HLS Final Report **Team 03**

Dilithium

1. Introduction

Post-quantum cryptography, also known as quantum-resistant cryptography, refers to cryptographic methods that are secure against an attack by a quantum computer. With the potential development of large-scale quantum computers in the future, it is important to consider the potential vulnerabilities of current cryptographic methods and to research and develop new methods that will remain secure against quantum attacks.

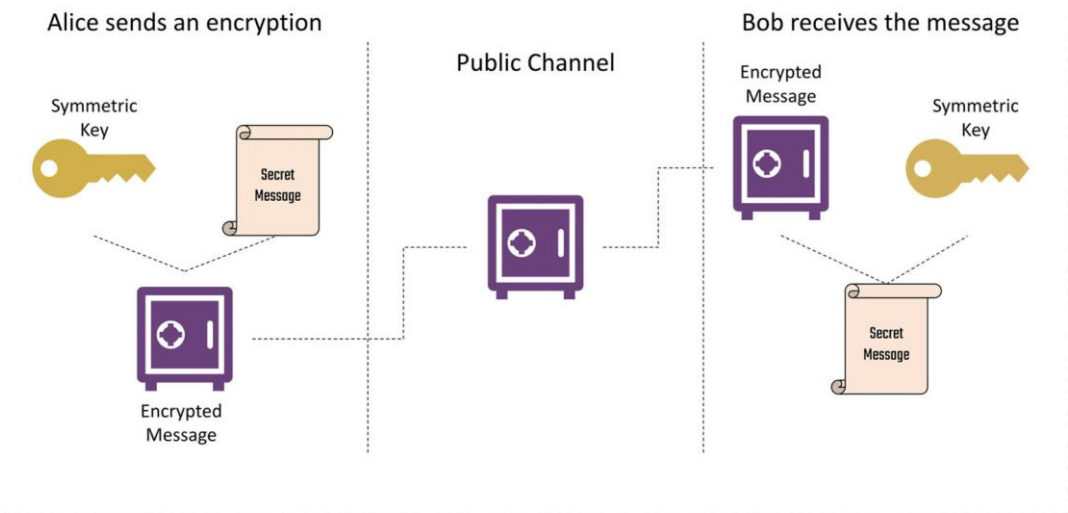


Figure: Symmetric cryptography in action

(An Introduction to Post-Quantum Cryptography - Crypto Quantique)

One approach to post-quantum cryptography is to use cryptographic methods based on mathematical problems that are believed to be hard to solve with a quantum computer. Examples of these problems include the learning with errors problem and the shortest vector problem. These problems have exponential complexity even with a quantum computer.

Hardware acceleration can be used to improve the performance of post-quantum cryptography, particularly for methods that may be slower to execute than their classical counterparts. Hardware acceleration can be achieved through specialized cryptographic hardware, such as cryptographic coprocessors and secure enclaves, or through general-purpose hardware acceleration techniques, such as graphics

processing units (GPUs) and field-programmable gate arrays (FPGAs).

There are several challenges to implementing post-quantum cryptography, including the potential for large key sizes and the need for frequent key updates. However, the potential benefits of secure communication in the face of a quantum computer make the development of post-quantum cryptography an important research area.

In this work, we chose **Dilithum**, a post-quantum public-key encryption, and digital signature algorithm, to demonstrate the efficiency of hardware acceleration using high-level synthesis. Dilithium is based on the learning with errors (LWE) problem. It is strongly secure under chosen message attacks based on the hardness of lattice problems over module lattices. Dilithium is one of the candidate algorithms submitted to the NIST post-quantum cryptography project.

2. Algorithm Workflow

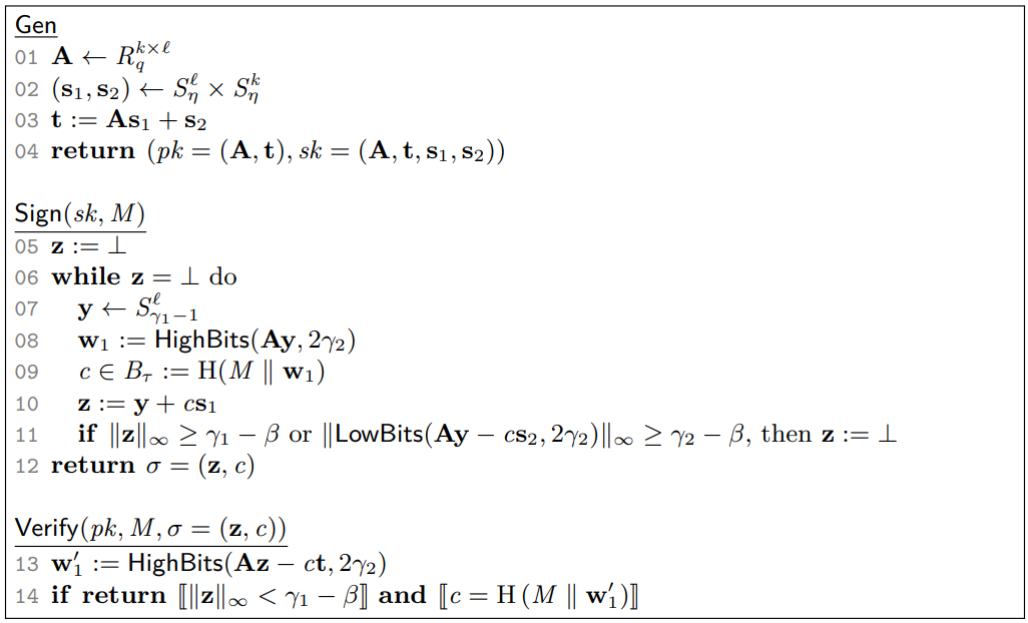
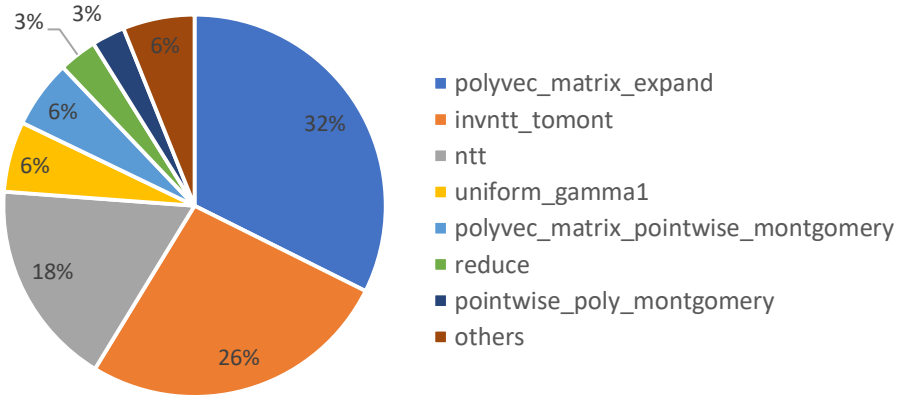


Figure: Template for the signature scheme

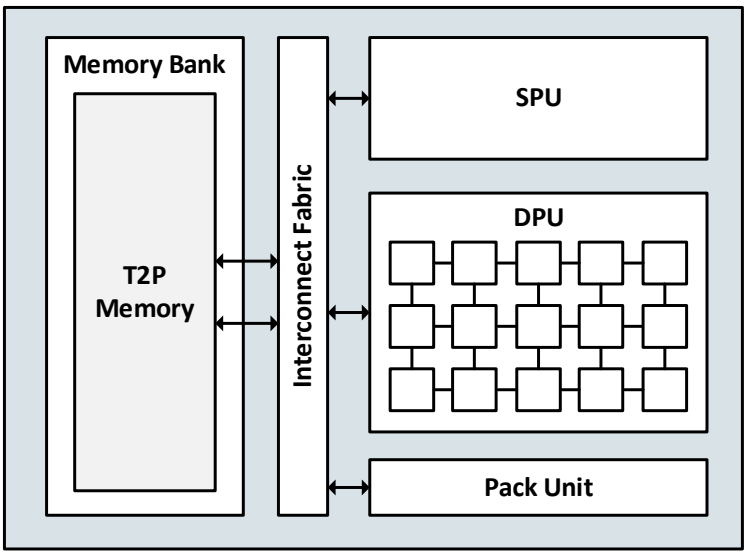
For key generation, the system generates a public/private keypair by sampling a secret key and then computing the corresponding public key. For signing, the signer first hashes the message using a cryptographic hash function. Then, the algorithm uses the secret key to generate a digital signature for the message. To verify the signature, the algorithm uses the public key and the message to check that the signature is valid. If the signature is valid, the receiver can be confident that the message was signed by the owner of the secret key.

To perform CPU profiling for algorithm analysis, we run the algorithm using an Intel Core i7-9700. The runtime result is shown below. It is self-evident that uniform sampling, the number theoretic transform (NTT), and matrix multiplication contribute to the computation bottleneck, for the three operations account for 82% of execution time. Hence, the main focus of this work is to accelerate these processes on FPGA using high-level synthesis.



3. Proposed Architecture for Dilithium

Below is the proposed fully-integrated architecture supporting operations for Dilithium. In the architecture, a memory bank stores all the vectors of polynomials during the computation. The SPU performs uniform sampling to generate a set of coefficients for the polynomial. The DPU is responsible for all element-wise computation, including NTT, inverse NTT, vector addition/multiplication, and matrix multiplication. The pack unit performs bit-packing that encodes vectors as byte strings. The interconnect fabric exchange and allocate data for different operations.



4. Design Implementation Detail

The DPU supports operations for all element-wise data computation. In a vector, an element is composed of a 256-degree polynomial. The 256-point polynomial coefficient is computed in a parallel manner. As a result, the DPU comprises 256 processing elements (PEs), with each processing element supporting different basic operations. The supported PE operation is as follows: addition, subtraction, multiplication, CADDQ (add Q if the element is smaller than 0), reduce (reduce the range of data to 0~Q), value pass, power2round (decompose an element into MSBs and LSBs), NTT, and inverse NTT.

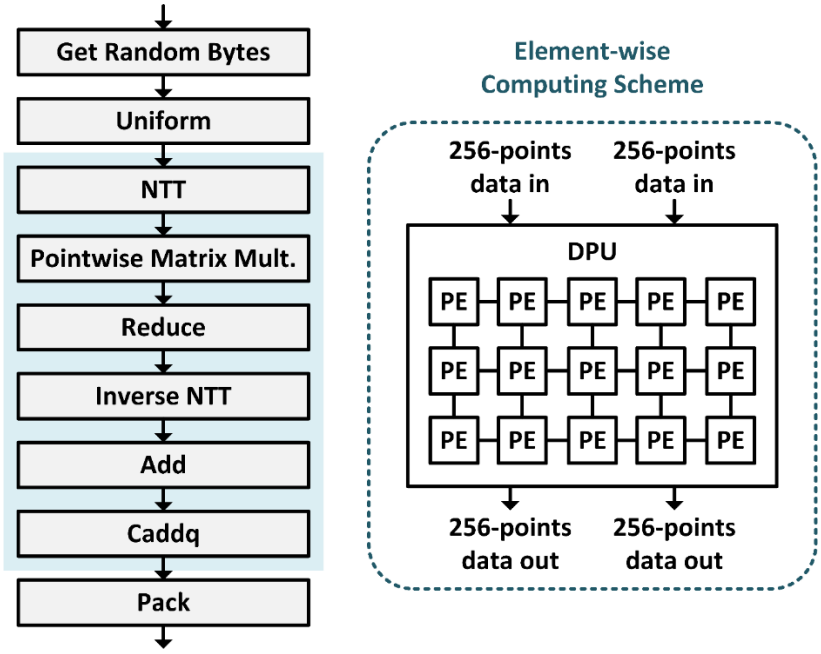


Figure: Element-wise computing scheme and its supported operations

To better utilize the massive computation resource in the DPU, the operation flow is scheduled in a pipelined manner, as shown below. The operation of the DPU can be separated into three stages: read data, execute, and write data. The read-data operation first reads two inputs (p1 & p2) to the DPU from the two-port memory bank. The DPU then performs the operation according to the instruction code. The write-data operation writes the computed result back to the memory bank. For each vector with a length of L, the needed operation can be executed in a pipelined fashion with a pipeline depth of L. In this way, the PE array of 256 units can be fully utilized without having any idle cycle waiting for the data.

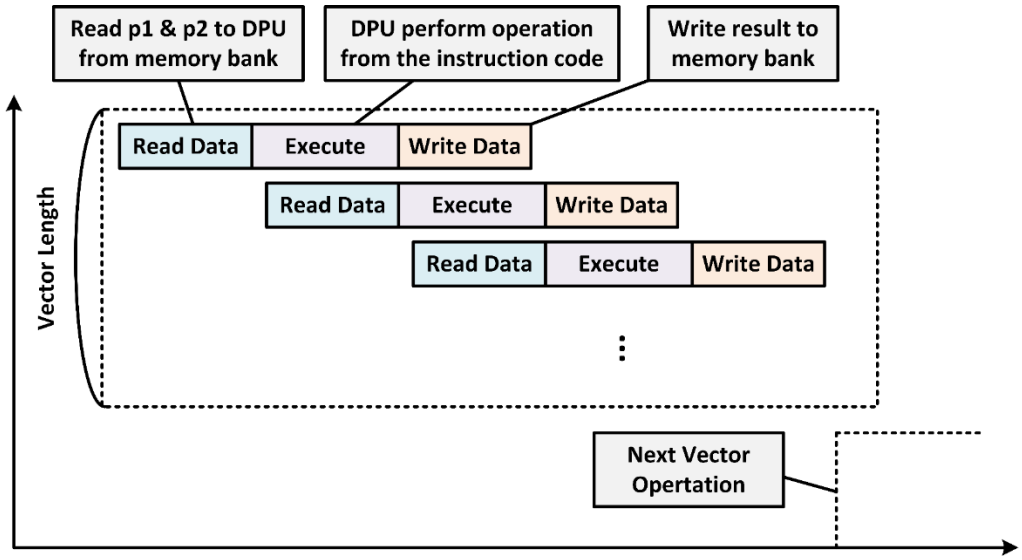


Figure: DPU operation flow

5. Code Implementation

To implement our proposed architecture using Vitis HLS, we must rewrite the original C code. The below shows our rewritten code structure for the DPU class.

```
class DPU
{
private:
    int32_t pMem [POLY_NUM][N];
    int32_t p1[N], p2[N], p3[N], p4[N];
    int32_t p5[N]; //for pack
public:
    int norm_err;
    int sig_err;
    int hn;
    DPU();
    void read_p1(uint8_t addr);
    void read_p2(uint8_t addr);
    void read_p5(uint8_t addr);
    void read_ntt(uint8_t addr);
    void read_intt(uint8_t addr);
    void write_p3(uint8_t addr);
    void write_p4(uint8_t addr);
    void write_p5(uint8_t addr);
    void dpu_func(const uint8_t addr1, const uint8_t addr2, const uint8_t addr3, uint8_t type, uint8_t arg, uint8_t itr);
    void dpu_pack(const uint8_t addr, uint8_t* rb, uint8_t* pb, int ptrs, uint8_t type, const int itr);
    void dpu_unit(uint8_t type);
};
```

The variable `pMem` is a two-port memory bank, and `p1-p4` is the 256-point array for the polynomial vector. For the read/write operation, the memory bank access is defined in the `read_*/write_*` function. For the execution, the functions `dpu_func` and `dpu_pack` define the DPU data control and instruction assignment for different operations. The function `dpu_unit` creates the 256 PE array for different element-wise operations.

The below shows our code structure for the DPU modules, including memory read/write functions and the 256 PE array (`dpu_unit`). For these operations, we added the pragma UNROLL to facilitate parallel processing. This ensures that all 256

coefficients can be accessed and processed at the same time. In the dpu_unit, all micro-operations are integrated into a single function. The DPU reads two inputs and generates two outputs for each cycle. To control the operation to be executed, an instruction code is sent into the function, and the case statement controls the execution.

```
void DPU::dpu_unit(uint8_t type){
    static int64_t tmp;
    for(int i=0;i<N;i++) {
#pragma HLS UNROLL
        switch(type){
            case OP_ADD:
                p3[i] = p1[i] + p2[i];
                break;
            case OP_SUB:
                p3[i] = p1[i] - p2[i];
                break;
            case OP_MUL:
                tmp = (int64_t)p1[i]*p2[i];
                p3[i] = tmp;
                p4[i] = (int32_t)(tmp>>32);
                break;
            case OP_RD32:
                p3[i] = (p1[i] + (1 << 22)) >> 23;
                break;
            case OP_CADDQ:
                p3[i] = p1[i] + ((p1[i] >> 31) & Q);
                break;
            case OP_PASS:
                p3[i] = p1[i];
                break;
            case OP_POW2ROUND://p1[i](a) --> p3[i](a1), p
                p3[i] = (p1[i] + (1 << (D-1)) - 1) >> D;
                p4[i] = p1[i] - (p3[i] << D);
                break;
            case OP_NTT:
```

```
void DPU::write_p3(uint8_t addr){
    for(int i=0;i<N;i++) {
#pragma HLS unroll
        pMem[addr][i] = p3[i];
    }
}
```

```
void DPU::read_p1(uint8_t addr){
    for(int i=0;i<N;i++) {
#pragma HLS unroll
        p1[i] = pMem[addr][i];
    }
}
```

To implement the NTT operations in the DPU, we adopted the butterfly architecture in ISSCC'22. The basic operations in the two functions are the same, with different execution orders. Hence, these element-wise additions and multiplications can be perfectly combined into our designed DPU.

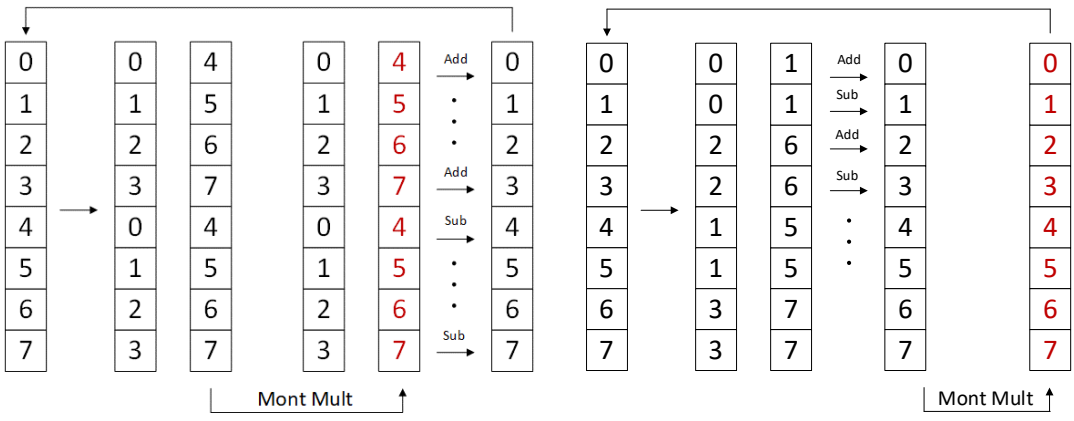
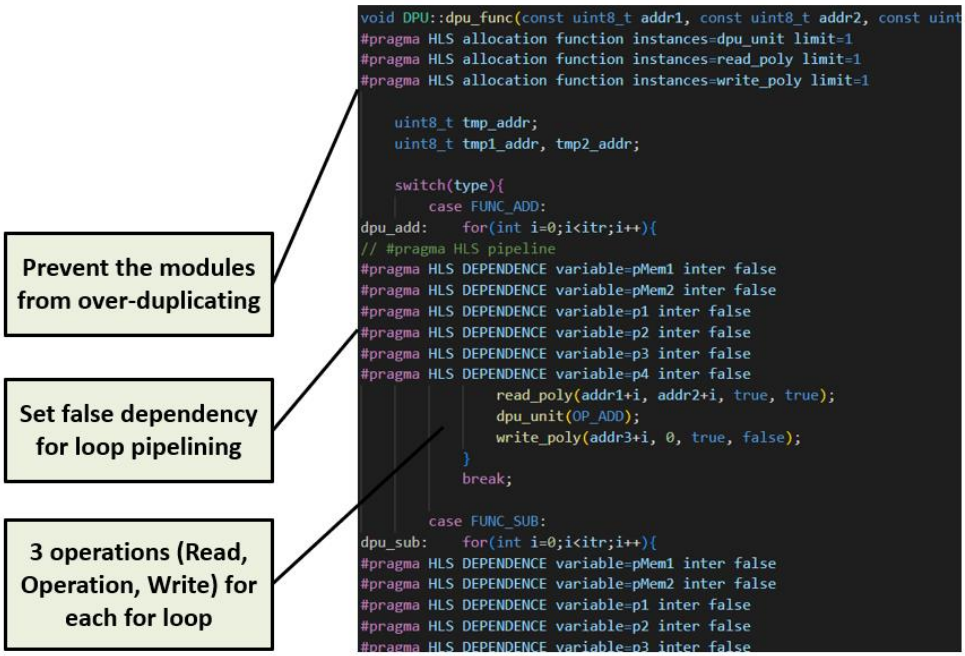
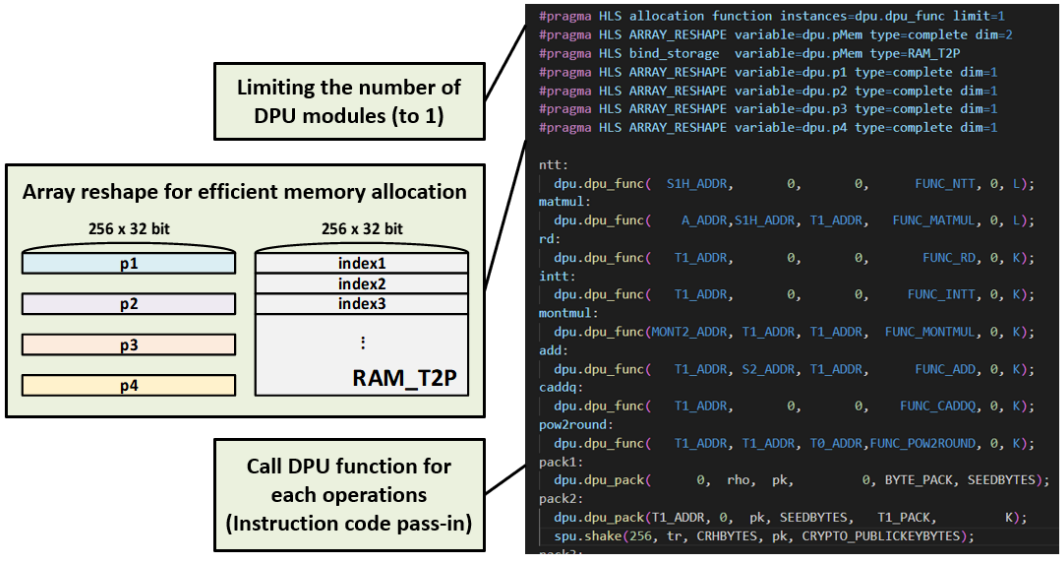


Figure: NTT (left) and inverse NTT (right) operation flow

The below shows the code structure and control logic for DPU operations. An array of poly can be processed independently for each micro-operation, so there is no data dependency between each array element. Therefore, we added pragma DEPENDENCE to clearly specify the false dependency to enable perfect pipeline execution.



Our top-level function is described below. All array and memory are reshaped using the pragma ARRAY_RESHAPE to ensure that all 256 coefficients can be accessed simultaneously. To read two inputs at the same time, we use true-two-port RAM by adding the pragma BIND_STORAGE. In the top-level function, all operations are written using the dpu_func function with different memory addresses and operation instructions.



6. Implementation Results and Observations

In this section, we describe our overall code structure and our observations using Viis HLS. Below is our source code for software validation and hardware implementation.

Kernel code:

--- src/HLS_Final_vitis_src/config.h

--- src/HLS_Final_vitis_src/dpu.cpp

--- src/HLS_Final_vitis_src/dpu.h

--- src/HLS_Final_vitis_src/spu.cpp

--- src/HLS_Final_vitis_src/spu.h

--- src/HLS_Final_vitis_src/wrapper.cpp

--- src/HLS_Final_vitis_src/wrapper.h

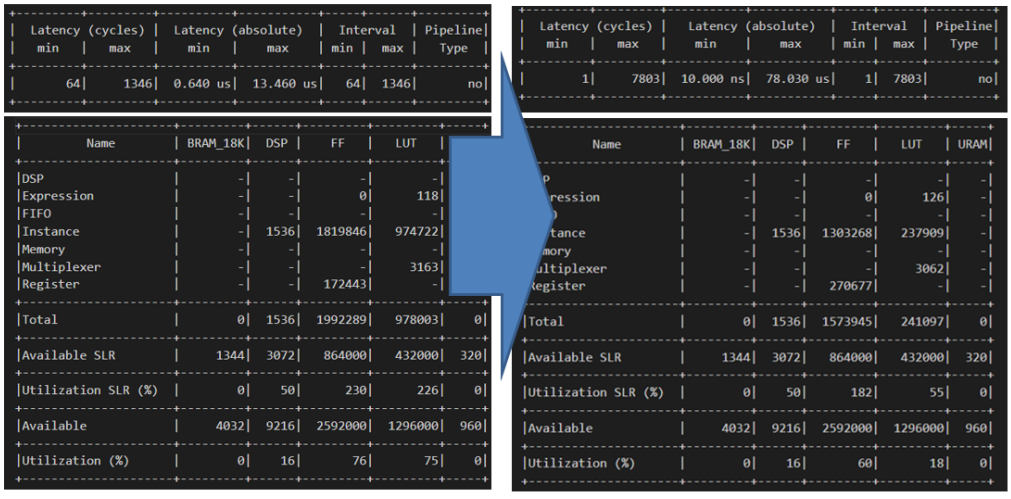
Testbench files:

--- src/HLS_Final_vitis_src/test_dilithium.cpp

--- src/HLS_Final_vitis_src/clean

During our implementation using Vitis HLS, we observed that:

1. The DPU unit has abnormally high latency and resource usage. After we changed our coding style from a “switch-case” to an “if-else” statement, the synthesis result significantly improved. The minimum latency dropped from 64 to 1, and the resource usage reduced drastically, as shown below.



Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
64	1346	0.640 us	13.460 us	64	1346	no

Name	BRAM_18K	DSP	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	118
FIFO	-	-	-	-
Instance	-	1536	1819846	974722
Memory	-	-	-	-
Multiplexer	-	-	-	3163
Register	-	-	172443	-
Total	0	1536	1992289	978003
Available SLR	1344	3072	864000	432000
Utilization SLR (%)	0	50	230	226
Available	4032	9216	2592000	1296000
Utilization (%)	0	16	76	75

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
1	7803	10.000 ns	78.030 us	1	7803	no

Name	BRAM_18K	DSP	FF	LUT	URAM
P	-	-	-	-	-
ression	-	-	0	126	-
	-	-	-	-	-
tance	-	1536	1303268	237909	-
mory	-	-	-	-	-
ltiplexer	-	-	-	3062	-
egister	-	-	270677	-	-
Total	0	1536	1573945	241097	0
Available SLR	1344	3072	864000	432000	320
Utilization SLR (%)	0	50	182	55	0
Available	4032	9216	2592000	1296000	960
Utilization (%)	0	16	60	18	0

Figure: Comparison using “switch-case” on the left and “if-else” on the right

2. In the beginning, we observe that resource usage is overly high, even using a data center FPGA like U50. After a thorough inspection with a breakdown analysis of the coding structure, we found that the tool automatically synthesizes multiple kernels when invoking a single function many times. Therefore, we use the ALLOCATION pragma with limit=1 to restrict the instance number to 1 so that the read/write control logic and the 256 PE array are only initiated once. The below shows our method. This efficiently reduces resource usage.

```
void DPU::dpu_func(const uint8_t addr1, const uint8_t addr2, con
#pragma HLS allocation function instances=dpu_unit limit=1
#pragma HLS allocation function instances=read_poly limit=1
#pragma HLS allocation function instances=write_poly limit=1
```

3. To enable pipeline processing in the DPU PE array, we need to set false dependency. However, Vitis HLS cannot set false dependencies for members inside a called function. Therefore, we added INLINE pragma in all read/write functions to ensure that the dependency is correctly set. By using this method, we can finally pipeline the design and shorten the II.

7. Vitis HLS Build Flow

Please refer to `README.md`.

The Vitis HLS build flow is the same as in Lab 1.

8. Result

Synthesis Result

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
dpu_keygen				87646	87646	87646
dpu_keygen_Pipeline_VITIS_LOOP_40_1				256	256	256
dpu_keygen_Pipeline_VITIS_LOOP_41_2				256	256	256
dpu_keygen_Pipeline_VITIS_LOOP_321_1				25	25	25
dpu_keygen_Pipeline_VITIS_LOOP_42_3_VITIS_LOOP_43_4				2048	2048	2048
shake_absorb_1	1945	4390	1860	28	28	28
dpu_keygen_Pipeline_VITIS_LOOP_48_5_VITIS_LOOP_49_6				2048	2048	2048
dpu_keygen_Pipeline_VITIS_LOOP_54_7				257	257	257
dpu_keygen_Pipeline_VITIS_LOOP_55_8				257	257	257
dpu_keygen_Pipeline_VITIS_LOOP_56_9				257	257	257
shake_squeeze_2				101	101	101
dpu_func	271	660	28	238	658	26
dpu_keygen_Pipeline_VITIS_LOOP_55_5				32	32	32
dpu_keygen_Pipeline_VITIS_LOOP_321_110				25	25	25
dpu_keygen_Pipeline_VITIS_LOOP_60_6				32	32	32
dpu_keygen_Pipeline_VITIS_LOOP_61_7				32	32	32
dpu_pack				2725	2725	2725
shake_absorb_3				1734	1734	1734
shake_squeeze				77	77	77
dpu_keygen_Pipeline_VITIS_LOOP_62_8				48	48	48
dpu_pack_4	2149	2344	1954	2404	2918	1952
VITIS_LOOP_31_1_VITIS_LOOP_32_2				55828	55828	55828
VITIS_LOOP_37_3				6791	6791	6791
VITIS_LOOP_42_4				6669	6669	6669

dpu_func	271	660	28	238	658	26
dpu_func_Pipeline_FUNC_POW2ROUND_LOOP1				14	14	14
dpu_func_Pipeline_FUNC_CADDQ_LOOP1				13	13	13
dpu_func_Pipeline_FUNC_MONTMUL_LOOP1				13	13	13
dpu_func_Pipeline_FUNC_RD_LOOP1				13	13	13
dpu_func_Pipeline_FUNC_ADD_LOOP1				13	13	13
dpu_func_Pipeline_FUNC_MONTMUL_LOOP2				13	13	13
dpu_func_Pipeline_FUNC_MONTMUL_LOOP3				13	13	13
dpu_func_Pipeline_FUNC_MONTMUL_LOOP4				13	13	13
dpu_func_Pipeline_FUNC_RD_LOOP2				13	13	13
dpu_func_Pipeline_FUNC_RD_LOOP3				13	13	13
FUNC_MATMUL_LOOP0				475	475	475
FUNC_NTT_LOOP0				577	577	577
FUNC_INTT_LOOP0				657	657	657

We compared our implementation result with high-end CPU & state-of-the-art FPGA architecture. For CPU comparison, we use Intel Core i7-9700 to run the same algorithm and compare the execution cycle time for key generation. For state-of-the-art FPGA comparison, we estimate the reported result in ICFPT'21[3]. We can see that our HLS implementation can achieve a comparable result in a short development time.

Latency (cycle)

	CPU	This Work
Total Key Generation	**446475**	**87646**

	ICFPT'21[3]	This Work
NTT	**1831**	**577**
Matrix Multiplication	**1754**	**475**
Inverse NTT	**1475**	**670**

9. Conclusion

Post-quantum cryptography is a rapidly developing field with the potential to provide secure communication in the face of future quantum computers. The use of hardware acceleration can help to improve the performance of these methods, making them more practical for real-world use. By using high-level synthesis, we can evaluate the architecture performance in a short period of time. The tool can efficiently accelerate the process for architecture searching and shorten the time to market for IC design.

10. GitHub Link:

https://github.com/konosuba-lin/HLS_Dilithium