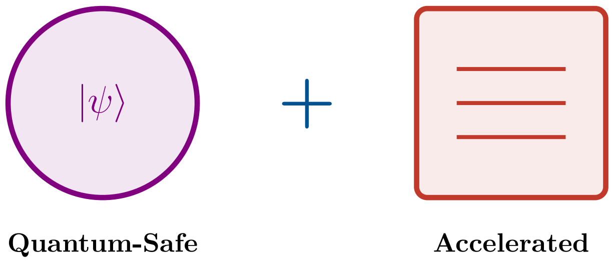
Post-Quantum Cryptography & Hardware Acceleration

GPU and FPGA-Based Compression

A Comprehensive Guide for Future-Proof Computing



Abdessamad JAOUAD

M2 Big Data & IoT

ENSAM Casablanca

January 3, 2026

Contents

1 Introduction **3**
1.1 The Quantum Threat . . . 3
1.1.1 Timeline Estimates . . . 3
1.2 The Performance Challenge . . . 3
1.3 Hardware Acceleration Solutions . . . 3
1.4 Document Overview . . . 4
1.5 Why This Matters . . . 4

2 Post-Quantum Cryptography Fundamentals **5**
2.1 What is Post-Quantum Cryptography? . . . 5
2.2 Why Current Cryptography Fails . . . 5
2.2.1 Shor's Algorithm . . . 5
2.2.2 Complexity Comparison . . . 5
2.3 Grover's Algorithm . . . 6
2.4 Mathematical Foundations of PQC . . . 6
2.5 Security Levels . . . 6

3 PQC Algorithm Families **7**
3.1 Overview of Algorithm Families . . . 7
3.2 Lattice-Based Cryptography . . . 7
3.2.1 What is a Lattice? . . . 7
3.2.2 Hard Lattice Problems . . . 7
3.2.3 LWE Problem Explained . . . 7
3.3 Hash-Based Signatures . . . 8
3.3.1 One-Time Signatures (OTS) . . . 8
3.3.2 Merkle Trees . . . 8
3.4 Code-Based Cryptography . . . 8
3.4.1 McEliece Cryptosystem (1978) . . . 8
3.5 Algorithm Family Comparison . . . 9

4 NIST PQC Standards **10**
4.1 Standardization Timeline . . . 10
4.2 The Four Standardized Algorithms . . . 10
4.3 ML-KEM (Kyber) . . . 10
4.3.1 How ML-KEM Works . . . 11
4.4 ML-DSA (Dilithium) . . . 11
4.5 SLH-DSA (SPHINCS+) . . . 11
4.6 Comparison Summary . . . 11

5 GPU Architecture and CUDA **13**
5.1 GPU vs CPU Architecture . . . 13
5.2 CUDA Programming Model . . . 13
5.2.1 Key Concepts . . . 13
5.2.2 Thread Hierarchy . . . 14
5.3 CUDA Memory Model . . . 14
5.4 CUDA Kernel Example . . . 14
5.4.1 Execution Flow . . . 14
5.5 Performance Considerations . . . 15
5.6 CUDA Libraries for Compression . . . 15

6 OpenCL and Cross-Platform Computing **16**
6.1 Introduction to OpenCL . . . 16
6.1.1 Platform Model . . . 16
6.1.2 OpenCL Architecture Layers . . . 16
6.2 Memory Model . . . 16
6.3 OpenCL Kernel Programming . . . 17
6.3.1 Kernel Syntax . . . 17
6.3.2 Key OpenCL Functions . . . 17
6.4 CUDA vs OpenCL Comparison . . . 18

7 FPGA Architecture and Programming **19**
7.1 Introduction to FPGAs . . . 19
7.2 FPGA Architecture . . . 19
7.2.1 Core Components . . . 19
7.2.2 Look-Up Table (LUT) Structure . . . 19
7.3 FPGA vs GPU vs CPU . . . 20
7.4 FPGA Programming Methods . . . 20
7.4.1 Traditional HDL Flow . . . 20
7.4.2 High-Level Synthesis (HLS) . . . 20
7.5 Major FPGA Vendors . . . 21

8 Hardware-Accelerated Compression **22**
8.1 Why Hardware Acceleration? . . . 22
8.2 GPU Compression Techniques . . . 22
8.2.1 Parallel LZ77 . . . 22
8.2.2 Parallel Huffman Coding . . . 23
8.3 FPGA Compression Implementations . . . 23
8.3.1 Streaming LZ77 Architecture . . . 23
8.3.2 Commercial FPGA Compression IP . . . 23
8.4 Hybrid CPU-GPU-FPGA Systems . . . 23
8.5 PQC Hardware Acceleration . . . 24
8.5.1 PQC Operations on Hardware . . . 24
8.5.2 Number Theoretic Transform (NTT) . . . 24

9 Conclusion and Future Directions **25**
9.1 Key Takeaways . . . 25
9.2 Future Trends . . . 25
9.2.1 Emerging Technologies . . . 25
9.3 Recommendations . . . 26
9.3.1 Implementation Roadmap . . . 26
9.4 Summary Table . . . 26

Chapter 1

Introduction

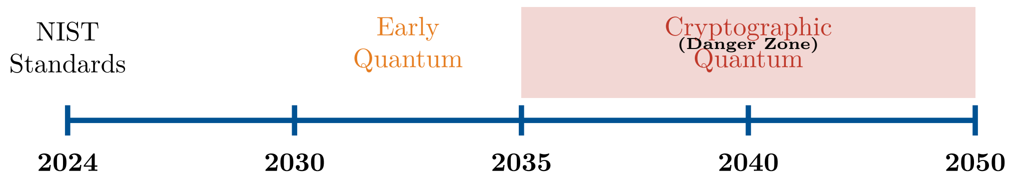
1.1 The Quantum Threat

The advent of **quantum computing** poses an existential threat to modern cryptography. Algorithms like **Shor's algorithm** can break RSA, ECC, and other public-key systems in polynomial time.

Harvest Now Decrypt Later Attack

Adversaries are already collecting encrypted data today, planning to decrypt it once quantum computers become available. This makes post-quantum migration **urgent**.

1.1.1 Timeline Estimates



1.2 The Performance Challenge

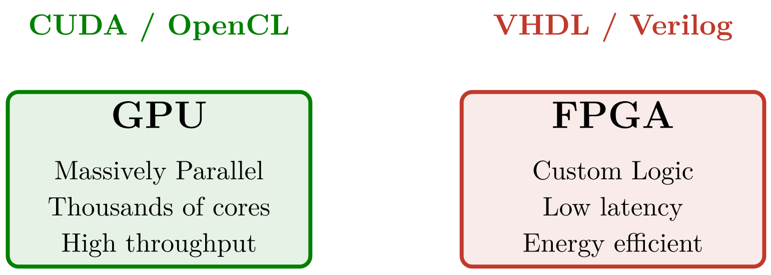
Post-quantum algorithms have significantly **larger key sizes** and **slower operations** than classical cryptography. Hardware acceleration becomes essential.

Table 1.1: Key Size Comparison: Classical vs. Post-Quantum

Algorithm	**Type**	**Public Key**	**Private Key**
RSA-2048	Classical	256 B	1.2 KB
ECDSA P-256	Classical	64 B	32 B
ML-KEM-768	PQC (Lattice)	1,184 B	2,400 B
ML-DSA-65	PQC (Lattice)	1,952 B	4,032 B
SLH-DSA-128s	PQC (Hash)	32 B	64 B

1.3 Hardware Acceleration Solutions

Two main approaches exist for accelerating cryptographic and compression workloads:



1.4 Document Overview

This guide covers two critical future trends in computing:

1. **Post-Quantum Cryptography (PQC)** – Cryptographic algorithms resistant to quantum attacks
2. **Hardware Acceleration** – GPU and FPGA-based acceleration for compression and cryptography

Chapter Roadmap

- **Chapter 2:** Post-Quantum Cryptography Fundamentals
- **Chapter 3:** PQC Algorithm Families
- **Chapter 4:** NIST PQC Standards
- **Chapter 5:** GPU Architecture & CUDA
- **Chapter 6:** OpenCL & Cross-Platform Computing
- **Chapter 7:** FPGA Architecture & Programming
- **Chapter 8:** Hardware-Accelerated Compression
- **Chapter 9:** Conclusion & Future Directions

1.5 Why This Matters

Critical Takeaway

The intersection of **post-quantum security** and **hardware acceleration** is not optional—it's essential for any system that needs to remain secure and performant in the coming decades.

Chapter 2

Post-Quantum Cryptography Fundamentals

2.1 What is Post-Quantum Cryptography?

Post-Quantum Cryptography (PQC) refers to cryptographic algorithms designed to resist attacks from both classical and quantum computers.

Key Distinction

PQC algorithms run on **classical computers**—they don't require quantum hardware. They're simply designed to resist quantum attacks.

2.2 Why Current Cryptography Fails

2.2.1 Shor's Algorithm

Shor's algorithm (1994) can efficiently solve two hard problems:

Quantum Vulnerable

Integer Factorization
$N = p \times q$
Breaks: RSA

Discrete Logarithm
$g^x \equiv h \pmod{p}$
Breaks: DH, ECDSA

2.2.2 Complexity Comparison

Table 2.1: Time Complexity: Classical vs Quantum

Problem	**Classical**	**Quantum (Shor)**
Factoring n-bit integer	$O(e^{n^{1/3}})$	$O(n^3)$
Discrete Log	$O(e^{n^{1/3}})$	$O(n^3)$

Exponential to Polynomial

Shor's algorithm reduces **exponential** problems to **polynomial** time—a catastrophic break for RSA and ECC.

2.3 Grover's Algorithm

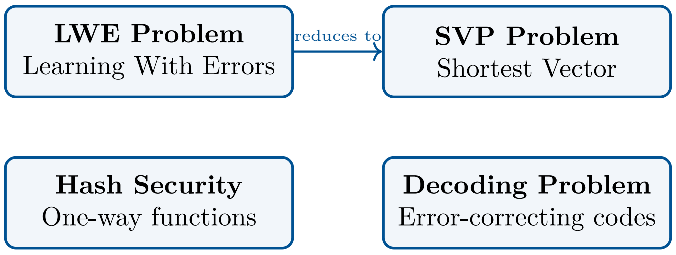
Grover's algorithm provides a **quadratic speedup** for searching unsorted databases:

- Classical search: $O(N)$
- Quantum search: $O(\sqrt{N})$

Impact on symmetric crypto: AES-128 becomes effectively AES-64. Solution: **double key sizes** (AES-256).

2.4 Mathematical Foundations of PQC

PQC relies on problems believed to be hard for *both* classical and quantum computers:



2.5 Security Levels

NIST defines 5 security levels based on equivalent classical security:

Table 2.2: NIST Security Levels

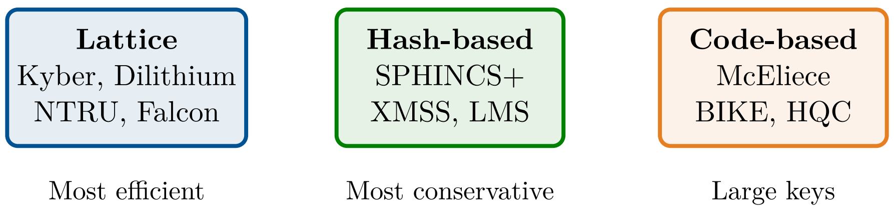
Level	Equivalent Security	Bits
1	AES-128 key search	128
2	SHA-256 collision	128
3	AES-192 key search	192
4	SHA-384 collision	192
5	AES-256 key search	256

Chapter 3

PQC Algorithm Families

Post-quantum algorithms are grouped into families based on their underlying mathematical problems.

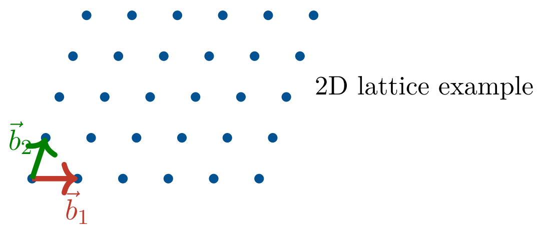
3.1 Overview of Algorithm Families



3.2 Lattice-Based Cryptography

3.2.1 What is a Lattice?

A **lattice** is a regular grid of points in n-dimensional space, generated by a set of basis vectors.



3.2.2 Hard Lattice Problems

1. **SVP** (Shortest Vector Problem): Find the shortest non-zero vector
2. **CVP** (Closest Vector Problem): Find the closest lattice point to a target
3. **LWE** (Learning With Errors): Distinguish noisy linear equations from random

3.2.3 LWE Problem Explained

Given samples of the form:

$$(\mathbf{a}_i, b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i) \mod q$$

Where $\mathbf{s}$ is secret and e_i is small noise. **Goal:** Recover $\mathbf{s}$.

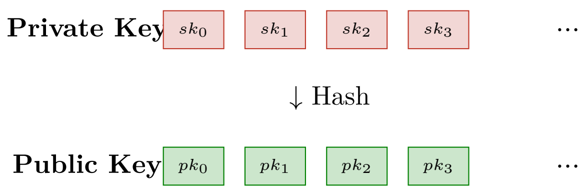
Why LWE is Hard

The noise e_i prevents simple linear algebra solutions. Without noise, solving is trivial. With noise, it becomes as hard as worst-case lattice problems.

3.3 Hash-Based Signatures

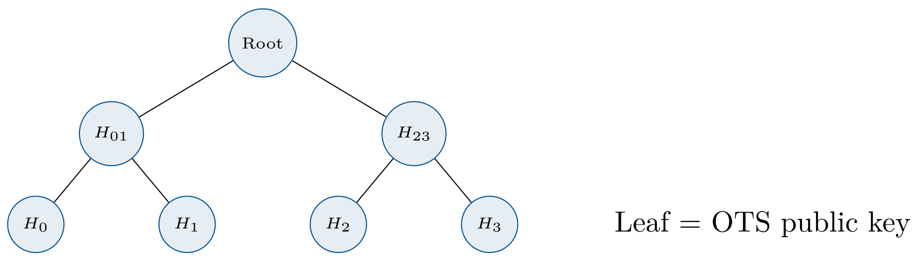
Hash-based signatures rely *only* on the security of hash functions—the most conservative assumption.

3.3.1 One-Time Signatures (OTS)



3.3.2 Merkle Trees

To sign multiple messages, OTS keys are organized in a **Merkle tree**:



3.4 Code-Based Cryptography

Based on the difficulty of decoding random linear codes.

3.4.1 McEliece Cryptosystem (1978)

1. Generate a decodable code with generator matrix G
2. Disguise it: $G' = SGP$ (scramble and permute)
3. Public key: G' Private key: S, G, P
4. Encrypt: $c = mG' + e$ (add errors)
5. Decrypt: Use private key to decode

Large Keys

McEliece public keys are $\sim$1 MB at 128-bit security. This limits its use to specific applications.

3.5 Algorithm Family Comparison

Table 3.1: PQC Family Comparison

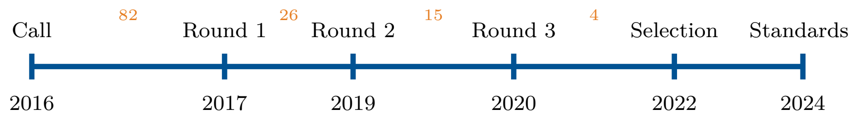
Family	**Key Size**	**Speed**	**Maturity**	**Best For**
Lattice	Small	Fast	Medium	General use
Hash-based	Small/Med	Slow	High	Long-term security
Code-based	Large	Fast	High	Key encapsulation

Chapter 4

NIST PQC Standards

In August 2024, NIST released the first post-quantum cryptography standards after an 8-year evaluation process.

4.1 Standardization Timeline



4.2 The Four Standardized Algorithms

FIPS 203: ML-KEM
(CRYSTALS-Kyber)
Key Encapsulation

FIPS 204: ML-DSA
(CRYSTALS-Dilithium)
Digital Signature

FIPS 205: SLH-DSA
(SPHINCS+)
Hash-based Signature

FIPS 206: FN-DSA
(Falcon)
Compact Signature

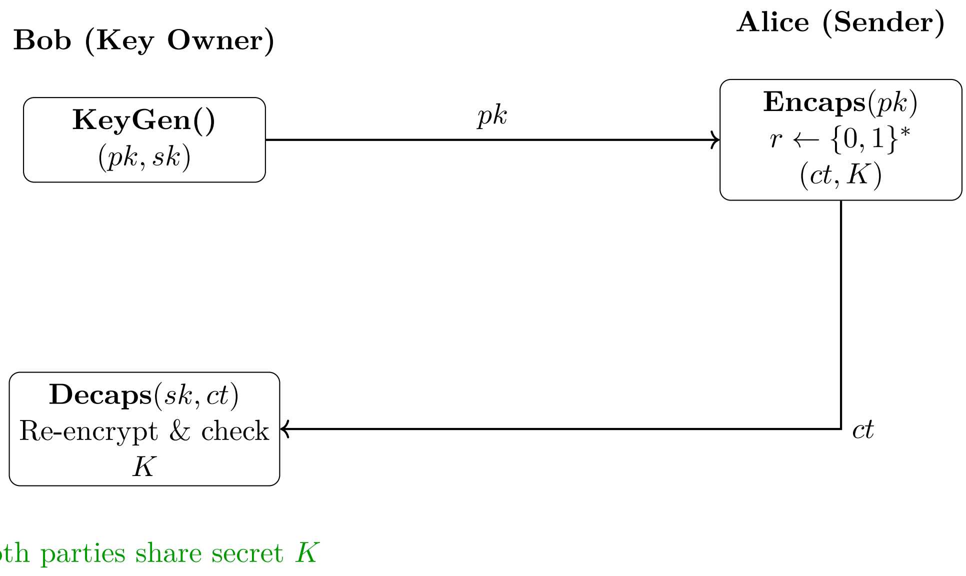
4.3 ML-KEM (Kyber)

Purpose: Key Encapsulation Mechanism (establishes shared secrets)

Table 4.1: ML-KEM Parameter Sets

Variant	Security	Public Key	Ciphertext	Shared Secret
ML-KEM-512	Level 1	800 B	768 B	32 B
ML-KEM-768	Level 3	1,184 B	1,088 B	32 B
ML-KEM-1024	Level 5	1,568 B	1,568 B	32 B

4.3.1 How ML-KEM Works



4.4 ML-DSA (Dilithium)

Purpose: Digital signatures (authentication)

Table 4.2: ML-DSA Parameter Sets

Variant	Security	Public Key	Private Key	Signature
ML-DSA-44	Level 2	1,312 B	2,560 B	2,420 B
ML-DSA-65	Level 3	1,952 B	4,032 B	3,309 B
ML-DSA-87	Level 5	2,592 B	4,896 B	4,627 B

4.5 SLH-DSA (SPHINCS+)

Purpose: Conservative hash-based signatures

Why SPHINCS+?

SPHINCS+ relies **only** on hash function security. If lattice problems are broken, SPHINCS+ remains secure. It's the "backup plan."

Trade-off: Small keys (32-64 bytes) but large signatures (8-50 KB).

4.6 Comparison Summary

Table 4.3: NIST Standards Comparison (Level 3)

Algorithm	Type	Pub Key	Priv Key	Sig/CT
ML-KEM-768	KEM	1,184 B	2,400 B	1,088 B
ML-DSA-65	Signature	1,952 B	4,032 B	3,309 B
SLH-DSA-128f	Signature	32 B	64 B	17,088 B

Recommendation

For most applications: Use ML-KEM for key exchange and ML-DSA for signatures. Use SLH-DSA when conservative security assumptions are critical.

Chapter 5

GPU Architecture and CUDA

5.1 GPU vs CPU Architecture

GPUs and CPUs are designed for fundamentally different tasks:

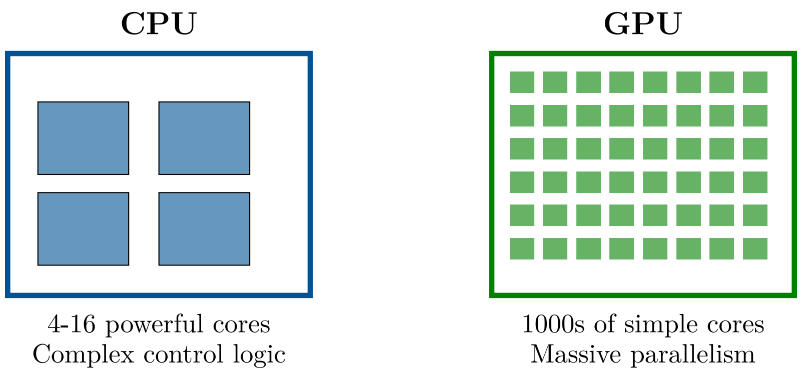


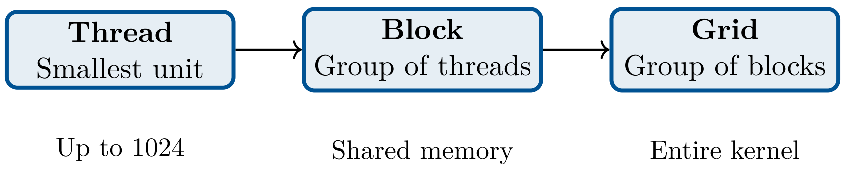
Table 5.1: CPU vs GPU Characteristics

Feature	**CPU**	**GPU**
Core count	4-64	1,000-16,000
Clock speed	3-5 GHz	1-2 GHz
Cache per core	Large (MB)	Small (KB)
Best for	Sequential tasks	Parallel tasks
Control flow	Complex branching	Simple SIMD

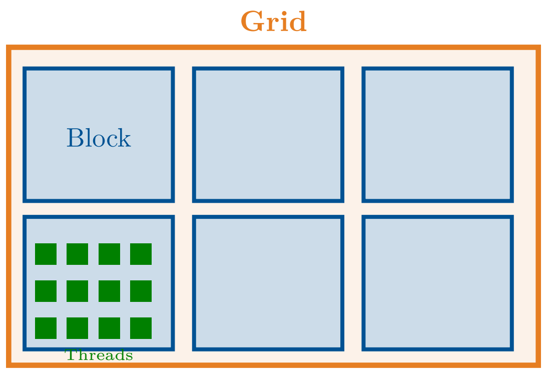
5.2 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform.

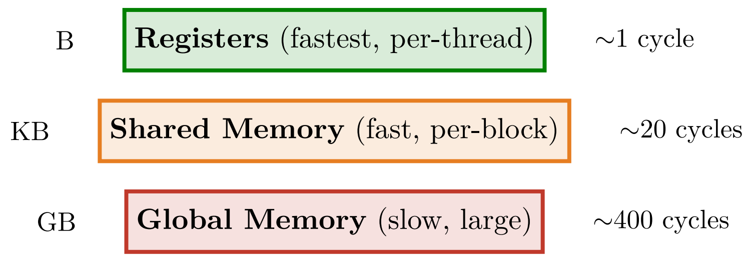
5.2.1 Key Concepts



5.2.2 Thread Hierarchy



5.3 CUDA Memory Model


B
Registers (fastest, per-thread)
~1 cycle
KB
Shared Memory (fast, per-block)
~20 cycles
GB
Global Memory (slow, large)
~400 cycles

5.4 CUDA Kernel Example

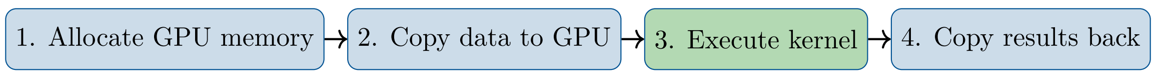
A **kernel** is a function that runs on the GPU:

Vector Addition Kernel

```
__global__ void vectorAdd(float *a, float *b,
                          float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

// Launch: vectorAdd<<<numBlocks, blockSize>>>(...)
```

5.4.1 Execution Flow



5.5 Performance Considerations

Optimization Tips

1. **Maximize parallelism:** Launch thousands of threads
2. **Minimize memory transfers:** CPU↔GPU is slow
3. **Use shared memory:** Cache frequently accessed data
4. **Coalesce memory access:** Adjacent threads access adjacent memory
5. **Avoid branch divergence:** Threads in a warp should follow same path

5.6 CUDA Libraries for Compression

NVIDIA provides optimized libraries:

Table 5.2: NVIDIA Compression Libraries

Library	**Algorithms**	**Throughput**
nvCOMP	LZ4, Snappy, DEFLATE, zstd	Up to 500 GB/s
cuBLAS	Linear algebra (for transforms)	Optimized
cuFFT	FFT (for frequency domain)	Optimized

nvCOMP Performance

nvCOMP can achieve **150**× speedup over CPU implementations for batch compression of small blocks.

Chapter 6

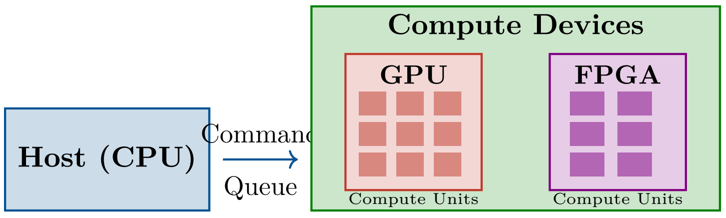
OpenCL and Cross-Platform Computing

6.1 Introduction to OpenCL

OpenCL (Open Computing Language) is an open standard for parallel programming across heterogeneous platforms—CPUs, GPUs, FPGAs, and DSPs from different vendors.

OpenCL provides **vendor-neutral** parallel computing, enabling the same code to run on AMD, Intel, NVIDIA, and ARM devices.

6.1.1 Platform Model



6.1.2 OpenCL Architecture Layers

1. **Platform Model**: Host + compute devices with compute units and processing elements
2. **Execution Model**: Kernels executed by work-items organized in work-groups
3. **Memory Model**: Global, local, constant, and private memory regions
4. **Programming Model**: Data-parallel and task-parallel execution

6.2 Memory Model

OpenCL defines four memory regions with different scope and performance characteristics:

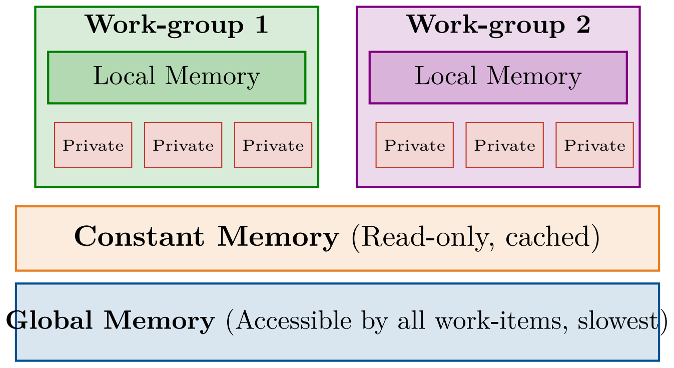


Table 6.1: OpenCL Memory Regions

Memory	Scope	Speed	Size
Global	All work-items	Slow	Large (GBs)
Constant	All (read-only)	Fast (cached)	Limited (64KB)
Local	Work-group	Fast	Small (48KB)
Private	Single work-item	Fastest	Very small

6.3 OpenCL Kernel Programming

6.3.1 Kernel Syntax

OpenCL kernels are written in OpenCL C, a subset of C99 with extensions:

Listing 6.1: OpenCL Vector Addition Kernel

```
__kernel void vector_add(
    __global const float* A,
    __global const float* B,
    __global float* C,
    const int N)
{
    int gid = get_global_id(0);
    if (gid < N) {
        C[gid] = A[gid] + B[gid];
    }
}
```

6.3.2 Key OpenCL Functions

Function	Purpose
`get_global_id(dim)`	Global work-item ID
`get_local_id(dim)`	Local ID within work-group
`get_group_id(dim)`	Work-group ID
`get_global_size(dim)`	Total number of work-items
`get_local_size(dim)`	Work-group size
`barrier()`	Synchronization within work-group

6.4 CUDA vs OpenCL Comparison

Table 6.2: CUDA vs OpenCL Feature Comparison

Aspect	**CUDA**	**OpenCL**
Vendor	NVIDIA only	Cross-platform
Performance	Highly optimized	Vendor-dependent
Terminology	Threads, Blocks, Grids	Work-items, Work-groups, NDRange
Memory	Shared, Global	Local, Global
Ecosystem	Mature libraries	Broader hardware support
Learning Curve	Easier	More complex

While OpenCL offers portability, **CUDA typically achieves 10-30% better performance** on NVIDIA hardware due to vendor-specific optimizations.

Chapter 7

FPGA Architecture and Programming

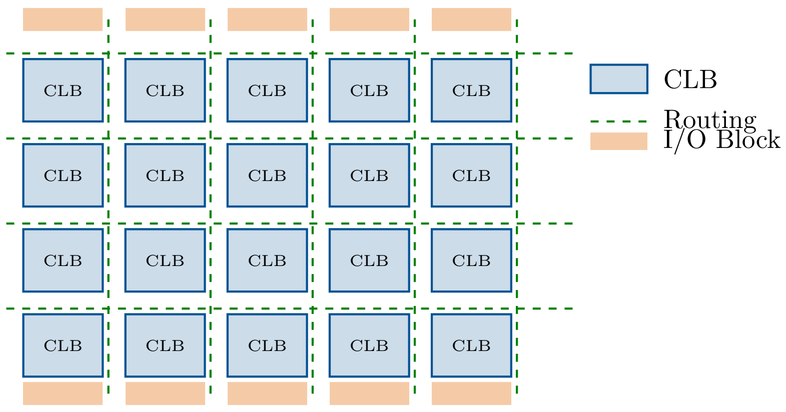
7.1 Introduction to FPGAs

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured after manufacturing to implement custom digital logic.

Unlike GPUs that execute instructions, FPGAs implement algorithms **directly in hardware**, achieving extreme parallelism and energy efficiency.

7.2 FPGA Architecture

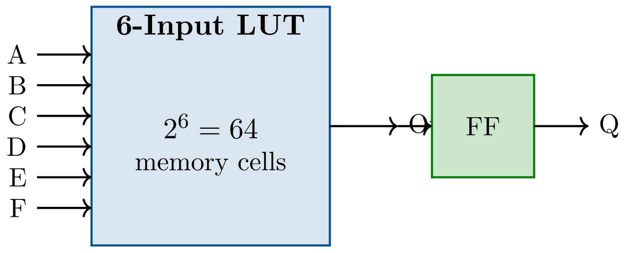
7.2.1 Core Components



- **Configurable Logic Blocks (CLBs)**: Contain LUTs, flip-flops, and multiplexers
- **Programmable Interconnect**: Routing matrix connecting CLBs
- **I/O Blocks**: Interface with external devices
- **Block RAM (BRAM)**: Dedicated memory blocks
- **DSP Slices**: Hardened multiply-accumulate units

7.2.2 Look-Up Table (LUT) Structure

A LUT implements any boolean function by storing the truth table in memory:



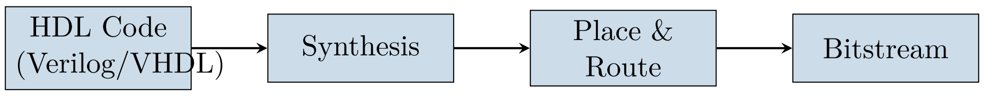
7.3 FPGA vs GPU vs CPU

Table 7.1: Hardware Platform Comparison

Metric	**CPU**	**GPU**	**FPGA**
Parallelism	Low (8-64 cores)	High (thousands)	Extreme (custom)
Flexibility	Very high	High	Medium
Power Efficiency	Low	Medium	High
Latency	Medium	High	Very low
Development Time	Fast	Medium	Slow
Throughput	Low	Very high	High

7.4 FPGA Programming Methods

7.4.1 Traditional HDL Flow



7.4.2 High-Level Synthesis (HLS)

Modern FPGAs support **High-Level Synthesis**, allowing C/C++/OpenCL code to be compiled to hardware:

Listing 7.1: HLS Example - Matrix Multiply

```c
void matmul(float A[N][K], float B[K][M], float C[N][M]) {
    #pragma HLS INTERFACE m_axi port=A
    #pragma HLS INTERFACE m_axi port=B
    #pragma HLS INTERFACE m_axi port=C

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            #pragma HLS PIPELINE II=1
            float sum = 0;
            for (int k = 0; k < K; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

HLS Pragmas

HLS pragmas guide the compiler to optimize for:

- `PIPELINE`: Enable pipelined execution
- `UNROLL`: Replicate loop iterations in hardware
- `ARRAY_PARTITION`: Distribute arrays across BRAMs

7.5 Major FPGA Vendors

Vendor	**Product Lines**	**Tools**
AMD/Xilinx	Versal, UltraScale+, Artix	Vivado, Vitis
Intel/Altera	Agilex, Stratix, Cyclone	Quartus Prime
Lattice	CrossLink-NX, ECP5	Radiant, Diamond
Microchip	PolarFire, SmartFusion	Libero SoC

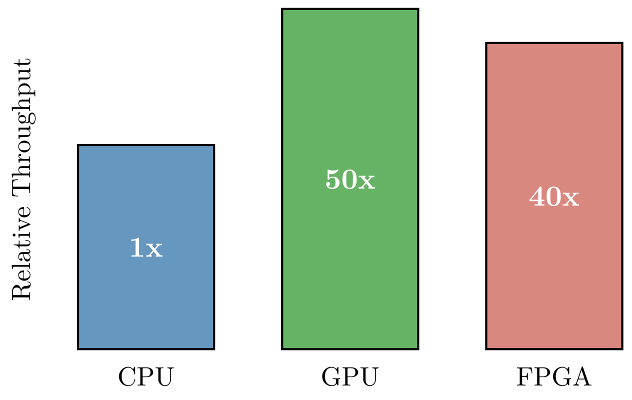
Chapter 8

Hardware-Accelerated Compression

8.1 Why Hardware Acceleration?

Data compression is computationally intensive. Hardware acceleration addresses:

1. **Throughput**: Process data faster than CPU alone
2. **Latency**: Reduce compression/decompression time
3. **Energy**: Lower power consumption per operation
4. **CPU Offloading**: Free CPU for other tasks

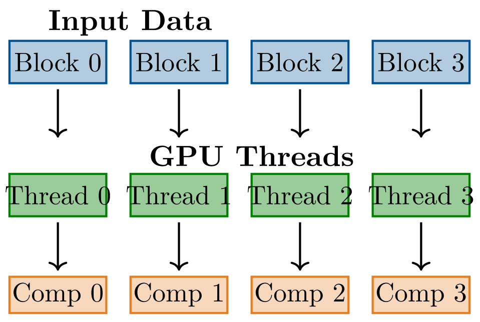


8.2 GPU Compression Techniques

8.2.1 Parallel LZ77

Traditional LZ77 is sequential (each match depends on previous output). GPU-optimized approaches:

- **Block-parallel**: Divide input into independent blocks
- **Hash-based matching**: Parallel hash table lookups
- **Batch processing**: Compress multiple streams simultaneously



8.2.2 Parallel Huffman Coding

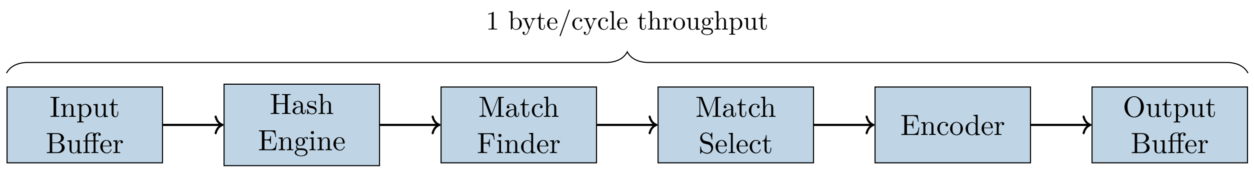
1. **Histogram**: Parallel frequency counting (atomic operations)
2. **Tree Building**: Sequential on CPU (small overhead)
3. **Encoding**: Parallel symbol-to-code mapping
4. **Bit Packing**: Parallel with careful synchronization

8.3 FPGA Compression Implementations

FPGAs excel at compression due to:

FPGAs implement compression as **streaming pipelines**—data flows through dedicated hardware stages with single-cycle latency per stage.

8.3.1 Streaming LZ77 Architecture



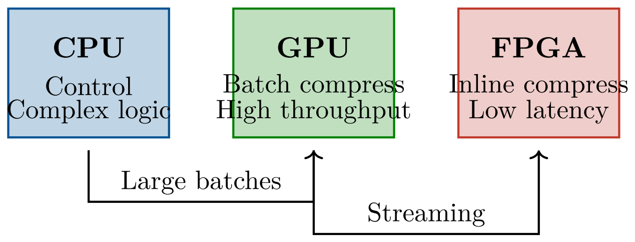
8.3.2 Commercial FPGA Compression IP

Table 8.1: FPGA Compression IP Cores

Vendor	Algorithm	Throughput	Ratio
AMD/Xilinx	GZIP/ZSTD	8 GB/s	2.5-3x
Intel	GZIP	10 GB/s	2.5x
Atomic Rules	ZSTD	100 Gbps	3x
Eideticom	LZ4/ZSTD	64 GB/s	2-3x

8.4 Hybrid CPU-GPU-FPGA Systems

Modern systems combine accelerators for optimal performance:



8.5 PQC Hardware Acceleration

Post-quantum cryptography is computationally expensive. Hardware acceleration is essential:

8.5.1 PQC Operations on Hardware

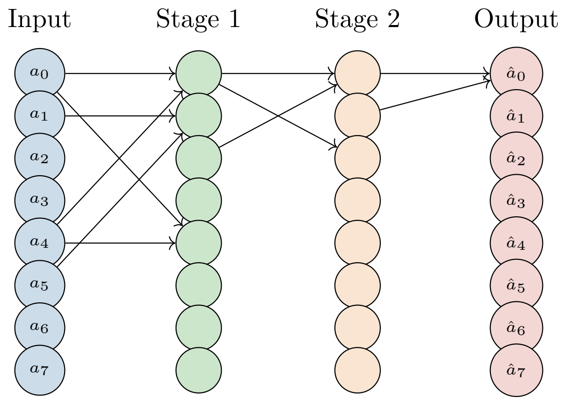
Table 8.2: PQC Hardware Acceleration Strategies

Operation	**GPU Approach**	**FPGA Approach**
NTT (Lattice)	Parallel butterfly ops	Pipelined butterflies
Polynomial mult	Batch NTTs	Streaming NTT
Matrix operations	GEMM kernels	Systolic arrays
Hashing (SHAKE)	Parallel Keccak	Pipelined Keccak
Sampling	Parallel rejection	Hardware sampler

Hardware-accelerated PQC can achieve **100-1000x** speedup over software implementations, making it practical for high-throughput applications.

8.5.2 Number Theoretic Transform (NTT)

NTT is the core operation in lattice-based cryptography. Similar to FFT but over finite fields:



NTT Parallelism

Each NTT stage has $n/2$ independent butterfly operations. GPUs process these in parallel; FPGAs pipeline them for streaming throughput.

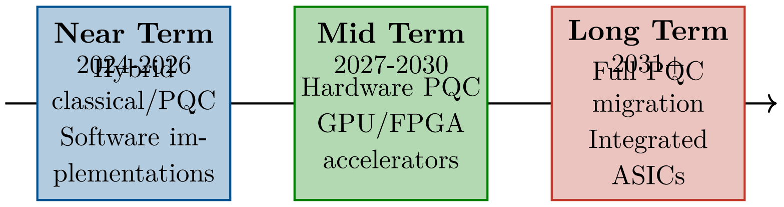
Chapter 9

Conclusion and Future Directions

9.1 Key Takeaways

1. **Quantum Threat**: Shor's algorithm breaks RSA/ECC; Grover weakens symmetric crypto. Migration to PQC is urgent.
2. **NIST Standards**: ML-KEM, ML-DSA, SLH-DSA, and FN-DSA provide standardized post-quantum security.
3. **Performance Challenge**: PQC algorithms have larger keys and higher computational costs than classical cryptography.
4. **GPU Acceleration**: Massive parallelism enables batch processing; ideal for servers and data centers.
5. **FPGA Acceleration**: Custom pipelines offer low latency and energy efficiency; ideal for embedded and networking.
6. **Compression + PQC**: Hardware-accelerated compression reduces the bandwidth impact of larger PQC keys.

9.2 Future Trends



9.2.1 Emerging Technologies

- **PQC ASICs**: Custom chips for specific PQC algorithms
- **In-memory Computing**: Process data where it resides
- **Photonic Accelerators**: Optical computing for NTT operations
- **Quantum-Safe HSMs**: Hardware security modules with PQC support

9.3 Recommendations

Organizations should begin planning PQC migration **now**. The "Harvest Now, Decrypt Later" threat means sensitive data encrypted today may be compromised in the future.

9.3.1 Implementation Roadmap

1. **Inventory**: Catalog all cryptographic assets and dependencies
2. **Prioritize**: Focus on long-lived secrets and critical systems
3. **Test**: Evaluate PQC algorithms for performance and compatibility
4. **Hybrid**: Deploy hybrid classical/PQC solutions initially
5. **Monitor**: Track NIST guidance and quantum computing advances
6. **Accelerate**: Implement hardware acceleration where needed

9.4 Summary Table

Table 9.1: Technology Selection Guide

Use Case	**Recommended**	**Rationale**
Server/Cloud	GPU (CUDA)	High throughput, batch processing
Networking	FPGA	Low latency, inline processing
Embedded/IoT	FPGA	Power efficiency, small footprint
General Purpose	CPU + Software	Flexibility, ease of deployment
Hybrid Systems	GPU + FPGA	Combine throughput and latency

The transition to post-quantum cryptography is not optional—it is a necessary evolution. Hardware acceleration will be essential to maintain performance in a post-quantum world.

"The quantum revolution in computing necessitates a quantum revolution in cryptography."