \text{Mod}(\tilde{a}, p) \)
   14. \( \tilde{b} \leftarrow \text{NTT}_p(\tilde{b}) \) \( \Rightarrow \) NTT Module
   15. \( \tilde{c}'_{i+1} \leftarrow \tilde{c}'_{i+1} \mod p_j \)
   16. \( \tilde{c}'_{i+1} \leftarrow \tilde{c}'_{i+1} \mod p_j \) \( \Rightarrow \) Dyd. M.
   17. end for
18. \( \tilde{c}' \leftarrow \text{INTT/NTT/MS}(\tilde{C}'_0, \tilde{C}'_1) \)
19. \( \tilde{c}' \leftarrow \text{CKKS.Add}(ct, ct') \)

Figure 6. High-level pipeline of KeySwitch module.
HEAX -- data off-chip v.s. on-chip trade off

<table>
<thead>
<tr>
<th>FPGA Device</th>
<th>HE Param. Set</th>
<th>KeySwitch Architecture Parameter Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arria10</td>
<td>( n = 2^{12} ) (Set-A)</td>
<td>( 1 \times \text{INTT}^{(8)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 3 \times \text{Dyad}^{(4)} \rightarrow 2 \times \text{INTT}^{(4)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 2 \times \text{MS}^{(2)} )</td>
</tr>
<tr>
<td></td>
<td>( n = 2^{12} ) (Set-A)</td>
<td>( 1 \times \text{INTT}^{(16)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 3 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(8)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 2 \times \text{MS}^{(4)} )</td>
</tr>
<tr>
<td>Stratix10</td>
<td>( n = 2^{13} ) (Set-B)</td>
<td>( 1 \times \text{INTT}^{(16)} \rightarrow 4 \times \text{NTT}^{(16)} \rightarrow 5 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(4)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 2 \times \text{MS}^{(4)} )</td>
</tr>
<tr>
<td></td>
<td>( n = 2^{14} ) (Set-C)</td>
<td>( 1 \times \text{INTT}^{(8)} \rightarrow 4 \times \text{NTT}^{(16)} \rightarrow 5 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(1)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 2 \times \text{MS}^{(4)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HE Param. Set</th>
<th>( n )</th>
<th>([\log_{\text{2}} n] + 1)</th>
<th>( k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-A</td>
<td>( 2^{12} )</td>
<td>109</td>
<td>2</td>
</tr>
<tr>
<td>Set-B</td>
<td>( 2^{13} )</td>
<td>218</td>
<td>4</td>
</tr>
<tr>
<td>Set-C</td>
<td>( 2^{14} )</td>
<td>438</td>
<td>8</td>
</tr>
</tbody>
</table>

- **Memory Element Comparison**
  - size of ksk (keyswitch key)
    - \( O(nk^2) > O(n^3) \) when \( n \geq 2^{14} \), put ksk into off-chip memory.
    - read once per each KeySwitch
  - twiddle factors
    - \( O(nk) \)
    - read \( k \) times during one KeySwitch
- **Store ksk off-chip across multiple DRAM banks to fully utilize the off-chip bandwidth**
- **Stream load from off-chip DRAM in the burst mode to feed ksk in the pipeline manner.**
PipeZK: Accelerator for Zero-Knowledge Proof

Zero-knowledge Proof: verify **proof without** knowing the **knowledge** of the data

PipeZK: Accelerate the whole system Zero-knowledge Proof with CPU+FPGA
Zero-Knowledge Proof

- **Application: Zero-knowledge Proof**
  - verify **proof without** knowing the **knowledge** of the data
  - “given a **function F** and an input x, I know a secret witness w that makes F(x, w) = 0”
  - Accelerate Proof generation.

- **Scalar vectors**
  - Each vector includes $n \lambda$-bit numbers
  - $n$ could be extremely large (millions) for real-world applications

- **Point vectors**
  - Each vector includes $n$ points on a pre-determined elliptic curve (EC)
    -> enable calculation over the finite field.
    -> compute-heavy

- **Latency of Proof generation**
  - Zcash: 30 seconds ($n=2$ million)
  - Filecoin: an hour ($n=128$ million)

---

Prover’s Computation

- Finite Field (like field), opposite of infinite field
  - E.g. Residue Number System under moduli q.
  - only [0, q-1] (total q number)

- Prover’s Comp -> generate proof $\Pi$
  - Poly (30%):
    - degree-n (up to millions) Polynomial $H_n$
    - coefficient bit-length
    - Bottleneck: Large-size NTT:
      - Complicated memory access pattern
      - large finite field multiplication & addition
  - MSM (~70%): n wide integer number multiplication -> computation-intensive
    - Duplicate multiplication module does not help much -> require results accumulation.
    - call for new algorithm -> Pippenger algorithm

- Why accelerator not CPU/GPU?
  - irregular global memory access -- GPU inefficiency.
  - insufficient computation throughput -- CPU parallelism underutilization.

PipeZK

- **Application: Zero-knowledge Proof**
  - Verify the proof without knowing the knowledge of the data.
  - “given a function F and an input x, I know a secret witness w that makes F(x, w) = 0”

- **Challenges:**
  - NTT
    - irregular strided access patterns
    - large bitwidth (up to 768 bits) of each element
  - multi-scalar multiplications (Pippenger algorithm[1])
    - load imbalance
    - source underutilization

- **Solutions:**
  - a novel high-level dataflow that recursively decomposes the large NTT kernels into smaller ones
  - pipelined dataflow based multi-scalar point multiplications on elliptic curves

---

PipeZK - NTT Formulaic representation

- \( \hat{a} \overset{\text{def}}{=} \text{NTT}(a) \)
  - \( a \) and \( \hat{a} \) are N-size array
  - \( \hat{a}[i] = \sum_{j=0}^{N-1} a[j] \omega_N^{ij} \)
  - \( \omega_N \) is Nth root of unity in the same field, called twiddle factors
    - constant values for a specific N
    - stored in off-chip memory (tens of MB for N=several Millions)

- **Challenge 1: memory access**
  - a million-size NTT with 256-bit data width will need over 64 MB for input data and twiddle factors [Level 1]
  - for feeding an 1024-size NTT module (100MHz) with 1024 elements access -> 2.98 TB/s bandwidth [Level 2]

- **Solution (next slide)**
  - Recursively Build NTT (Two-level)
    - Level 1: Terabit (application) -> recursive gigabit/megabit (block)
    - Level 2: Each recursive gigabit/megabit (block) -> Recursive pipeline butterfly (operator)
Input: [0], [1], ..., [N-1]; ([j] is a coefficient of a polynomial; N is up to million)

- Input Reshape
- Step 1: Run I-size NTT recurrently for J times
- Step 2: N element-wise mul
- Step 3: Run J-size NTT recurrently for I times
- Output Reshape
PipeZK: 2-level decomposition for large-size NTT

1024-size NTT

- Depth of FIFO decrease with number of stage increasing
  - Stride decrease for first ele. of each stage
- First stage Operation:
  - 0~511 cycles -> new data coming -> stored in FIFO
  - 512~1023 cycles -> front of FIFO and new input get calculated
    - one output directly sent to the next stage.
    - another output gets stored back to the FIFO and sent to the next stage at a later point, termed **second-half-res**
  - 1024~1535 cycles -> new data coming -> stored in FIFO
    - Simultaneously output **second-half-res** to next stage
A walk-through Example for 4-size NTT
I/J-size NTT in PipeZK: Pipeline-PE pipeline

- Core: store data internally to satisfy the stride
  - Store Latency = parallel-to-serial latency
  - Performance: N-size NTT takes “2N - 1 + (Core Latency)*logN” cycle
- Pros:
  - Low bandwidth requirement -> a single coefficient per cycle
  - Simple control (compared with MUX in HEAX).
- Cons:
  - Storage Overhead -> (N-1)* coefficient FIFO storage.
PipeZK: 2-level decomposition for large-size NTT (2)

1024-size NTT

- Latency: $13 \log N + \frac{N}{t} + \frac{NT}{t}$
  - Warm-up
  - Data buffering
  - Process $N$ in $T$ modules

- Warm-up
  - A stage takes 13 cycle to in arithmetic operations
  - Total number of stage: $\log N$

- Data buffering
  - $N=1024 \approx 512+256+128+64+32+16+8+4+2+1$
  - Latency of storing data in all FIFO.

- Process $N$ in $T$ modules
  - One pipeline pass -> output one element
    - 1 hw + 1 NTT workload -> $N$ cycles
    - $t$ hw + $T$ NTT workload -> $N*T/t$ cycles

- Could start from any stage -> tune for different size < 1024
  - Power of 2

Limitation? My guess
Workload Summary

Operation in HEAX [1]

Operation in PipeZK [2]

Core Operation are **NTT/INTT & MUL**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096 dim 54 bit data</td>
<td>NTT/INTT</td>
<td>Large Dim (up to million)</td>
</tr>
<tr>
<td>Scalar Modulus multiplication</td>
<td>MUL</td>
<td>Long-bitwidth data (up to 768 bit)</td>
</tr>
</tbody>
</table>
4096-size NTT in HEAX: Multiple PE design[1]

Input: \( x[0], x[1], \ldots, x[n-1] \); \( x[j] \) is a coefficient of a polynomial

- Core: Rely complex control of MUXing to fetch irregular data -> feed NTT/INTT core
- Performance: \( n \)-size NTT takes \( \frac{n \log n}{2n_{\text{NTT}}} \) cycle
- Pros: Intuitive
- Cons: not scalable for large size.
  - \( 4^{*}n_{\text{NTT}} \text{og} 2(n_{\text{NTT}}) \text{MUX (MUX2)} \)

## NTT Comparison

<table>
<thead>
<tr>
<th>HEAX</th>
<th>Item</th>
<th>PipeZK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM - MUX - Multiple PE</td>
<td>Core</td>
<td>MEM - Pipeline PE</td>
</tr>
<tr>
<td>4096</td>
<td>Workload</td>
<td>million -&gt; 1024 -&gt; pipeline</td>
</tr>
<tr>
<td>( \frac{n \log n}{2 n_{\text{NTT}}} )</td>
<td>Latency</td>
<td>2N - 1 + (Core Latency)*logN</td>
</tr>
<tr>
<td>Complex control with lower storage</td>
<td>Features</td>
<td>Simple control but with higher storage</td>
</tr>
<tr>
<td>Superlinear Scale Overhead</td>
<td>Scalability</td>
<td>Easy to scale</td>
</tr>
</tbody>
</table>

- **HEAX Item**
  - MEM - MUX - Multiple PE
  - 4096
- **PipeZK**
  - Core
  - Workload
  - Latency
  - Features
  - Scalability
MUL Comparison

- Challenge: Long bitwidth data in MUL -> decomposition
- HEAX -- E.g. \( q = 105 \) \( 103 \times x^2 + 87 \times x + 79 \)
  - Decomposition \((q_0, q_1, q_2) = (3, 5, 7)\)
    - \( q_0 = 3 \) \( (103 \text{ mod } 3) \times x^2 + (87 \text{ mod } 3) \times x + (79 \text{ mod } 3) = x^2 + 1 \)
    - \( q_1 = 5 \) \( (103 \text{ mod } 5) \times x^2 + (87 \text{ mod } 5) \times x + (79 \text{ mod } 5) = 3 \times x^2 + 2 \times x + 4 \)
    - \( q_2 = 5 \) \( (103 \text{ mod } 7) \times x^2 + (103 \text{ mod } 7) \times x + (79 \text{ mod } 7) = 5 \times x^2 + 3 \times x + 2 \)
- PipeZK -- Group (sparsity)
  - Large Bit number division into multiple small group
  - Leverage of sparsity.
## NTT Comparison

<table>
<thead>
<tr>
<th>HEAX</th>
<th>Item</th>
<th>PipeZK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM - MUX - Multiple PE</td>
<td>Core</td>
<td>MEM - Pipeline PE</td>
</tr>
<tr>
<td>4096</td>
<td>Workload</td>
<td>million -&gt; 1024 -&gt; pipeline</td>
</tr>
<tr>
<td>(\frac{n \log n}{2 \cdot n_{C_{NTT}}})</td>
<td>Latency</td>
<td>2N - 1 + (Core Latency)*logN</td>
</tr>
<tr>
<td>Complex control with lower storage</td>
<td>Features</td>
<td>Simple control but with higher storage</td>
</tr>
<tr>
<td>Superlinear Scale Overhead</td>
<td>Scalability</td>
<td>Easy to scale</td>
</tr>
<tr>
<td>(4^*nc_{NTT} + \log(\frac{n_{C_{NTT}}}{nc_{NTT}}))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MUL Comparison

- Challenge: Long bitwidth data in MUL -> decomposition

- HEAX -- E.g. q=105 down 103*x^2 + 87*x + 79
  - Decomposition (q_0, q_1, q_2) = (3,5,7)
    - q_0 = 3 down (103 mod 3)* x^2 + (87 mod 3) * x + (79 mod 3) = x^2 + 1
    - q_1 = 5 down (103 mod 5) * x^2 + (87 mod 5) * x + (79 mod 5) = 3* x^2 + 2* x + 4
    - q_2 = 5 down (103 mod 7) * x^2 + (103 mod 7) * x + (79 mod 7) = 5* x^2 + 3* x + 2

- PipeZK -- Group (sparsity)
  - Large Bit number division into multiple small group
  - Leverage of sparsity.
## NTT Comparison

<table>
<thead>
<tr>
<th>HEAX</th>
<th>Item</th>
<th>PipeZK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM - MUX - Multiple PE</td>
<td>Core</td>
<td>MEM - Pipeline PE</td>
</tr>
<tr>
<td>4096</td>
<td>Workload</td>
<td>million -&gt; 1024 -&gt; pipeline</td>
</tr>
<tr>
<td>$\frac{n \log n}{2n^{\text{NTT}}}$</td>
<td>Latency</td>
<td>$2N - 1 + (\text{Core Latency}) \times \log N$</td>
</tr>
<tr>
<td>Complex control with lower storage</td>
<td>Features</td>
<td>Simple control but with higher storage</td>
</tr>
<tr>
<td>Superlinear Scale Overhead</td>
<td>Scalability</td>
<td>Easy to scale</td>
</tr>
</tbody>
</table>
MUL Comparison

• Challenge: Long bitwidth data in MUL -> decomposition

• HEAX -- E.g. \( q=105 \) \( 103 \times x^2 + 87 \times x + 79 \)
  • Decomposition \( (q_0, q_1, q_2) = (3,5,7) \)
    • \( q_0 = 3 \) \( (103 \mod 3) \times x^2 + (87 \mod 3) \times x + (79 \mod 3) = x^2 + 1 \)
    • \( q_1 = 5 \) \( (103 \mod 5) \times x^2 + (87 \mod 5) \times x + (79 \mod 5) = 3 \times x^2 + 2 \times x + 4 \)
    • \( q_2 = 5 \) \( (103 \mod 7) \times x^2 + (103 \mod 7) \times x + (79 \mod 7) = 5 \times x^2 + 3 \times x + 2 \)

• PipeZK -- Group (sparsity)
  • Large Bit number division into multiple small group
  • Leverage of sparsity.