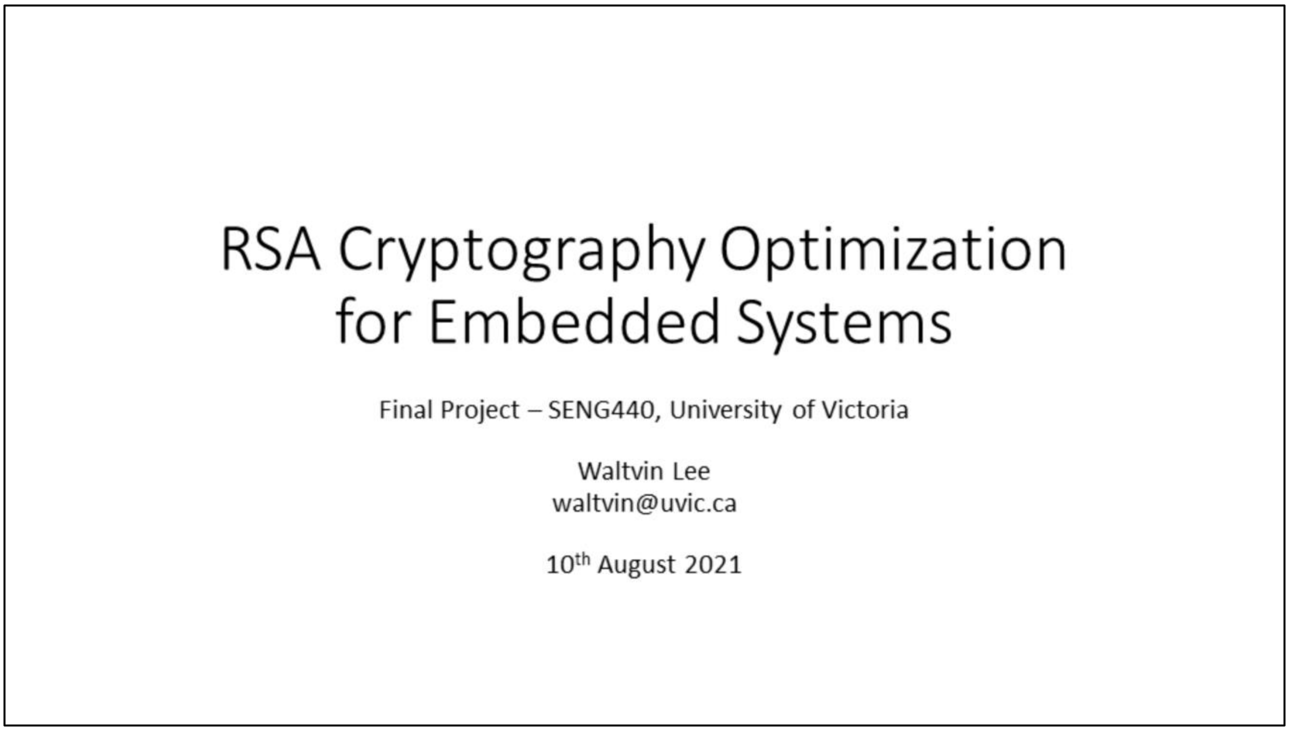


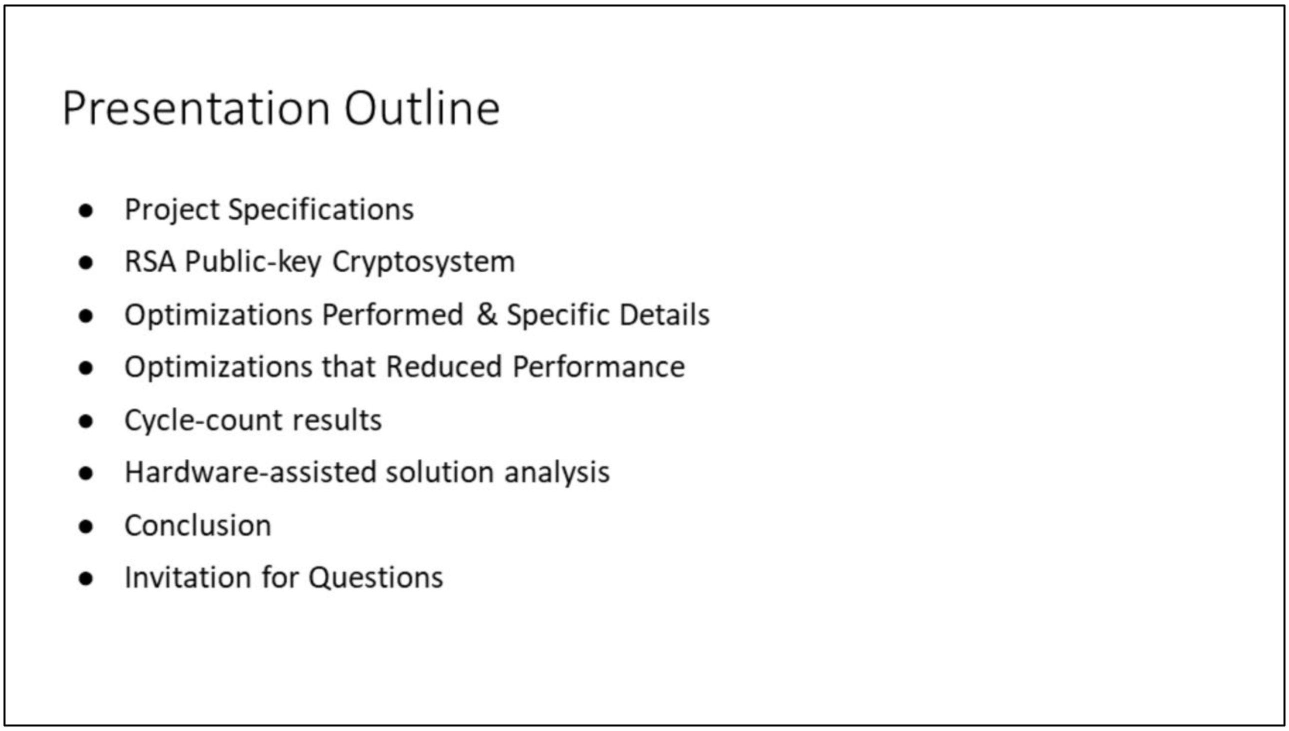
RSA Cryptography Optimization for Embedded Systems

Final Project – SENG440, University of Victoria

Waltvin Lee
waltvin@uvic.ca

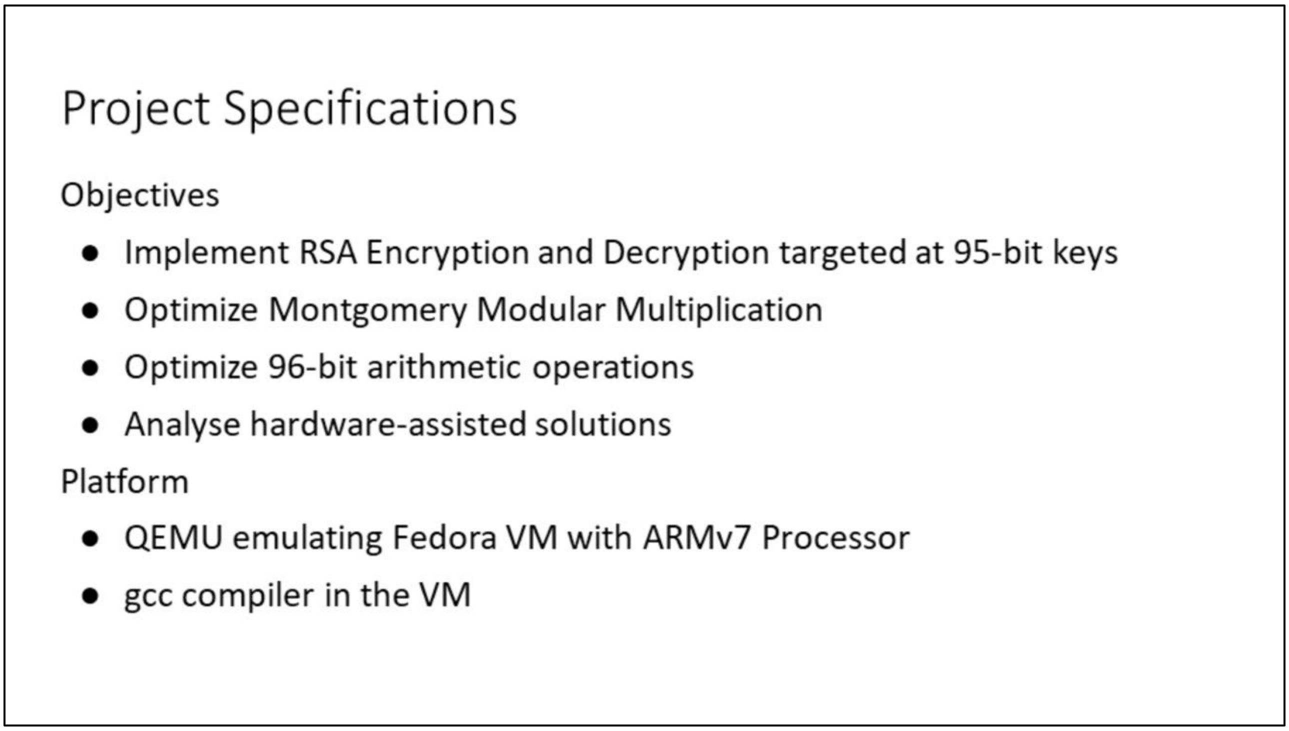
10th August 2021

My name is Waltvin Lee, and I will be presenting on RSA Cryptography Optimization for Embedded Systems, the final project for the course SENG440.



Presentation Outline

- Project Specifications
- RSA Public-key Cryptosystem
- Optimizations Performed & Specific Details
- Optimizations that Reduced Performance
- Cycle-count results
- Hardware-assisted solution analysis
- Conclusion
- Invitation for Questions



Project Specifications

Objectives

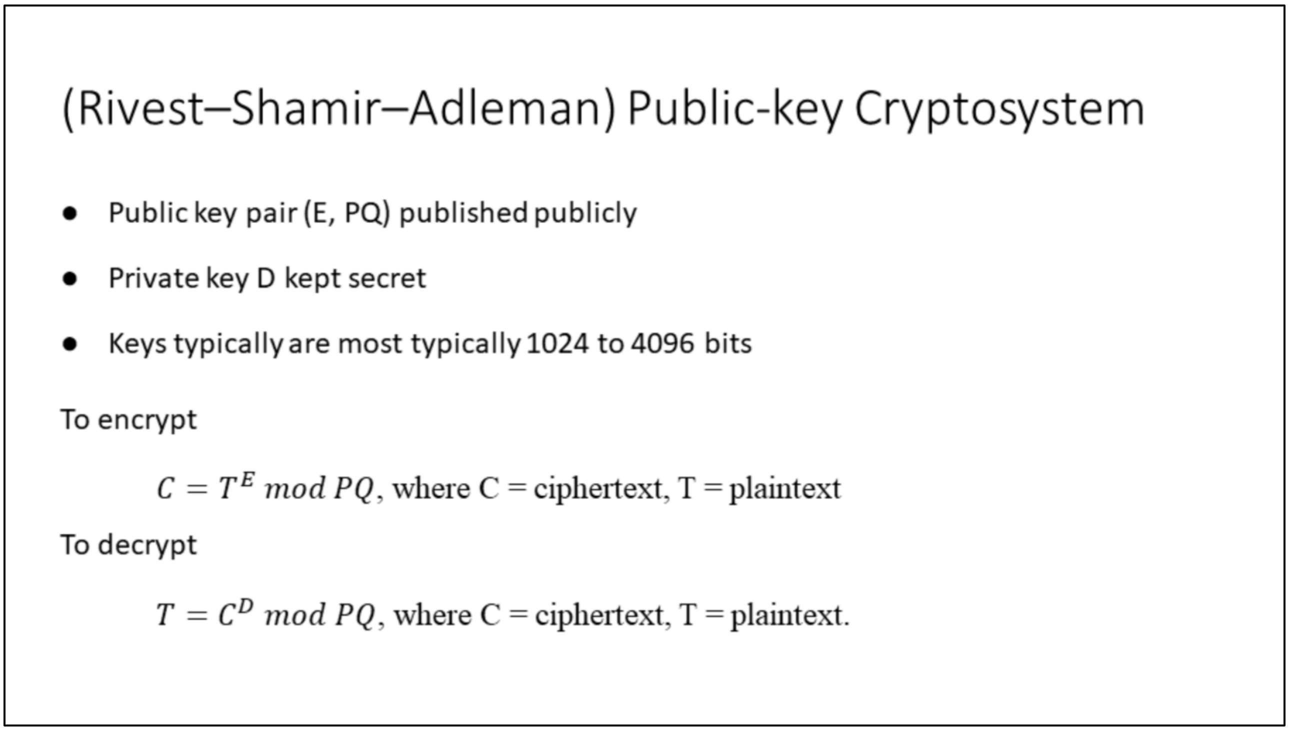
- Implement RSA Encryption and Decryption targeted at 95-bit keys
- Optimize Montgomery Modular Multiplication
- Optimize 96-bit arithmetic operations
- Analyse hardware-assisted solutions

Platform

- QEMU emulating Fedora VM with ARMv7 Processor
- gcc compiler in the VM

The project objective is to Implement RSA Encryption and Decryption targeted at 95-bit keys. Next is to optimize Montgomery Modular Multiplication and any 96-bit arithmetic operations it does. And finally to perform an analysis on hardware-assisted solution

The platform used in the project is a Fedora VM with an ARMv7 Processor emulated by QEMU. And the compiler that will be used is gcc.



(Rivest–Shamir–Adleman) Public-key Cryptosystem

- Public key pair (E, PQ) published publicly
- Private key D kept secret
- Keys typically are most typically 1024 to 4096 bits

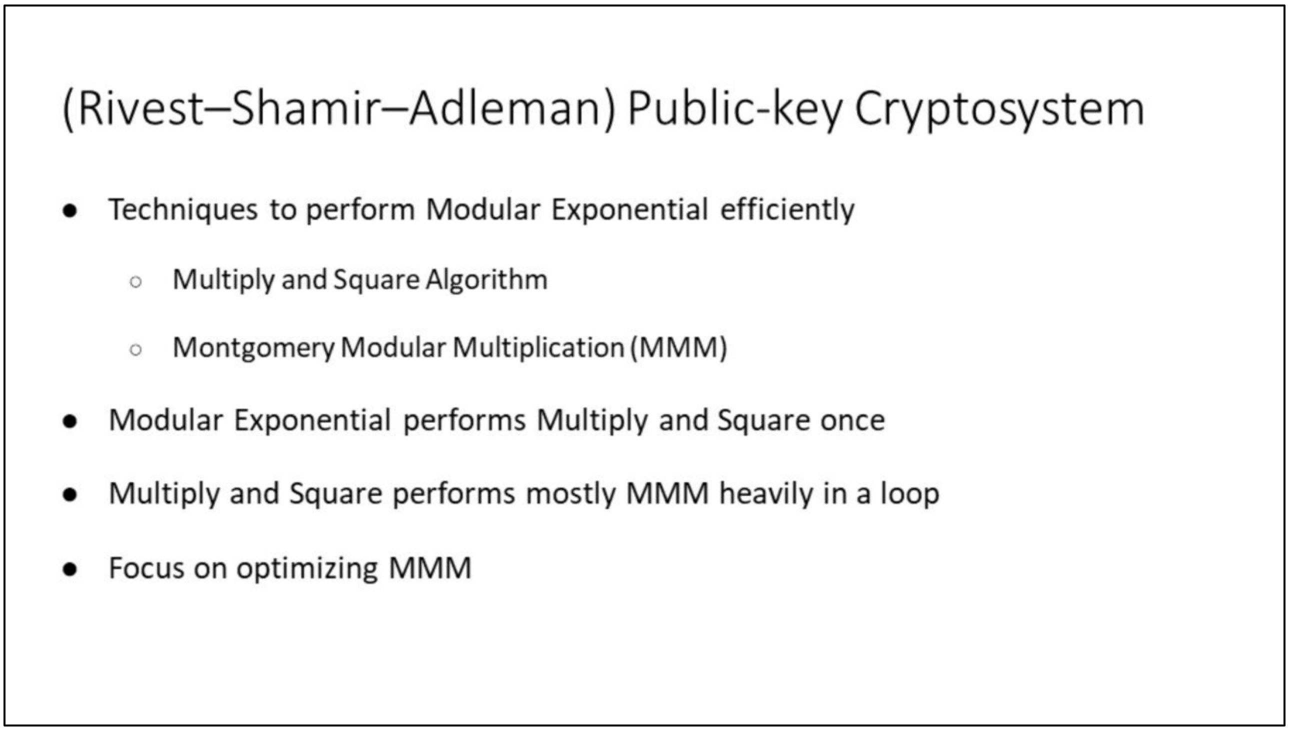
To encrypt

$C = T^E \, mod \, PQ$, where C = ciphertext, T = plaintext

To decrypt

$T = C^D \, mod \, PQ$, where C = ciphertext, T = plaintext.

The RSA Cryptosystem consists of a public key pair E and PQ, and a private key D. These keys are most typically large integers from 1024 to 4096 bits. To encrypt, we follow the formula $C = T^E \, mod \, PQ$, to decrypt, we follow the formula $T = C^D \, mod \, PQ$ where C = ciphertext, T = plaintext. Given that E, PQ, and D are very very large integers, we can see how this modular exponential operations will be very costly.

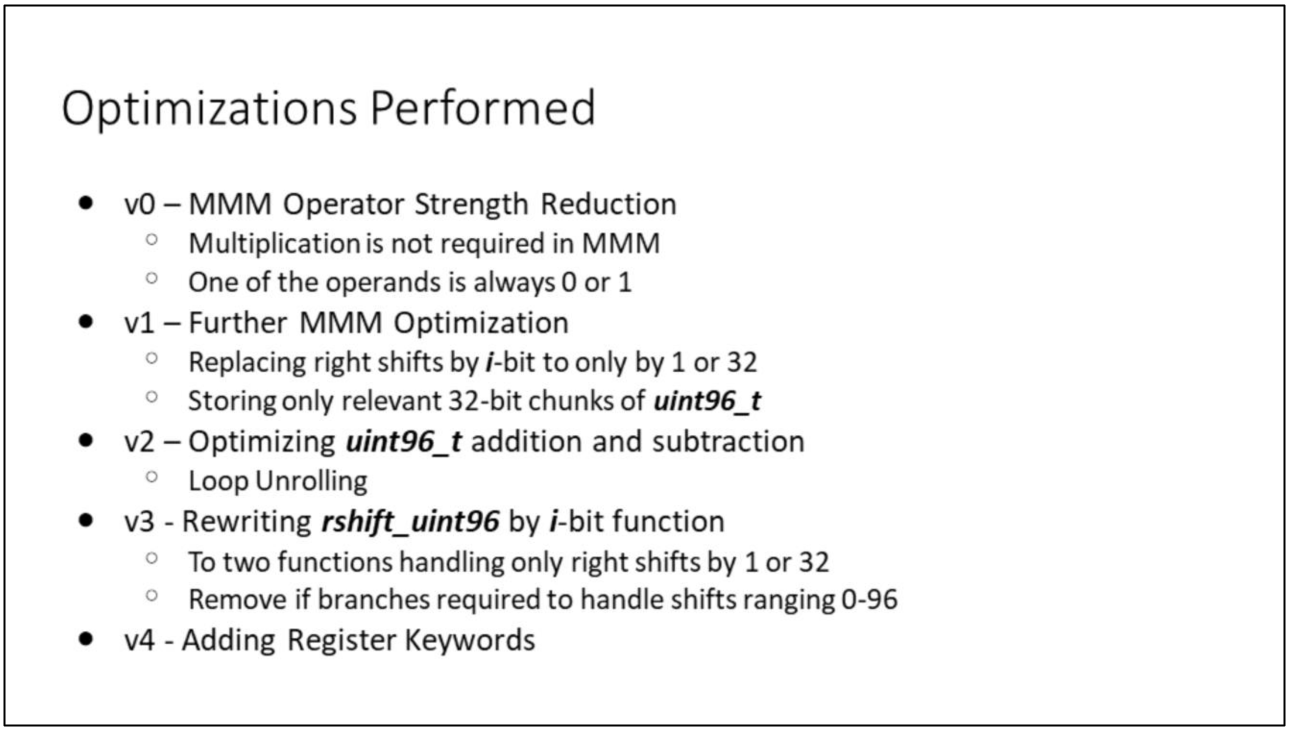


(Rivest–Shamir–Adleman) Public-key Cryptosystem

- Techniques to perform Modular Exponential efficiently
    - Multiply and Square Algorithm
    - Montgomery Modular Multiplication (MMM)
- Modular Exponential performs Multiply and Square once
- Multiply and Square performs mostly MMM heavily in a loop
- Focus on optimizing MMM

The two techniques used to help perform modular exponential efficiently are the multiply and square algorithm and MMM.

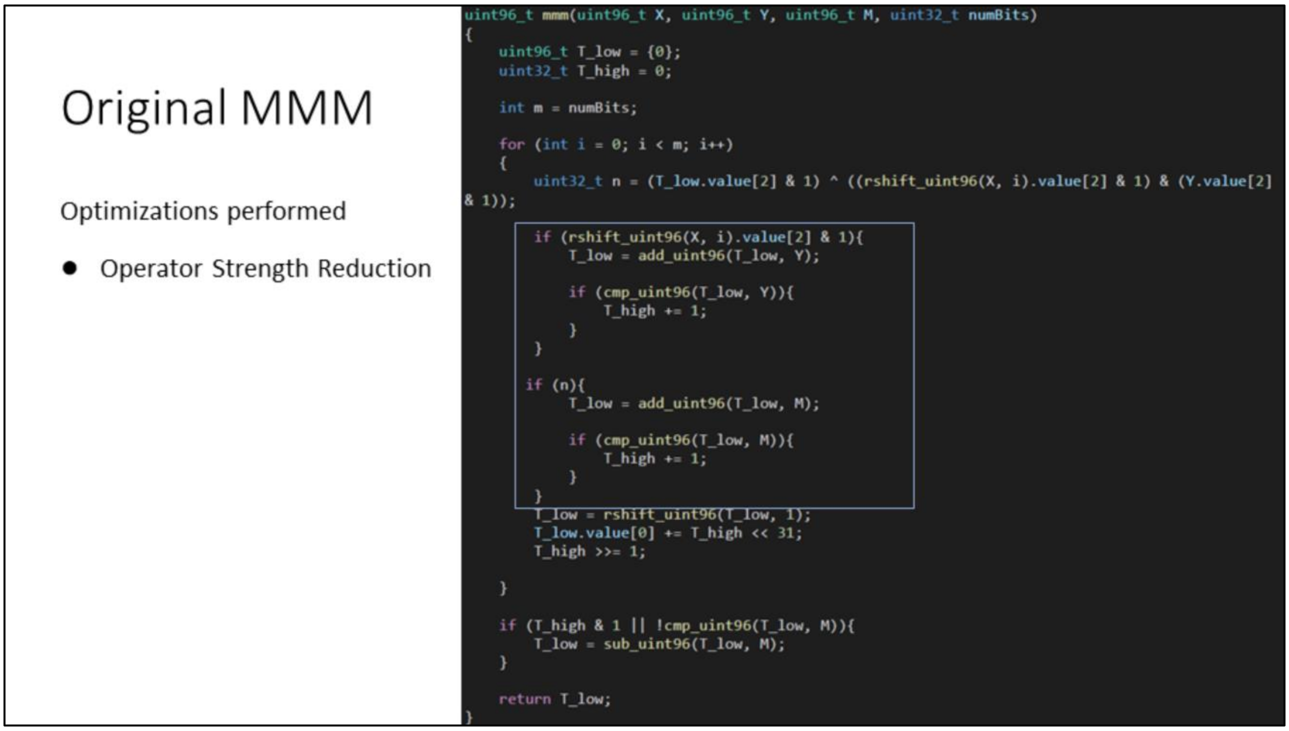
For every modular exponential, multiply and square algorithm will be called once, however, the multiply and square algorithm will require MMM, and will call MMM in a loop. Therefore MMM makes up for the majority of the process and thus the focus to optimize specifically MMM.


Optimizations Performed

- v0 – MMM Operator Strength Reduction
  - Multiplication is not required in MMM
  - One of the operands is always 0 or 1
- v1 – Further MMM Optimization
  - Replacing right shifts by ***i***-bit to only by 1 or 32
  - Storing only relevant 32-bit chunks of ***uint96_t***
- v2 – Optimizing ***uint96_t*** addition and subtraction
  - Loop Unrolling
- v3 - Rewriting ***rshift_uint96*** by ***i***-bit function
  - To two functions handling only right shifts by 1 or 32
  - Remove if branches required to handle shifts ranging 0-96
- v4 - Adding Register Keywords

These are the optimizations performed for the project

Firstly, Operator Strength Reduction is performed by replacing multiplication in MMM with if branching, as one of the operands is always 0 or 1.
Secondly, Further MMM Optimization is done by replacing right shift operations by *i*-bit to constants 1 or 32. And also choosing to store only relevant 32-bit chunks of the 96-bit types.
Third, Loop unrolling was performed to the 96-bit addition and subtraction routines.
Later, the 96 bit right shift function is rewritten to 2 separate functions handling only right shifts by 1 or 32. This removes the if branches that were required to handle shifts ranging from 0 to 96.
Finally, register keywords are added to help the compiler on choosing what to keep in registers.



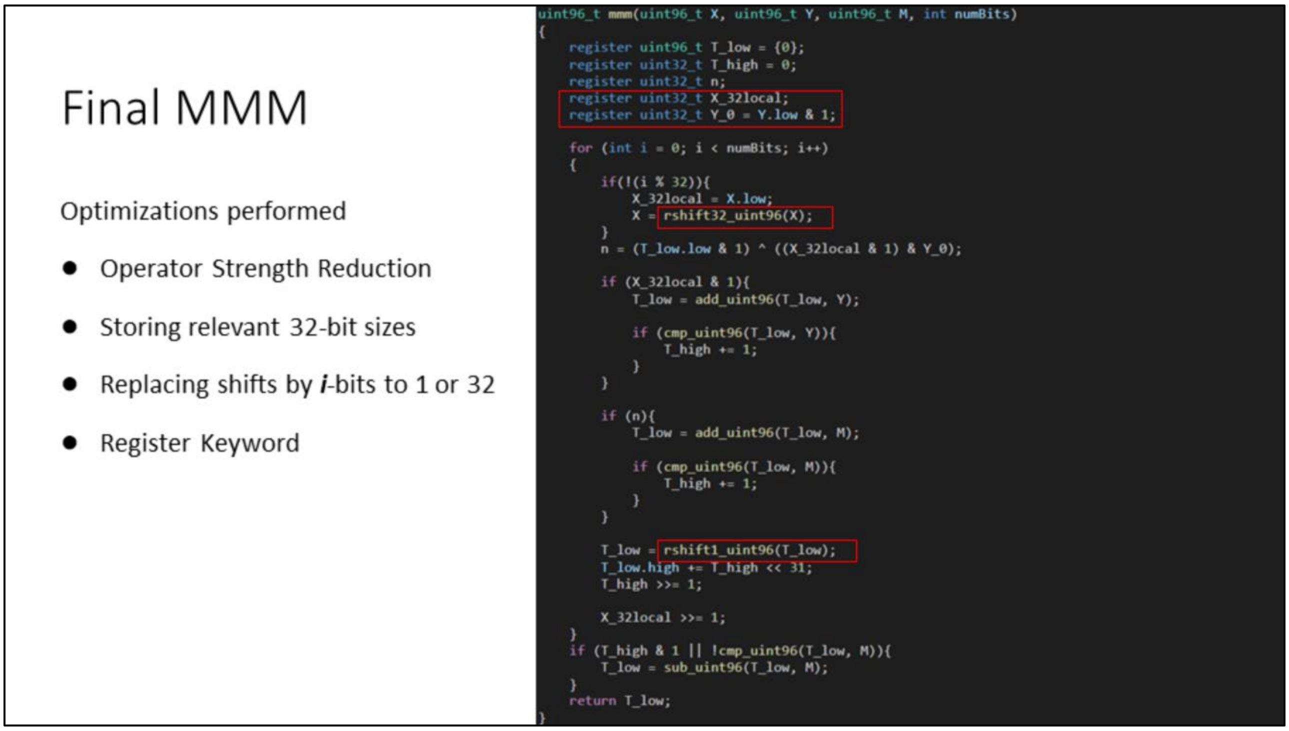
Original MMM

Optimizations performed

- Operator Strength Reduction

```
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, uint32_t numBits)
{
    uint96_t T_low = {0};
    uint32_t T_high = 0;

    int m = numBits;

    for (int i = 0; i < m; i++)
    {
        uint32_t n = (T_low.value[2] & 1) ^ ((rshift_uint96(X, i).value[2] & 1) & (Y.value[2]
& 1));

        if (rshift_uint96(X, i).value[2] & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

       if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }
        T_low = rshift_uint96(T_low, 1);
        T_low.value[0] += T_high << 31;
        T_high >>= 1;

    }

    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }

    return T_low;
}
```

Here is the original MMM code, with the only optimization performed being operator strength reduction replacing the multiplication operation with if branches, as seen in the blue box.



Final MMM

Optimizations performed

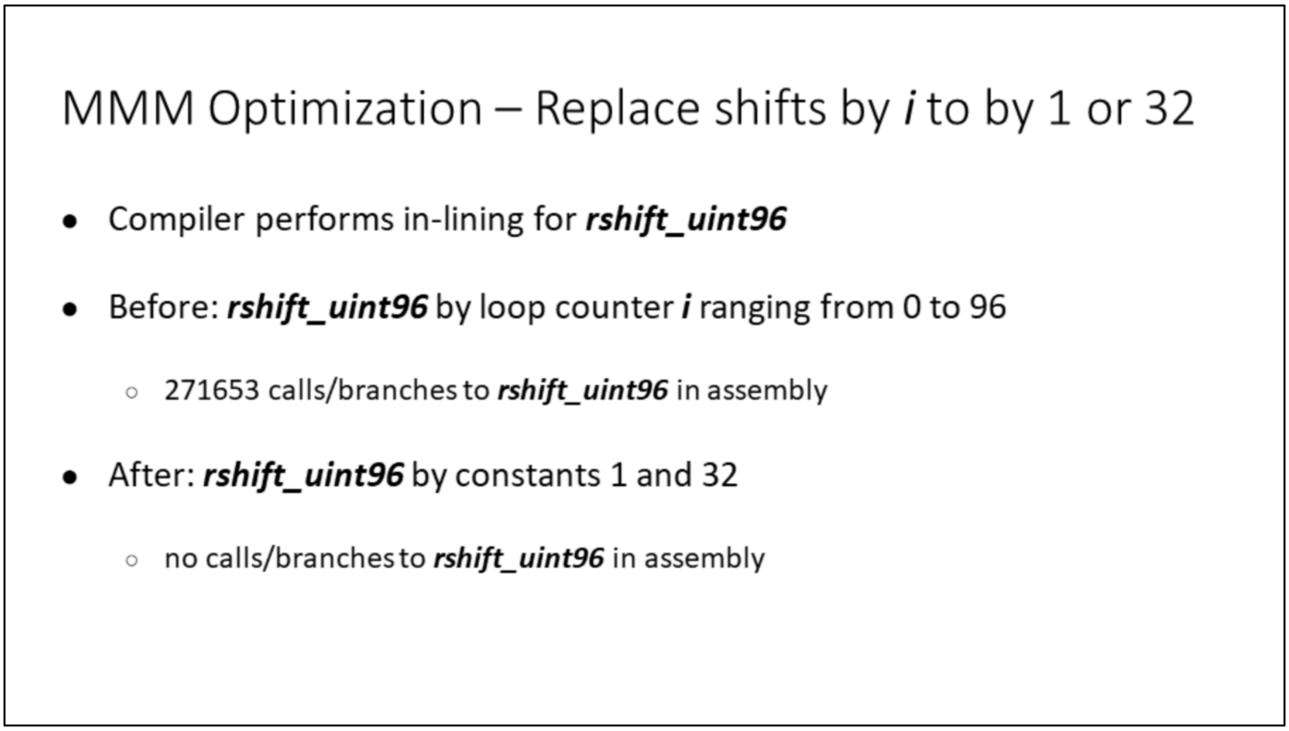
- Operator Strength Reduction
- Storing relevant 32-bit sizes
- Replacing shifts by ***i***-bits to 1 or 32
- Register Keyword

```c
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, int numBits)
{
    register uint96_t T_low = {0};
    register uint32_t T_high = 0;
    register uint32_t n;
    register uint32_t X_32local;
    register uint32_t Y_0 = Y.low & 1;

    for (int i = 0; i < numBits; i++)
    {
        if(!(i % 32)){
            X_32local = X.low;
            X = rshift32_uint96(X);
        }
        n = (T_low.low & 1) ^ ((X_32local & 1) & Y_0);

        if (X_32local & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

        if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }

        T_low = rshift1_uint96(T_low);
        T_low.high += T_high << 31;
        T_high >>= 1;

        X_32local >>= 1;
    }
    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }
    return T_low;
}
```

Here is the final MMM code after performing all optimizations. We can see that we now choose to store only relevant 32-bit sizes of the 96-bit variable X into variable named X_32local to be used in every loop iteration. This is done to hope that the compiler will store this 32-bit variable into a register instead of storing full 96-bit X.

Replacement of right shifts by loop counter i to only right shifts by 1 or 32 is also done, and this allows for compiler to perform in-lining, we will look at this more later.

Finally, register keywords was added to help the compiler in deciding what should be stored in registers.

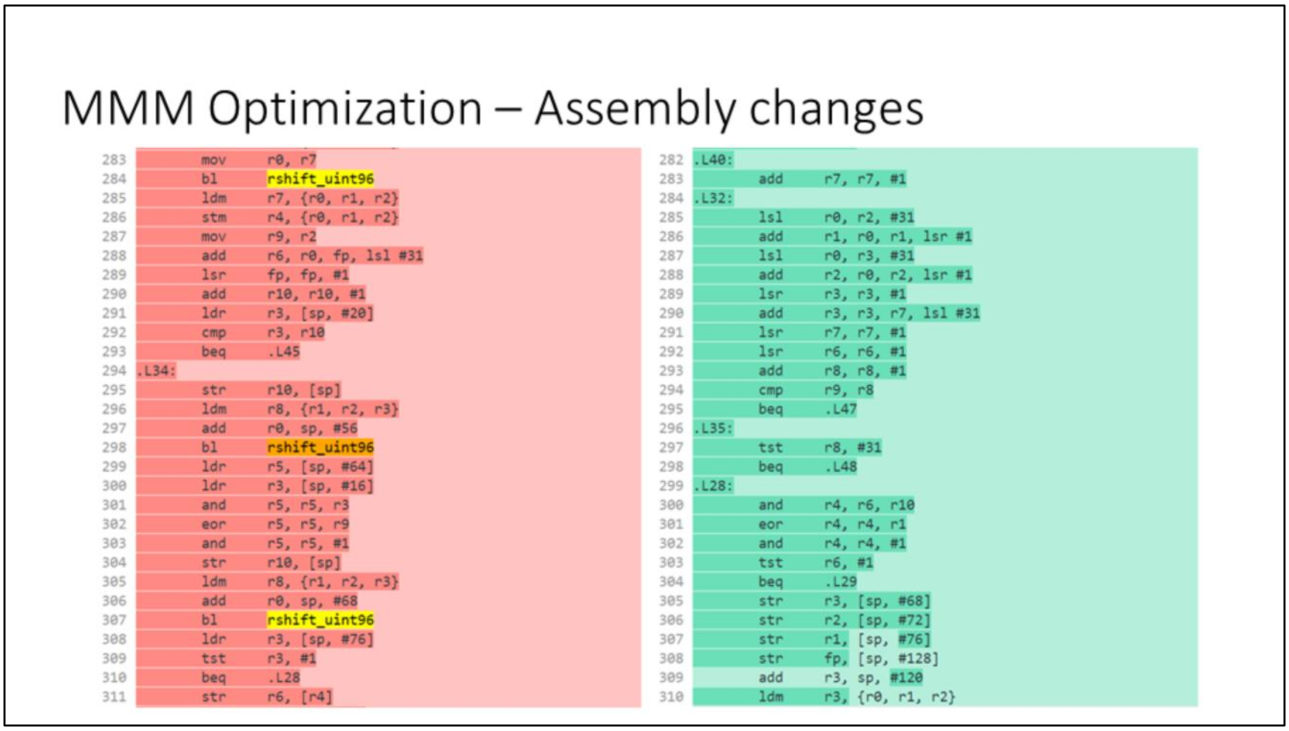


MMM Optimization – Replace shifts by ***i*** to by 1 or 32

- Compiler performs in-lining for ***rshift_uint96***
- Before: ***rshift_uint96*** by loop counter ***i*** ranging from 0 to 96
    - 271653 calls/branches to ***rshift_uint96*** in assembly
- After: ***rshift_uint96*** by constants 1 and 32
    - no calls/branches to ***rshift_uint96*** in assembly

Going into more details of the replacement of 96-bit right shifts by variable ***i*** to right shifts by constants 1 or 32

This optimization allowed the compiler to perform in-lining for the function.
We know this by looking at gprof reports. Before optimization, there were 200 thousand calls to the 96-bit right shift routine, after optimization, there were 0 calls reported by gprof.

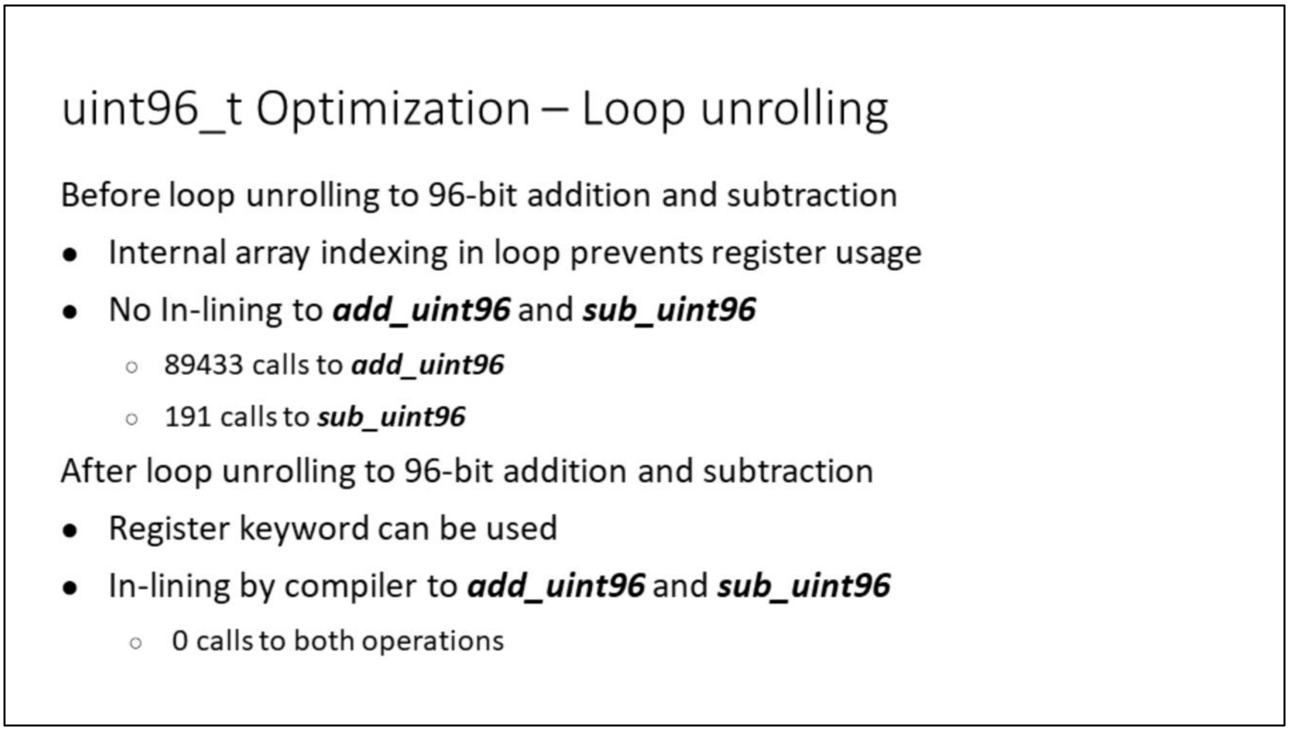


MMM Optimization – Assembly changes

```
        mov     r0, r7
        bl      rshift_uint96
        ldm     r7, {r0, r1, r2}
        stm     r4, {r0, r1, r2}
        mov     r9, r2
        add     r6, r0, fp, lsl #31
        lsr     fp, fp, #1
        add     r10, r10, #1
        ldr     r3, [sp, #20]
        cmp     r3, r10
        beq     .L45
.L34:
        str     r10, [sp]
        ldm     r8, {r1, r2, r3}
        add     r0, sp, #56
        bl      rshift_uint96
        ldr     r5, [sp, #64]
        ldr     r3, [sp, #16]
        and     r5, r5, r3
        eor     r5, r5, r9
        and     r5, r5, #1
        str     r10, [sp]
        ldm     r8, {r1, r2, r3}
        add     r0, sp, #68
        bl      rshift_uint96
        ldr     r3, [sp, #76]
        tst     r3, #1
        beq     .L28
        str     r6, [r4]
```

```
.L40:
        add     r7, r7, #1
.L32:
        lsl     r0, r2, #31
        add     r1, r0, r1, lsr #1
        lsl     r0, r3, #31
        add     r2, r0, r2, lsr #1
        lsr     r3, r3, #1
        add     r3, r3, r7, lsl #31
        lsr     r7, r7, #1
        lsr     r6, r6, #1
        add     r8, r8, #1
        cmp     r9, r8
        beq     .L47
.L35:
        tst     r8, #31
        beq     .L48
.L28:
        and     r4, r6, r10
        eor     r4, r4, r1
        and     r4, r4, #1
        tst     r6, #1
        beq     .L29
        str     r3, [sp, #68]
        str     r2, [sp, #72]
        str     r1, [sp, #76]
        str     fp, [sp, #128]
        add     r3, sp, #120
        ldm     r3, {r0, r1, r2}
```

- By quickly looking at the assembly, we can also see that the branches are removed and lsl and lsr operations are added directly.



uint96_t Optimization – Loop unrolling

Before loop unrolling to 96-bit addition and subtraction

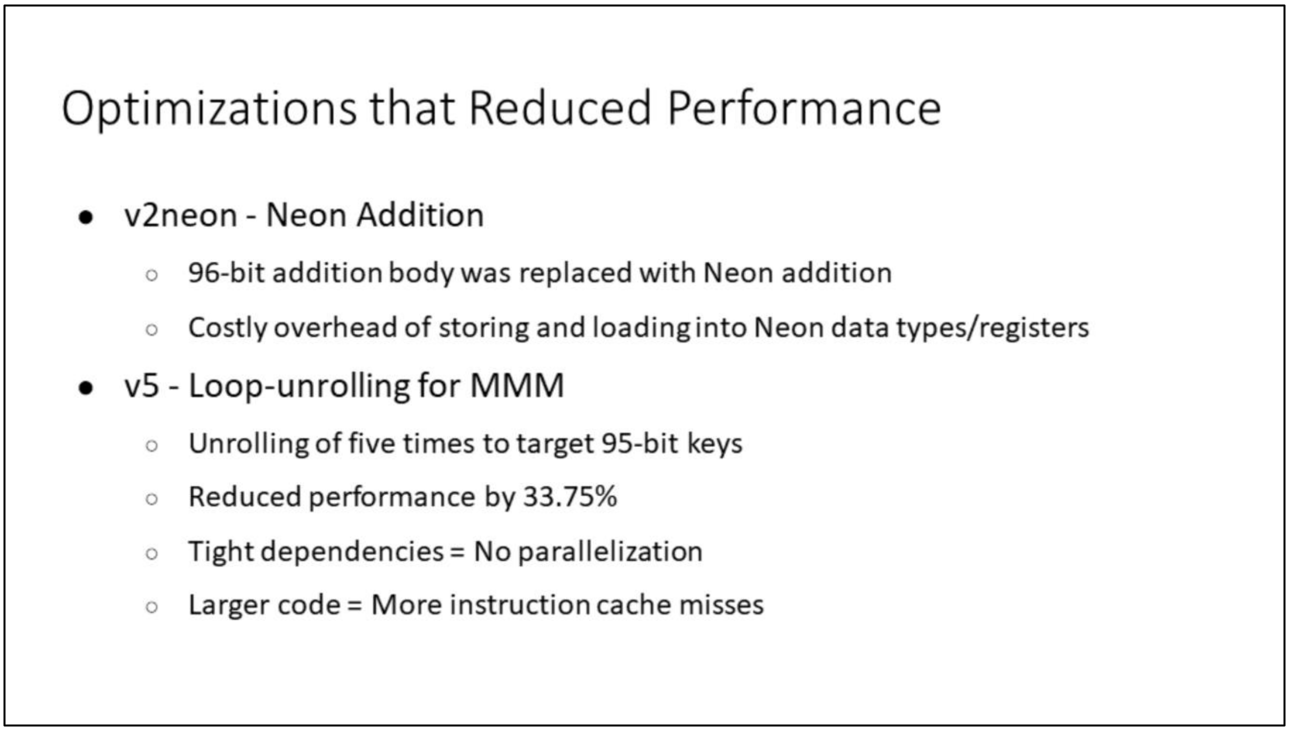
- Internal array indexing in loop prevents register usage
- No In-lining to ***add_uint96*** and ***sub_uint96***
    - 89433 calls to ***add_uint96***
    - 191 calls to ***sub_uint96***

After loop unrolling to 96-bit addition and subtraction

- Register keyword can be used
- In-lining by compiler to ***add_uint96*** and ***sub_uint96***
    - 0 calls to both operations

Now we look into more details on the optimizations made to the 96-bit operations add and subtract. Before performing loop unrolling, a loop is used to index the low, mid and high 32-bits of the 96-bit type. This however prevents register usage and the register keyword also can not be used. There were also no in-lining made by the compiler for the add and sub operations, causing large amount of calls to these operations.

After loop unrolling is performed to the add and sub operations, the register keyword can now be used, and in-lining was also made by the compiler to these operations, resulting in 0 calls to both operations reported by gprof.



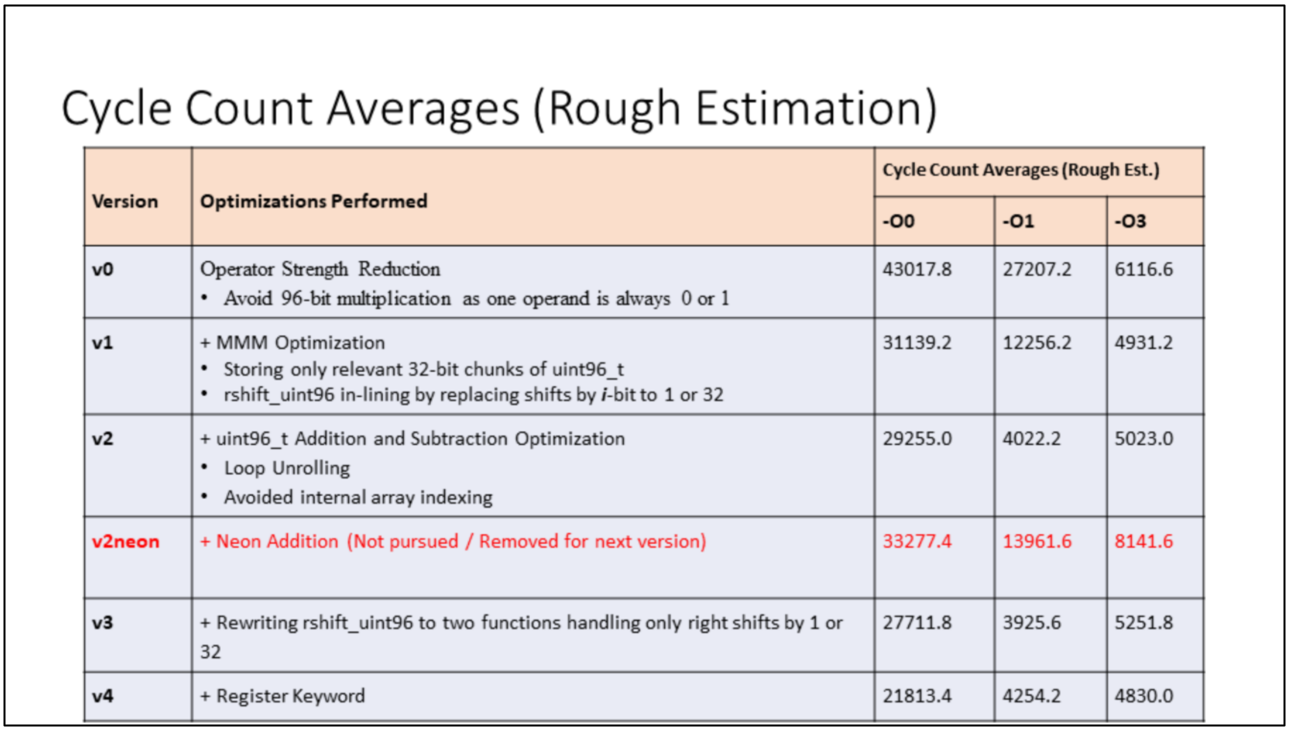
Optimizations that Reduced Performance

- v2neon - Neon Addition
  - 96-bit addition body was replaced with Neon addition
  - Costly overhead of storing and loading into Neon data types/registers
- v5 - Loop-unrolling for MMM
  - Unrolling of five times to target 95-bit keys
  - Reduced performance by 33.75%
  - Tight dependencies = No parallelization
  - Larger code = More instruction cache misses

I will now go through some of the optimization attempts that reduced performance.

Neon addition was attempted by implementing it into 96-bit addition routine. However, the overhead of storing and loading into neon data types/registers from general datatypes are too costly.
A consideration was made to work entirely in Neon datatypes, but this will result in one register being wasted as Neon has 64-bit or 128-bit registers and we are working with 96-bits

Loop-unrolling was attempted for the loop in MMM function.
Given that the target key size for this project was 95-bit keys, the loop was unrolled five times, because 5 is the lowest divisor of 95 and the loop will run for 95 iterations. Doing this however reduced performance by 33.75%. This is probably because there are tight dependencies in MMM, and no parallelization could be benefited even after performing loop unrolling. Loop unrolling by 5 times also increased the code size by around 400 assembly lines, and this may have resulted in more instruction cache misses.



Cycle Count Averages (Rough Estimation)

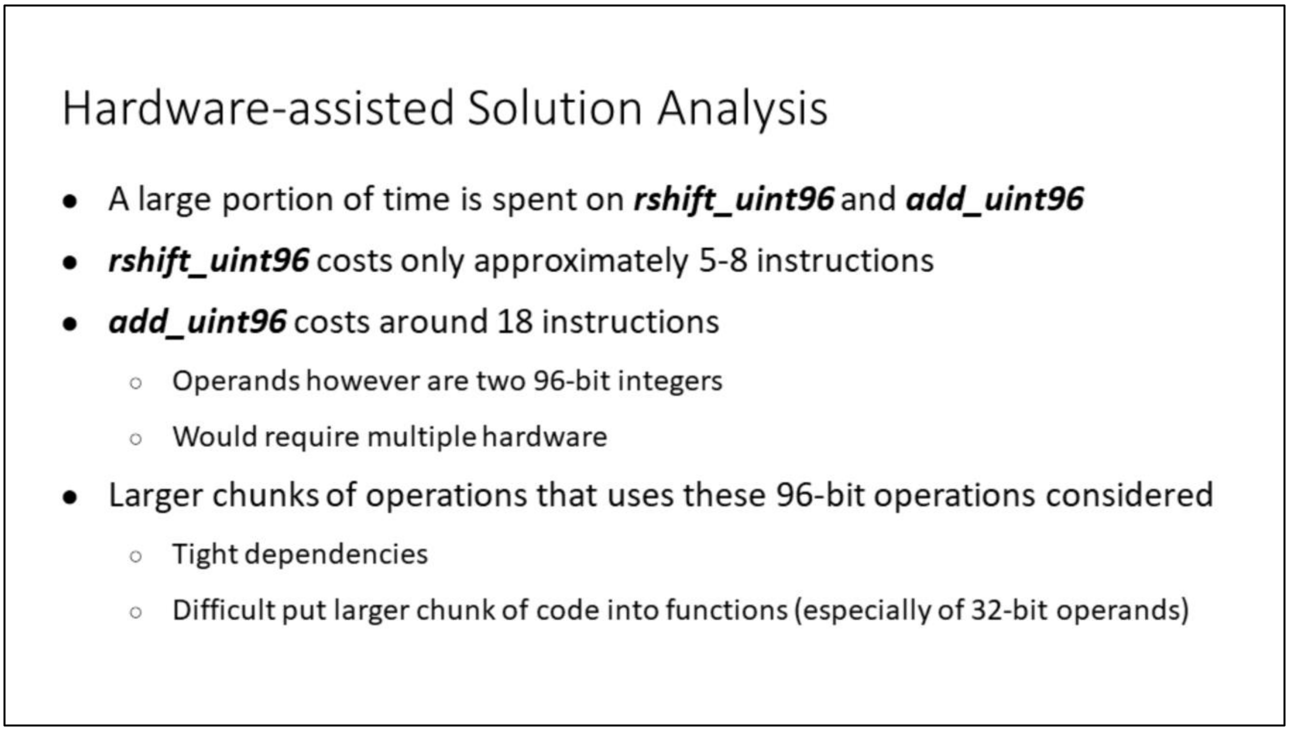
Version	Optimizations Performed	Cycle Count Averages (Rough Est.)		
		-O0	-O1	-O3
v0	Operator Strength Reduction • Avoid 96-bit multiplication as one operand is always 0 or 1	43017.8	27207.2	6116.6
v1	+ MMM Optimization • Storing only relevant 32-bit chunks of uint96_t • rshift_uint96 in-lining by replacing shifts by *i*-bit to 1 or 32	31139.2	12256.2	4931.2
v2	+ uint96_t Addition and Subtraction Optimization • Loop Unrolling • Avoided internal array indexing	29255.0	4022.2	5023.0
v2neon	+ Neon Addition (Not pursued / Removed for next version)	33277.4	13961.6	8141.6
v3	+ Rewriting rshift_uint96 to two functions handling only right shifts by 1 or 32	27711.8	3925.6	5251.8
v4	+ Register Keyword	21813.4	4254.2	4830.0

From the results, MMM optimization seems to provide positive results for all situations. Neon Addition on the other hand reduced the performance.

Other optimizations seem to provide improvements in some optimization levels but remain the same for others. This is expected as some optimization level may already include the optimization that was manually performed.

For example, in v3, rewriting the 96-bit right shift function into two functions handling only shifts of 1 or 32 removes the if branches in the function. This actually benefited the –O0 flag because it does not perform in-lining optimization for this function and does not automatically optimize away the if branches even if only constants 1 or 32 is used as the argument.

An interesting observation is that after performing manual loop unrolling, -O1’s performance seems to be better than -O3. This is probably because -O3 performed some extra optimizations that were counterproductive.

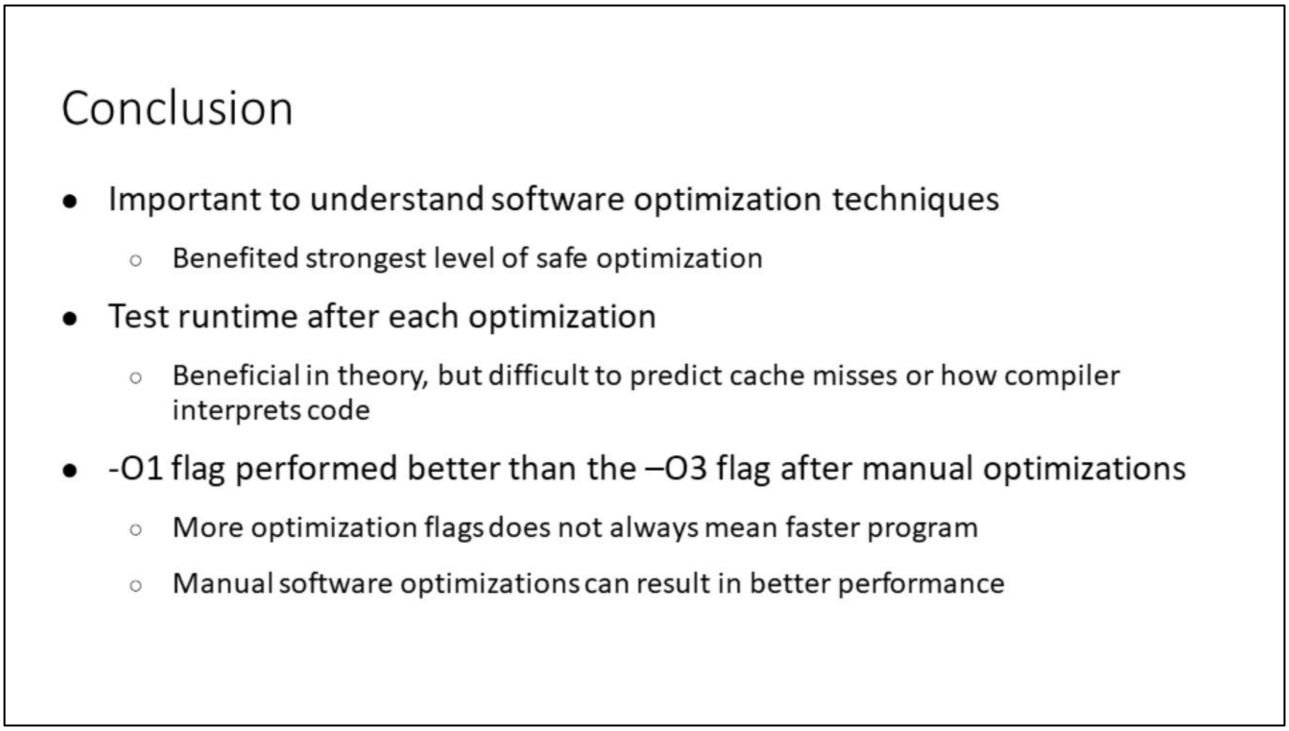


Hardware-assisted Solution Analysis

- A large portion of time is spent on ***rshift_uint96*** and ***add_uint96***
- ***rshift_uint96*** costs only approximately 5-8 instructions
- ***add_uint96*** costs around 18 instructions
  - Operands however are two 96-bit integers
  - Would require multiple hardware
- Larger chunks of operations that uses these 96-bit operations considered
  - Tight dependencies
  - Difficult put larger chunk of code into functions (especially of 32-bit operands)

For the hardware-assisted solution, I first looked at what specific functions are being used most often. The largest amount of time are spent on 96-bit right shifts and add operations, however, these operations take only 5 – 20 instructions, and furthermore these functions use 96-bit operands, and it is difficult to create hardware-assisted solution when it is limited to only 32-bit operands.

Larger chunks of operations that uses these smaller operations were considered, but in MMM there are tight dependencies in every line, and it was extremely difficult see which larger chunk of code can be made into functions, and especially harder with 32-bit operands given that we were working with 96-bit types.



Conclusion

- Important to understand software optimization techniques
    - Benefited strongest level of safe optimization
- Test runtime after each optimization
    - Beneficial in theory, but difficult to predict cache misses or how compiler interprets code
- -O1 flag performed better than the –O3 flag after manual optimizations
    - More optimization flags does not always mean faster program
    - Manual software optimizations can result in better performance

To conclude, this project showed that it is important to understand and apply software optimizations techniques manually and not heavily rely on compiler’s optimization. Although compilers today optimize very efficiently, manual software optimization has improved performance for even the strongest level of safe optimization that the gcc compiler can offer.

It is also important to test the program’s runtime after each optimization step because some optimization techniques seem beneficial in theory, but it is difficult to predict how the compiler will compile the code or any instruction cache misses that come up, and these might result in performance loss.

Given that the results showed that –O1 flag performed better than the –O3 flag after manual optimizations, we know that using more optimization flags does not necessarily always mean faster program. More importantly, performing manual software optimizations can result in better performance than just adding more compiler optimization flags.

Any Questions?

That will be the end of the presentation. Thank you for listening. Are there any questions?