UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Accelerating FHE Operations on a Processing-in-Memory System

Beschleunigung von FHE-Operationen auf einem Processing-in-Memory System

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Niklas Klinger

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Jonas Sander

Lübeck, den 11. Dezember 2024

Abstract

Fully homomorphic encryption (FHE) is a promising technology for secure cloud computing, as it allows computations directly on encrypted data. However, FHE is computationally expensive and often memory bound on conventional computer architectures. Processing-in-Memory (PIM) is an alternative hardware architecture which integrates processing units and memory on the same chip or memory module. PIM enables higher memory bandwidth than conventional architectures and could thus be suitable for accelerating FHE. In this work, we test UPMEM's programmable, general-purpose PIM system and evaluate its suitability for accelerating FHE operations. We incorporate many optimisations, including residue number system and number-theoretic transform techniques and achieve over $5\times$ speed-up compared to previous work. Additionally, we benchmark the runtime and energy efficiency of our implementation against Microsoft SEAL, a popular open-source FHE library. Our results show that the current version of UPMEM PIM is unsuitable for accelerating FHE operations compared to other optimised implementations, because it is constrained by multiplication performance.

Zusammenfassung

Vollständig homomorphe Verschlüsselung (engl. FHE) ist eine vielversprechende Technologie für sicheres Cloud Computing, da sie Berechnungen direkt auf verschlüsselten Daten ermöglicht. FHE ist jedoch rechenintensiv und auf konventionellen Computerarchitekturen oft durch die Leistung des Speichers beschränkt. Processing-in-Memory (PIM) ist eine alternative Hardwarearchitektur, die Recheneinheiten und Speicher auf demselben Chip oder Speichermodul integriert. PIM ermöglicht eine höhere Speicherbandbreite als konventionelle Architekturen und könnte somit geeignet sein, um FHE zu beschleunigen. In dieser Arbeit betrachten wir das programmierbare PIM-System von UPMEM und evaluieren, ob es zur Beschleunigung von FHE-Operationen geeignet ist. Wir verwenden viele Optimierungen, unter anderem Restklassendarstellungen und zahlentheoretische Transformationen (diskrete Fouriertransformationen über einem Ring), wodurch wir eine $5\times$ Beschleunigung im Vergleich zu früheren Arbeiten erreichen. Außerdem vergleichen wir unsere Implementierung bezüglich Laufzeit und Energieeffizienz mit Microsoft SEAL, einer bekannten open-source FHE-Bibliothek. Unsere Ergebnisse zeigen, dass die aktuelle Version von UPMEM PIM aufgrund ihrer begrenzten Multiplikationsgeschwindigkeit nicht geeignet ist, um FHE-Operationen im Vergleich zu anderen optimierten Implementierungen zu beschleunigen.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Lübeck, 11. Dezember 2024

Contents

1 Introduction

Data security is becoming increasingly important as more applications rely on cloud computing. Especially in areas such as healthcare, where confidentiality and privacy of sensitive data is critical. Homomorphic encryption (HE) allows computations to be performed on encrypted data without decrypting it and without revealing any information about the inputs and outputs of the computation apart from their lengths. The results can be decrypted using the secret key corresponding to the original encrypted data. Because HE allows computations directly on encrypted data, it can significantly enhance security in cloud computing contexts. Using HE, even highly sensitive computations can be outsourced to cloud providers or other third parties, because the data can stay encrypted throughout the whole process. This is particularly relevant in healthcare, where the handling of sensitive medical records and patient information must comply with strict data protection regulations, such as the General Data Protection Regulation (GDPR) in the European Union. HE could enable secure data analysis, research and healthcare services in the cloud, while ensuring compliance with increasingly stringent privacy laws around the world.

To achieve confidentiality, current HE schemes rely on noise which is added to the encrypted data. When performing computations on this data, the noise compounds, which poses a limit on how many operations can be performed, as too much noise makes the results impossible to decrypt. Fully homomorphic encryption (FHE) solves this problem by introducing a special noise reducing operation called bootstrapping. This allows arbitrary sequences of operations to be performed on the encrypted data, as long as bootstrapping is performed at appropriate times. However, the main drawback of FHE is its high memory- and computational-intensity compared to working directly on the unencrypted data. Bootstrapping operations are especially expensive and thus responsible for the majority of the time required for typical computations on homomorphically encrypted data.

Processing-in-Memory (PIM) is a hardware architecture which integrates processing units and memory on the same chip or memory module. PIM enables low-latency, high-bandwidth memory access compared to a traditional architecture, in which processing units and memory are separated. PIM could thus be suitable for accelerating memory-intensive applications like FHE, which we want to evaluate in this work.

1.1 Motivation

Traditional computer architectures can become bottlenecked by memory accesses, which means that some of their compute units have to stay idle, because they cannot receive enough data to operate on. Memory bottlenecks often occur when performing FHE operations, which have comparatively low *arithmetic intensity* [dCAY+21], as they access a lot of data, but only perform few operations on it, before requiring more data. Additionally, some FHE operands, especially those required for bootstrapping, can be too large for the caches of these traditional architectures, which further decreases their memory performance. To some extend, this can be remedied by ever bigger caches, wider memory buses and higher memory frequencies to increase the bandwidth between memory and compute units. However, this data movement has high energy costs and already accounts for more than half of the total energy usage in many applications [BGK+18].
As an alternative approach, PIM architectures "bring memory and compute together" and can thus achieve high bandwidths without the extensive memory systems of traditional architectures. UPMEM[1], whose PIM architecture we will evaluate, claims significant speed-ups and improved energy efficiency in many memory intensive applications [UPM23]. Their general-purpose architecture may thus also be suitable for accelerating FHE operations by eliminating the memory bottleneck. However, the UPMEM PIM system also presents challenges: The system is massively parallel, the processing units are simple, comparatively weak, cannot easily communicate with each other, and their memory is split and must be managed manually (see Section 2.2). We will have to address these challenges when implementing FHE operations on the UPMEM PIM system.

1.2 Research Focus

We want to evaluate whether the UPMEM PIM system is suitable for accelerating FHE operations. Specifically, we want to answer the following research question: Is the UPMEM PIM system suitable for improving the throughput or energy efficiency of FHE operations compared to other optimised implementations?
Since current FHE schemes operate on polynomial rings, we will focus on the performance of the UPMEM PIM system for polynomial additions, multiplications and modulus operations. We will also evaluate how different parameters of the encryption schemes and the corresponding properties of the ciphertexts affect the suitability of the UPMEM PIM system for accelerating these operations. This includes the polynomial modulus and thus the length of the ciphertexts (e.g. 1024, 4096, or more coefficients) and the ciphertext

[1]UPMEM Homepage: `https://www.upmem.com/`

moduli, i.e. the value range of the coefficients and specifically whether they fit into common word sizes (32-bit, 64-bit, etc.) and how these parameters can be adjusted to improve performance.

1.3 Related Work

Many different ways of accelerating FHE operations have been explored, including CPU optimisations using wide-register SIMD instruction sets like SSE and AVX for parallelisation, GPU implementations for even further parallelisation [BVL+21, JKA+21], and hardware solutions such as FPGAs [PNPM15, CRS17] and ASICs. There has also been research on the bottlenecks for further FHE optimisations, with the general conclusion being that FHE is memory-bound [dCAY+21]. This paints a promising picture for accelerating FHE using in- or near-memory computing systems like UPMEM PIM.
Previous work on accelerating FHE using PIM, like CryptoPIM [NGI+20] and MeNTT [LPY22], has focused on custom hardware designs. We will instead evaluate FHE acceleration using a general-purpose PIM system and without designing custom circuits. A team of researches at ETH Zürich has already published a short evaluation of HE operations on an UPMEM PIM system [GKG+23]. We want to build on this work, which to our knowledge is currently the only paper on this subject.
Additionally, there are algorithmic and mathematical approaches for reducing the cost of FHE operations or polynomial operations in general, like the (fast) number-theoretic transform (NTT), which are used by many FHE implementations. We will implement these as well, so that we can accurately compare the UPMEM PIM system with existing FHE implementations.

2 Background

In this chapter, we go into more detail on FHE and the UPMEM architecture.

2.1 Fully Homomorphic Encryption (FHE)

Many current FHE schemes are based on the Learning With Errors (LWE) or Ring Learning With Errors (RLWE) problem and use noisy polynomials as ciphertexts. RLWE based FHE schemes like BFV [Bra12, FV12] and BGV [BGV14] operate on polynomial rings denoted as $\mathbb{Z}_q[x]/(x^n + 1)$, i.e. the ring of integer polynomials modulo the polynomial $x^n + 1$ and with coefficients modulo q (other rings are possible, but this is the most popular choice). The two main operations on the ciphertexts, which are required for the scheme to be fully homomorphic, require polynomial addition and polynomial multiplication. It may also be required to perform polynomial modulus and coefficient modulus operations, to ensure that the ciphertexts stay within the relevant polynomial ring. Other operations like bootstrapping can be separated into multiple of these basic operations.

2.1.1 Steps in an FHE Scheme

Usage of an FHE scheme can be logically split into the following steps:

1. Selecting encryption parameters
2. Key generation
3. Encrypting data
4. Computing on encrypted data
5. Decrypting results

In this work, we consider a scenario in which a client wants to offload computations to an untrusted server (e.g. a cloud provider) using FHE. This means that key generation, encryption and decryption are performed by the client, while the server handles computations on the encrypted data and potentially specifies encryption parameters that the client should use. Our focus is the computation on encrypted data, which corresponds to the server side of this interaction.

2.2 The UPMEM Architecture

UPMEM PIM is a programmable near-memory computing architecture (as opposed to in-memory computing), in which simple, general-purpose processors, called Data Processing Units (DPUs), are co-located with DRAM chips on special memory modules (PIM DIMMs). The DPUs are custom 32-bit RISC processors running at up to 400 MHz. They support 16 independent threads[2] and up to 11 of these threads can run concurrently at any given time (see Section 2.2.2). Each PIM DIMM has a capacity of 8 GB and contains 128 DPUs, with each DPU having access to a 64 MB slice of the module's main RAM (MRAM). However, DPUs can only directly compute on an additional, smaller memory, called the working memory (WRAM), of which each DPU has 64 KB. Special direct memory access (DMA) instructions are used to transfer data between the WRAM and MRAM of a DPU. Current UPMEM platforms support up to 20 PIM DIMMs, for a total of 2560 DPUs and a capacity of 160 GB.

2.2.1 Arithmetic Operations

UPMEM DPUs have a native word size of 32-bit and can perform simple operations like addition, subtraction and bitwise manipulation (and, or, not, shifts, etc.) of 32-bit integers with single instructions. Using instructions like add-with-carry, these operations can also easily be performed on bigger integers. However, the DPU hardware does not directly support 32x32-bit or even 16x16-bit integer multiplications. The hardware multiplier can only perform 8x8-bit (byte) multiplications, which yield 16-bit results. The DPU compiler implements 16x16-bit and 64x64-bit multiplications by chaining together many of these byte multiplications and appropriately shifting and adding the intermediate results. For 32x32-bit multiplication, the compiler uses `mul_step` instructions instead, which works as follows: Consider the calculation of $A \cdot B$, where the result is initially zero. If A's n-th bit is set, then $2^n \cdot B$ is added to the result. This step is repeated for all $n \in \{0, \ldots, 31\}$, which takes one cycle each. As an optimisation, A is always set to the smaller of the two factors and the algorithm stops as soon as all remaining (upper) bits of A are zero. Integer division and modulo operations are implemented similarly, using `div_step` instructions for operands up to 32-bit and a software implementation for 64-bit operands. The DPUs do not have hardware supported floating point operations; they are implemented in software instead, which results in poor performance. As such, floating point operations should be avoided for most applications.

[2]These capabilities refer to the v1B DPU model. UPMEM also provides a v1A model with slight differences. (See `https://sdk.upmem.com/2024.1.0/03_ProgrammingWithUpmemDpu.html#dpu-chip-characteristics`)

2.2.2 Threading and Pipeline

DPUs feature a 14-stage pipeline, but the three first and last stages of dependent instructions can execute in parallel [GHF+21]. Thus, only 11 active threads are needed to fully saturate the pipeline. Active threads are scheduled in a round-robin way and threads generally stay active[3] unless they are explicitly waiting (e.g. on a mutex) or performing a DMA operation. These DMA operations execute sequentially and threads waiting on a DMA operation are placed in a special queue. This means that only one thread of each DPU can access the MRAM at a time.

2.2.3 Communication Between DPUs and Host

DPUs can communicate with the host CPU via the DDR interface. Inter-DPU communication is not possible directly and must go through the host CPU instead, meaning that the data must first be copied from one DPU to the CPU's main memory and then copied from the main memory to another DPU. This process is slow, as it requires synchronisation between all three components. Thus, inter-DPU communication should be avoided whenever possible.
As with normal DRAM modules (DIMMs), the memory chips on PIM DIMMs are organised into ranks and banks, interleaving multiple 8-bit wide chips to achieve a total bus width of 64-bit. Logically consecutive bytes are thus actually stored in different memory chips belonging to different DPUs, as shown in Figure 2.1. Transferring continuous data between DPUs and the host CPU therefore requires a transposition step. The UPMEM SDK performs this automatically when using the provided data transfer functions.

[3]Threads can also be stopped using a special `stop` instruction.

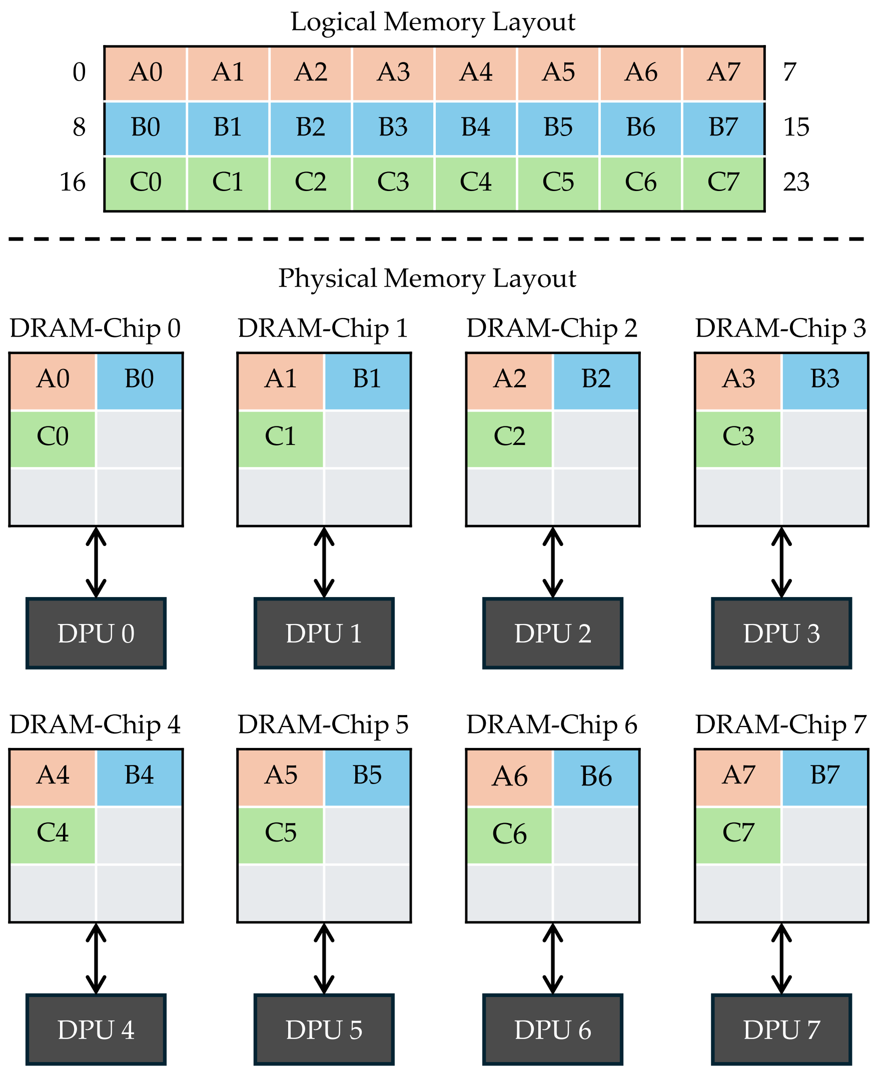


Figure 2.1: Schematic representation of the physical memory layout on memory modules with interleaved DRAM-chips. In this example, the logically consecutive 8-byte word A is physically split over 8 DRAM-chips. Thus, every DPU can only access a small part of A, unless the data is first transposed. The same is true for B and C.

3 Testing the UPMEM System

In this chapter, we familiarise ourselves with the UPMEM system by performing first benchmarks and implementing basic polynomial operations.

3.1 Benchmarking Basic Arithmetic

As a starting point, we conduct benchmarks to determine the time (number of cycles) required to perform basic arithmetic operations (addition and multiplication) on integers of different widths[4]. We test with a fully utilised pipeline (at least 11 active threads). Note that when using a single thread, the number of cycles required for these operations is 11-times higher. The benchmark results are averaged over 16000 executions and the test overhead (e.g. loop counting) is measured separately and subtracted from the results. This allows us to accurately measure the runtime of the tested operations. The results are shown in Figure 3.1. Note that the runtime of 32-bit multiplication depends on the value (significant bit count) of the factors (see Section 2.2.1) and the result shown here is for a worst-case scenario of 32-bit factors. Smaller factors reduce the runtime by one cycle per unset significant bit. For example, 27-bit factors only require 38 cycles.
Most of these results are identical to the number of instructions generated for the operations, meaning that all tested instructions take the same amount of time to execute. This is expected from the DPUs' RISC architecture, which is highly parallel, but whose execution of each thread is relatively simple. The Instruction Set Architecture Manual[5] states that "the apparent latency of any instruction is always one cycle" due to the revolving pipeline design and suspension of threads performing DMA operations, which take multiple cycles. The runtime of non-DMA operations can thus be very accurately predicted by analysing the number of instructions that would be executed. This technique perfectly matches the benchmark results for 8x8-bit, 16x16-bit and 32x32-bit multiplication, with only the 64x64-bit multiplication taking a few cycles longer than expected. This is because of too many accesses to the register file, which the manual calls out as a special circumstance, in which an instruction has to be replayed, requiring one additional cycle. However, this register file access limitation rarely affects normal code and the instruction

[4]This time includes the compiler generated function calls for multiplications larger than 16-bit.
[5]UPMEM Instruction Set Architecture Manual: `https://sdk.upmem.com/2024.1.0/201_IS.html#efficient-scheduling`

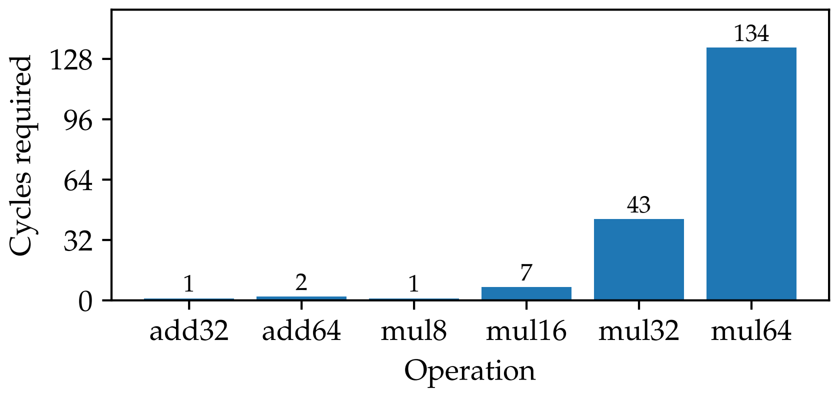


Figure 3.1: DPU benchmark results for basic arithmetic operations on integers of different widths.

count analysis can be very useful for optimising code and quickly comparing different implementations.

3.2 Basic Polynomial Operations

We could not reproduce the results from researches at ETH Zürich [GKG+23], as some test conditions are unclear to us and our own results are either too slow or too fast. We will instead conduct our own benchmarks of basic polynomial operations on DPUs, by starting with a basic implementation and then exploring possible optimisations.

3.2.1 Test Conditions

We test modular additions and modular element-wise multiplications on different numbers of polynomials. These polynomials have 4096 coefficients with 109 bits each, which are typical parameters for the BFV and BGV schemes.

Level of Optimisation

We start with an unoptimised implementation, which uses a naive modular reduction and only loads the data from MRAM which it immediately needs. This means that each thread only buffers the two 128-bit integers (one coefficient per polynomial) which it is currently adding or multiplying. We use the Karatsuba algorithm for the multiplying these 128-bit numbers. We do not use number-theoretic transform (NTT) or residue number system (RNS) techniques.

Multi-Threading

We use coarse-grained multi-threading, in which each thread operates on its own polynomials independently from other threads (as opposed to fine-grained multi-threading, in which multiple threads simultaneously operate on the same polynomials). With this coarse-grained multi-threading, the number of possible concurrent threads depends on the number of polynomials being processed.
For example, with 8 polynomials to be multiplied (4 pairs of 2), only 4 polynomial operations have to be performed and since every thread independently performs such an operation as a whole, only 4 concurrent threads can be utilised. Because DPUs can run up to 11 threads concurrently, they would thus be underutilised when having fewer than 22 polynomials to operate on.

3.2.2 128-Bit and 256-Bit Arithmetic

Since the UPMEM compiler only supports integers up to 64-bit, we have to manually implement the required 128-bit arithmetic. We represent a 128-bit integer as a pair of 64-bit integers — the low part and the high part. In our initial C implementation, addition is performed separately on these parts and the high part is additionally incremented by one, when the low part overflows. Subtraction is implemented similarly. In comparisons, the high parts are compared first and the low parts are compared only if the high parts are equal. The modular reduction after an addition is implemented by comparing the sum with the modulus, which is then conditionally subtracted. For modular subtraction, the minuend and subtrahend are compared, to conditionally add the modulus.
We implement 128-bit multiplication with the Karatsuba algorithm using 32-bit multiplication as the base. This yields a 256-bit result, which we represent as four 64-bit integers. Basic 256-bit operations are implemented similarly to the 128-bit operations described above. However, we also require modular reduction back to a 128-bit result.
We implement a naive modular reduction using a basic division algorithm: First, the divisor is scaled up (left-shifted) until it has the same bit-length as the dividend. Second, if the scaled-up divisor is less-than-or-equal to the dividend, it is subtracted from the dividend. Third, the divisor is right-shifted once and the algorithm repeats from the second step, until the divisor is back at its initial size. The dividend is thus sequentially reduced by $2^l \cdot m$, $2^{l-1} \cdot m, \ldots, 2^2 \cdot m$, $2^1 \cdot m$ and finally $2^0 \cdot m = m$, where m is the divisor/modulus and l is the initial size difference (in bits) between dividend and divisor. Thus, the initial dividend gets reduced modulo m.
Figure 3.2 shows the runtime of these operations compared to operations on native 32-bit and 64-bit integers.

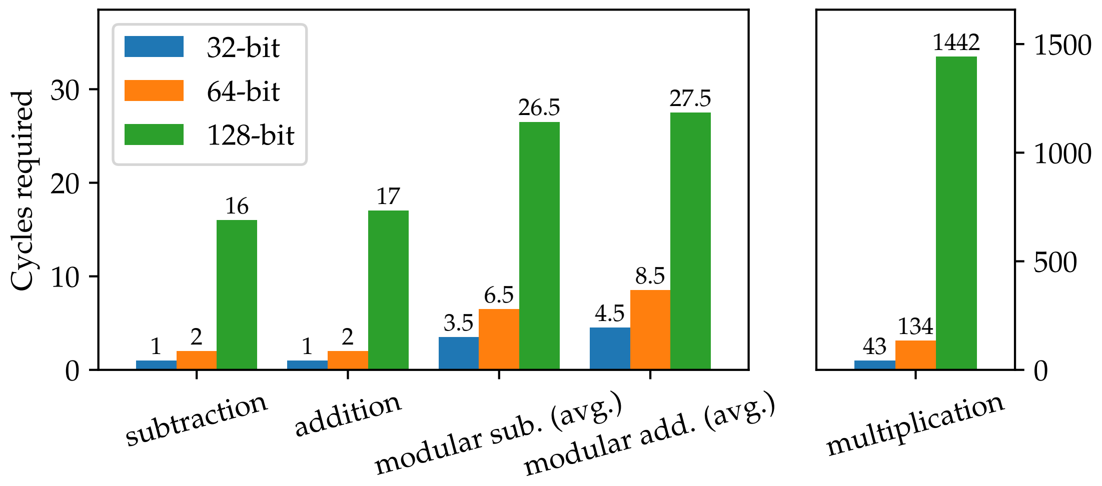


Figure 3.2: Runtimes of 128-bit operations (initial implementation) compared to operations on native 32-bit and 64-bit integers. The modular subtraction and modular addition results are averaged between the case in which a reduction is necessary and the case in which it is not. The runtime of 32-bit multiplication depends on the value of the factors (see Section 3.1).

3.2.3 Initial Results

We test our implementation of modular addition and modular element-wise multiplication on random polynomials with 4096 109-bit coefficients. We use coarse-grained multithreading and each thread only buffers the two coefficients which it is immediately operating on. The tests are run on an UPMEM cloud machine with 2220 DPUs and the polynomials are split as evenly as possible between all DPUs.

We separately measure the following times:

- Time required for transferring the test data to the DPUs.
- Time required for computations on the DPUs (measured both by the host and a cycle counter on the DPUs).
- Time required for retrieving the results from the DPUs.
- Total time required.

We do not measure initialisation steps (like allocating DPUs or generating random test data) and they are not included in the total time measurement.

Modular Polynomial Addition

Figure 3.3 shows the results of testing modular polynomial addition on different numbers of polynomials. Most of the time is spent transferring data, while the actual computations

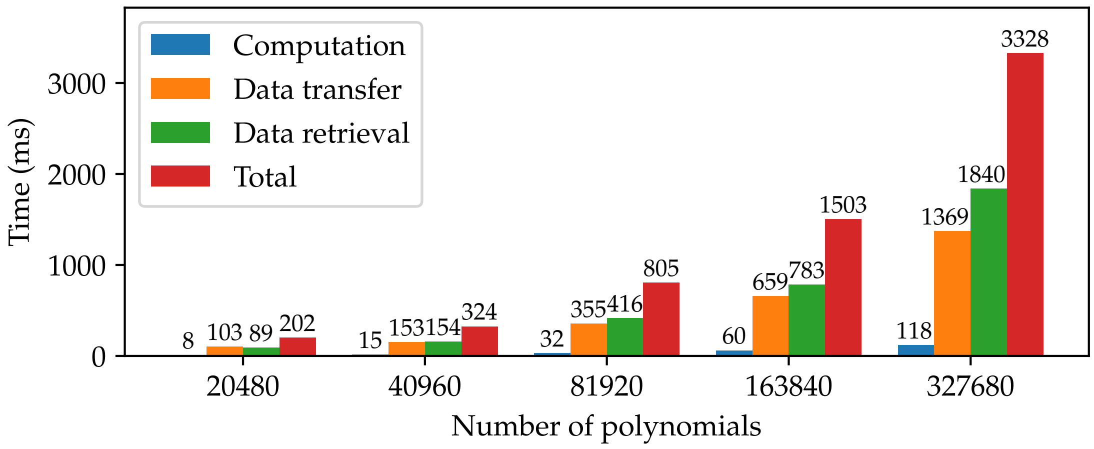


Figure 3.3: DPU benchmark results for modular polynomial addition on different numbers of polynomials with 4096 109-bit coefficients (initial implementation).

only take tens of milliseconds. Note that the data transfer overhead being larger than the computation time is expected for the isolated addition operations in this microbenchmark and does not imply that the UPMEM system is unsuitable for accelerating HE operations. In a real HE use-case, multiple operations would be performed on the data, including more complex operations like multiplication, thus amortising the transfer cost.

Modular Element-Wise Multiplication

Figure 3.4 shows the results of testing modular element-wise multiplication on different numbers of polynomials. Compared to the modular polynomial addition tests, the computation times are much higher and make up the majority of the total time measurements in these tests. We can also see that the computation times are the same for the test cases with up to 40960 polynomials. This is expected due to the coarse-grained threading model and potential underutilisation of DPUs described above. With 2220 DPUs, each DPU has to process the following number of polynomials (rounded):

- For **5120** total polynomials: 2.3
- For **10240** total polynomials: 4.6
- For **20480** total polynomials: 9.2
- For **40960** total polynomials: 18.5
- For **81920** total polynomials: 36.9

We expect DPUs to be underutilised when having fewer than 22 polynomials to operate on and the benchmark results reflect this.

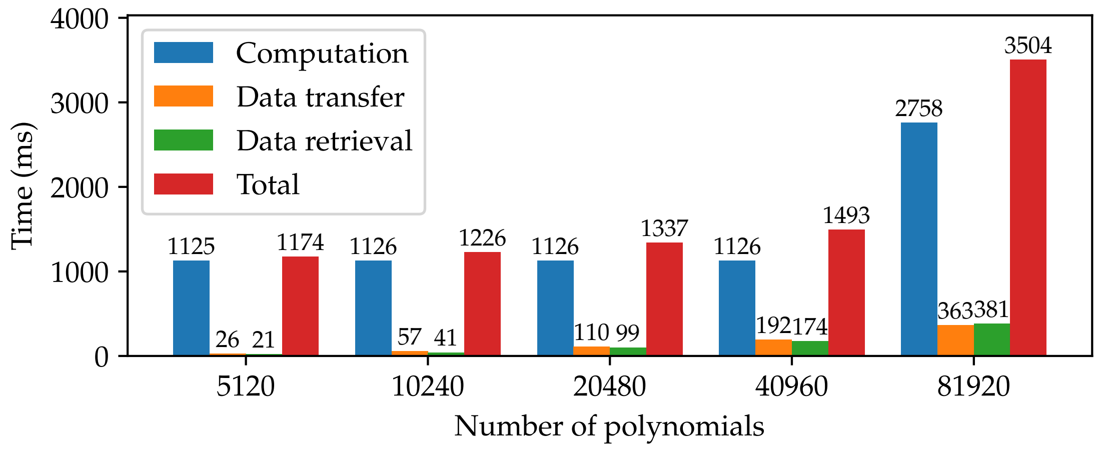


Figure 3.4: DPU benchmark results for modular element-wise multiplication on different numbers of polynomials with 4096 109-bit coefficients (initial implementation).

3.2.4 Optimisations

We optimise the 128-bit arithmetic and polynomial operations. Figure 3.5 visualises the time reduction of these optimisations and Figure 3.6 shows their effect on the modular polynomial addition and modular element-wise multiplication tests.

Buffering MRAM Accesses

We reduce the overhead associated with MRAM accesses by increasing the access size. Instead of always fetching only the current coefficient, we buffer some additional coefficients in WRAM, which we will need later on, thus decreasing the frequency and contention of MRAM accesses.

For our optimised implementation, we give each thread up to 512 bytes of buffer space (larger buffers would overflow the stack). The result is a time reduction between 48% and 74% in the modular polynomial addition tests (depending on the number of polynomials), but only about 0.15% in the modular element-wise multiplication tests. This shows that the modular addition tests are constrained by memory accesses, while the modular multiplication tests are mostly constrained by computational performance.

Addition and Subtraction

Our initial implementation performs 128-bit addition by adding the 64-bit high and low parts separately, checking the low part for overflow (the sum being less than one of the inputs) and then conditionally incrementing the high part by one. Subtraction is implemented similarly. However, this takes up to three operations (addition, comparison, incrementing), when a single carry-aware operation could achieve the same result. This is because C does not provide a standard way of accessing the carry-out of an operation and inputting it into the next one.

We have thus created an optimised implementation, which uses the DPUs' `addc` and `subc` instructions via inline assembly, reducing the time required for addition and subtraction to just 10 cycles each. This corresponds to a 41.2% time reduction for addition, a 37.5% time reduction for subtraction and a 27.3% / 24.5% time reduction for modular addition and modular subtraction respectively. However, this only translates to an additional time reduction between 5.6% and 1.2% in the modular polynomial addition tests, as they are still constrained by memory accesses.

These optimisations also improve the runtime of modular multiplication, because its division-based modular reduction step benefits from faster base operations (addition and subtraction). The time reduction for modular multiplication is about 10%, which also translates into a ~10% time reduction in the modular element-wise multiplication tests, as they are constrained by computational performance.

Coalescing Comparison and Subtraction

We have previously described a simple modular subtraction, which compares the minuend and subtrahend, to then conditionally add the modulus to their difference. This operation can be optimised by coalescing the comparison and the subtraction. We can achieve this by checking the processor flags (e.g. the carry or zero flag) after the subtraction. This is also how comparisons are usually implemented in hardware, so our initial implementation essentially performs two subtractions (one comparison and one "normal" subtraction), when one could suffice. Our optimised implementation, which coalesces comparison and subtraction using inline assembly, achieves an additional 25% time reduction for modular subtraction. While the modular polynomial addition and modular element-wise multiplication tests do not utilise this operation, other operations like modular polynomial subtraction or NTT (see next chapter) can benefit from this type of optimisation.

Summary and Further Optimisations

With these relatively simple optimisations, we have achieved a total time reduction of ~70% in the modular polynomial addition tests and ~10% in the modular element-wise multiplication tests. Notice that the modular polynomial addition mostly benefits from the optimised buffering, while the modular element-wise multiplication mostly benefits from the computational improvements, which is in-line with our previous observations.

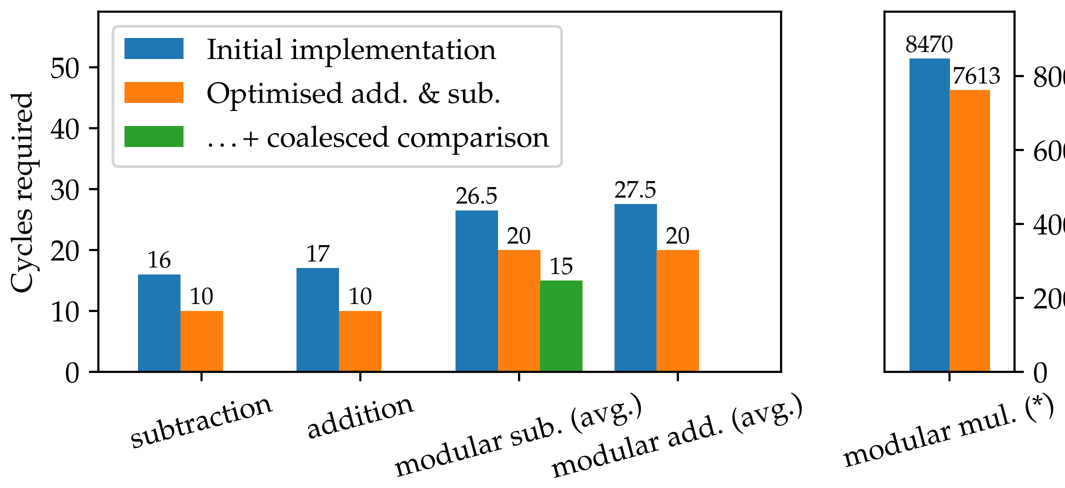
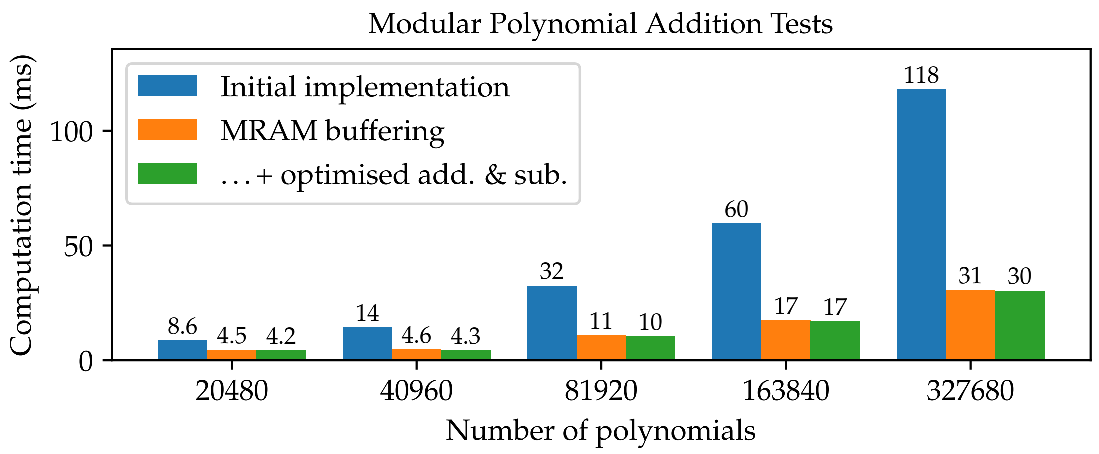


Figure 3.5: Performance comparison between initial and optimised DPU implementations of operations on 128-bit integers. The modular subtraction and modular addition results are averaged between the case in which a reduction is necessary and the case in which it is not. Because the runtime of the modular multiplication (*) is dependent on its operands, our comparison uses a fixed set of inputs. However, different inputs might produce different results for this operation.

However, modular element-wise multiplication remains slow and modular polynomial multiplication takes over an hour to complete (not shown here). There are some known methods which we could apply to optimise these 128-bit modular multiplications. For example, we could implement Barrett reduction instead of our naive modular reduction based on division. We could also apply number-theoretic transform (NTT) techniques to drastically reduce the runtime of modular polynomial multiplication, as NTT allows computing this operation in $\mathcal{O}(n \log n)$ steps, instead of $\mathcal{O}(n^2)$.

However, even with these optimisations, we would still be computing on 128-bit and 256-bit values, which as we have seen, is much slower than 32-bit or 64-bit arithmetic on DPUs (see Figure 3.2 and Figure 3.5). To improve performance on these large coefficients, we can instead represent them using a residue number system (RNS). This decomposes a number into *residues*, which are the remainders when dividing by a set of moduli. If we choose moduli which are 32-bit or smaller, we can then compute on values of the DPUs native word size, which is faster.

In the following chapter, we will use this technique to improve performance and will thus focus on 32-bit and 64-bit arithmetic. We will also apply the aforementioned optimisations like NTT on these smaller word sizes, instead of implementing them for large coefficients directly.



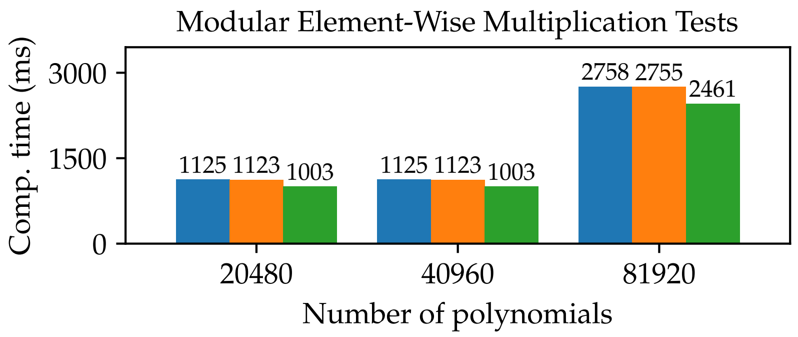


Figure 3.6: Performance comparison between initial and optimised DPU implementations of modular operations on polynomials with 4096 109-bit coefficients. For modular element-wise multiplication, not all test cases are shown, because the test cases up to 40960 polynomials have identical results.

4 Accelerating FHE

We implement the number-theoretic transform (NTT) and leverage residue number systems (RNS) to accelerate polynomial operations on DPUs. We test and evaluate our implementation and compare it to Microsoft SEAL's [SEA23] optimised CPU implementation regarding both runtime and energy efficiency.

4.1 Number-Theoretic Transform (NTT)

The NTT is a generalisation of the discrete Fourier transform (DFT) to a finite field, like integers modulo a prime q. Fourier transforms take in vectors of numbers (e.g. polynomial coefficients) and output same size vectors, which can be thought of as evaluations of the input polynomial at specific points, called roots of unity. While roots of unity are easy to compute for DFTs, which operate on complex numbers, they are harder to find in finite fields and are often pre-computed or hard-coded for NTT. These Fourier transforms of polynomials are useful, because polynomial multiplications (and additions) can then be performed pointwise, i.e. in linear time. Since NTT, as well as its inverse (iNTT), can be applied in $\mathcal{O}(n \log n)$ time, for example using the Cooley-Tukey algorithm, the total time required for polynomial multiplication using NTT is also in $\mathcal{O}(n \log n)$. By comparison, a naive implementation would take $\mathcal{O}(n^2)$ time and even optimised implementations like the Karatsuba algorithm with $\mathcal{O}(n^{\log_2 3})$ time complexity are asymptotically slower than NTT-based multiplication. Performance can be improved further by storing the polynomials in their NTT-form and performing multiple operations on them, before inverting the transform only as the last step of a computation. However, some operations require the coefficient form (i.e. the non-transformed representation), which can make it necessary to perform multiple transformations as part of a computation.

4.1.1 Concepts Required for NTT

We define primitive roots of unity, describe working moduli and briefly explain the difference between positive-wrapped and negative-wrapped convolutions relevant to NTT.

Primitive n-th Root of Unity

Definition 4.1 (Primitive n-th root of unity). Let $\mathbb{Z}_q$ be an integer ring modulo q and let $n \in \mathbb{N}_+$ be a positive integer. Then $\omega \in \mathbb{Z}_q$ is a primitive n-th root of unity in $\mathbb{Z}_q$ if and only if

$$\omega^n \equiv 1 \mod q$$

and

$$\omega^k \not\equiv 1 \mod q$$

for all $k < n$, $k \in \mathbb{N}_+$.

For a primitive n-th root of unity $\omega \in \mathbb{Z}_q$, it follows that $\omega^{kn+i} = (\omega^n)^k \omega^i \equiv \omega^i \mod q$ for all $k, i \in \mathbb{N}_0$. And for a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$ (i.e. $\psi^{2n} = (\psi^n)^2 \equiv 1 \mod q$), it additionally follows that $\psi^n \equiv -1 \mod q$.

Working Modulus

For our purposes, NTT operates on integers modulo a prime q. We require the resulting ring $\mathbb{Z}_q$ to contain specific primitive roots of unity, dependent on the length of the input. Additionally, our prime q should be larger than all input values, so that they can be unambiguously represented in $\mathbb{Z}_q$. If operations on the transformed data must not overflow (i.e. be reduced modulo q), then q must also be larger than all expected results.
To meet these requirements, q should be a prime larger than all inputs (and expected results), which satisfies $q = 2nk + 1$ for some positive integer $k \in \mathbb{N}_+$, where n is the length of the input. Because q is prime, the multiplicative group $\mathbb{Z}_q$ must have a generator and Euler's theorem guarantees that it also has a primitive $2n$-th root of unity. A modulus q which meets these requirements is known as a *working modulus*. It is one of the parameters of an NTT.

Positive- and Negative-Wrapped Convolutions

Mathematically, the multiplication of two polynomials can also be described as a convolution of their coefficient vectors. But what if the resulting vector is too large, for example because we are working in $\mathbb{Z}_q[x]/(x^4+1)$ for some q, but the resulting polynomial contains x^4 and x^5 terms?
In this case, we differentiate between *positive-wrapped* and *negative-wrapped* convolutions. Both can be computed with NTT, but they require different implementations. A positive-wrapped (or *cyclic*) convolution can be thought of as operating modulo $x^n - 1$ (or in this case $x^4 - 1$), while a negative-wrapped (or *negacyclic*) convolution operates modulo $x^n + 1$.

4.1.2 NTT Definitions

Note that in the following definitions, all operations are performed in their respective integer ring $\mathbb{Z}_q$, for example n^{-1} refers to the inverse of n in $\mathbb{Z}_q$.

Positive-Wrapped Transform

Definition 4.2 (NTT$^+$). Let $c \in \mathbb{Z}_q^n$ be a vector (of coefficients) with length n and let ω be a primitive n-th root of unity in $\mathbb{Z}_q$. The positive-wrapped number-theoretic transform (NTT$^+$) of c with parameters (q, ω) is a vector $t \in \mathbb{Z}_q^n$, where

$$t_i = \sum_{j=0}^{n-1} \omega^{ij} c_j \mod q$$

for $i = 0, 1, \ldots, n-1$.

Definition 4.3 (iNTT$^+$). Let $t \in \mathbb{Z}_q^n$ be a vector with length n and let ω be a primitive n-th root of unity in $\mathbb{Z}_q$. The inverse positive-wrapped number-theoretic transform (iNTT$^+$) of t with parameters (q, ω) is a vector $c \in \mathbb{Z}_q^n$, where

$$c_i = n^{-1} \sum_{j=0}^{n-1} \omega^{-ij} t_j \mod q$$

for $i = 0, 1, \ldots, n-1$.

Note that the inverse transform of a transform with the same parameters (q, ω) is always the original input, i.e. $\text{iNTT}^+(\text{NTT}^+(c)) = c$ for all $c \in \mathbb{Z}_q^n$. The same holds for the following negative-wrapped transform (NTT$^-$) and its inverse (iNTT$^-$) if they use the same parameters (q, ψ).

Negative-Wrapped Transform

Definition 4.4 (NTT$^-$). Let $c \in \mathbb{Z}_q^n$ be a vector (of coefficients) with length n and let ψ be a primitive $2n$-th root of unity in $\mathbb{Z}_q$. The negative-wrapped number-theoretic transform (NTT$^-$ or just NTT) of c with parameters (q, ψ) is a vector $t \in \mathbb{Z}_q^n$, where

$$t_i = \sum_{j=0}^{n-1} \psi^{2ij+j} c_j \mod q$$

for $i = 0, 1, \ldots, n-1$.

Definition 4.5 (iNTT$^-$). Let $t \in \mathbb{Z}_q^n$ be a vector with length n and let ψ be a primitive $2n$-th root of unity in $\mathbb{Z}_q$. The inverse negative-wrapped number-theoretic transform (iNTT$^-$ or just iNTT) of t with parameters (q, ψ) is a vector $c \in \mathbb{Z}_q^n$, where

$$c_i = n^{-1} \sum_{j=0}^{n-1} \psi^{-2ij-i} t_j \mod q$$

for $i = 0, 1, \ldots, n-1$.

Since HE usually uses the negative-wrapped transform (NTT$^-$ and iNTT$^-$), we will also refer to it simply as NTT and iNTT.

4.1.3 Example

Consider two polynomials A and B from the polynomial ring $\mathbb{Z}_{41}[x]/(x^4+1)$, which we want to multiply:

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 + 4x^3 \\ B(x) &= 3 + 1x + 4x^2 + 1x^3 \end{aligned}$$

Naively, we could calculate:

$$\begin{aligned} A(x) \cdot B(x) &= 1(3 + 1x + 4x^2 + 1x^3) + 2x(3 + 1x + 4x^2 + 1x^3) \\ &+ 3x^2(3 + 1x + 4x^2 + 1x^3) + 4x^3(3 + 1x + 4x^2 + 1x^3) \\ &= 3 + 7x + 15x^2 + 24x^3 + 18x^4 + 19x^5 + 4x^6 \\ &\equiv -15 + -12x + 11x^2 + 24x^3 \mod x^4 + 1 \\ &\equiv 26 + 29x + 11x^2 + 24x^3 \mod 41 \end{aligned}$$

But this has quadratic complexity, because every coefficient of A must be multiplied with every coefficient of B. If we instead perform an NTT on A and B with parameters $q = 41$ and $\psi = 3$:

$$\begin{aligned} T_A &= \text{NTT}^-(A) = (19, 40, 37, 31) \\ T_B &= \text{NTT}^-(B) = (28, 38, 9, 19) \end{aligned}$$

These vectors can then be multiplied pointwise:

$$T_{AB} = T_A \circ T_B \mod q = (532, 1520, 333, 589) \mod q = (40, 3, 5, 15)$$

And we can finally perform an inverse NTT:

$$AB = \text{iNTT}^{-}(T_{AB}) = (26, 29, 11, 24)$$

To get the correct result. Of course the NTT itself still has quadratic complexity if implemented naively based on the definition, but fast NTT implementations like the Cooley-Tukey algorithm can perform an NTT or inverse NTT in $\mathcal{O}(n \log n)$ time. The full multiplication then takes three $\mathcal{O}(n \log n)$ transforms and one $\mathcal{O}(n)$ pointwise multiplication for a total time complexity of $\mathcal{O}(n \log n)$.

4.1.4 Fast NTT Algorithms

NTT is a generalisation of the DFT and many fast DFT algorithms can also be generalised to work with it. This includes the fast Fourier transform by Cooley-Tukey [CT65] and many of its variations.

In the following, we will show how these can be applied to NTT. We will focus on the radix-2 case, which recursively splits a length n NTT into two smaller NTTs of length $n/2$. The base case is $n = 1$, since the NTT of a single element is just the identity function. We will also assume that n is always a power of two, as is typical for HE.

Similarly, an iNTT of length n can also be decomposed into two smaller iNTTs of length $n/2$, which we will not show here.

Cooley-Tukey NTT

Starting with Definition 4.4 (NTT^{-}):

$$t_i = \sum_{j=0}^{n-1} \psi^{2ij+j} c_j \mod q$$

We can split the sum into even indices ($2j$) and odd indices ($2j+1$):

$$\begin{aligned} t_i = &\sum_{j=0}^{n/2-1} \psi^{2i(2j)+2j} c_{2j} \\ &+ \sum_{j=0}^{n/2-1} \psi^{2i(2j+1)+2j+1} c_{2j+1} \mod q \end{aligned}$$

If we carry out the multiplications, we can extract the common factor ψ^{2i+1} from the second sum:

$$t_i = \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j} + \sum_{j=0}^{n/2-1} \psi^{4ij+2j+(2i+1)} c_{2j+1} \mod q \tag{4.1}$$

$$= \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j} + \psi^{2i+1} \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j+1} \mod q \tag{4.2}$$

With the common factor extracted, we can see that the two sums are almost identical, except that the first sum operates on the even inputs (c_{2j}) and the second sum operates on the odd inputs (c_{2j+1}). We can also see that the sums match the NTT definition for length $n/2$ and a primitive n-th root of unity $\psi' = \psi^2$, which suggests that we can decompose a length n NTT into two smaller NTTs of length $n/2$. The master theorem tells us that this would achieve a runtime of $\mathcal{O}(n \log n)$.

However, as the equation currently stands, this does not work, because these smaller NTTs only produce $n/2$ values, but we need results for $i = 0, 1, \ldots, n-1$. Thus, we still need a way to compute the upper $n/2$ values. Replacing i with $i + n/2$ yields:

$$t_{i+n/2} = \sum_{j=0}^{n/2-1} \psi^{4(i+n/2)j+2j} c_{2j} + \psi^{2(i+n/2)+1} \sum_{j=0}^{n/2-1} \psi^{4(i+n/2)j+2j} c_{2j+1} \mod q$$

$$= \sum_{j=0}^{n/2-1} \psi^{2nj} \psi^{4ij+2j} c_{2j} + \psi^{n} \psi^{2i+1} \sum_{j=0}^{n/2-1} \psi^{2nj} \psi^{4ij+2j} c_{2j+1} \mod q$$

Notice that we were able to extract the "$+n/2$"-part into the factors ψ^{2nj} and ψ^n.

Recalling that $\psi^{2nj} \equiv 1 \mod q$ and $\psi^n \equiv -1 \mod q$ (as ψ is a primitive $2n$-th root of unity in $\mathbb{Z}_q$), we can simplify as follows:

$$t_{i+n/2} = \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j} - \psi^{2i+1} \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j+1} \mod q \tag{4.3}$$

Which produces an equation for the upper $n/2$ results, which is very similar to Equation 4.2 and depends on the same smaller NTTs. The only difference is that the second sum gets subtracted instead of added.

If we denote the sum over the even coefficients as $E_i = \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j}$ and the sum over the odd coefficients as $O_i = \sum_{j=0}^{n/2-1} \psi^{4ij+2j} c_{2j+1}$, we can combine Equation 4.2 and

Equation 4.3, to compute t_i as:

$$\begin{aligned} t_i &= E_i + \psi^{2i+1} O_i \mod q \\ t_{i+n/2} &= E_i - \psi^{2i+1} O_i \mod q \end{aligned} \quad \text{for } i = 0, 1, \ldots, n/2 - 1$$

or equivalently:

$$t_i = \begin{cases} E_i + \psi^{2i+1} O_i \mod q & \text{if } i < n/2 \\ E_{i-n/2} - \psi^{2i-n+1} O_{i-n/2} \mod q & \text{if } i \geq n/2 \end{cases} \quad \text{for } i = 0, 1, \ldots, n-1$$

where E_i and O_i can be obtained by two smaller NTTs of length $n/2$.
This computation is often referred to as a *butterfly* and visualised as follows:

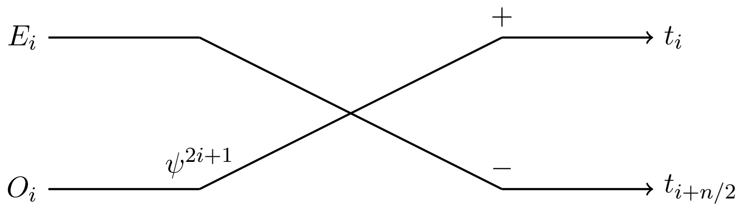


Figure 4.1: Visualisation of a butterfly operation in a basic Cooley-Tukey NTT. The intermediate result O_i is multiplied by ψ^{2i+1} and then added to / subtracted from the intermediate result E_i to produce the respective outputs t_i and $t_{i+n/2}$.

Iterative Operation

Above, the Cooley-Tukey NTT is described recursively. However, it can also be implemented iteratively, which is often preferred. In this case, the (sub-)NTTs are grouped by their length (which corresponds to their recursive depth) and computed group by group. For example, all length 2 NTTs are computed first, then all length 4 NTTs, all length 8 NTTs, etc. This results in $\log_2(n)$ passes over the data, which are also known as *stages*.

In-Place Operation and Bit Reversal

The basic idea described above can be directly implemented using $\mathcal{O}(n)$ auxiliary storage, but an in-place implementation is slightly harder. To illustrate the problem, consider a fast NTT, as first described, with length $n = 32$. For in-place operation, the inputs and outputs overlap, so input E_0 (the first even element) is in the same place as output t_0, input O_0 (the first odd element) is in the same place as output t_1, etc. We can fetch E_0 and O_0 to compute $t_0 = E_0 + \psi^1 O_0 \mod q$ and $t_{16} = E_0 - \psi^1 O_0 \mod q$. Storing output t_0 overwrites input E_0, which is unproblematic, because E_0 has already been used. However, storing output t_{16}

overwrites input E_8 before it was read, which will produce incorrect results. In general, performing a butterfly on inputs E_i and O_i, which are continuous in memory, produces results t_i and $t_{i+n/2}$, which are in opposite halves of the output. The same problem occurs during the computation of E_i, O_i and their recursively smaller NTTs.

One solution is to always store the butterfly results back into the input positions. This changes each NTT's output, but in a predictable way, which is equivalent to rotating the index bits of the output elements by one. This is because elements which were supposed to be in opposite halves of the output (i.e. differ in the most significant bit of their index), are next to each other instead (i.e. differ in the least significant bit of their index).

Over all $\log_2(n)$ stages, the result is a full bit reversal of the elements' positions. This can be offset by explicitly performing another bit reversal, to restore the normal element order. However, for some operations, like NTT-based multiplication (convolution) as in HE, the order of elements is not actually important. Thus, as an additional optimisation, we can skip this second bit reversal and instead directly operate on the data in bit-reversed order.

Twiddle Factors

The butterfly shown in Figure 4.1 operates on intermediate results and powers of the root of unity ψ. These root-of-unity-powers can be seen as an auxiliary input to the algorithm. They are often pre-computed and additionally modified for the specific implementation (e.g. stored in bit-reversed order or pre-multiplied by a scaling factor). These pre-computed, modified powers of the root of unity ψ are called *twiddle factors*.

4.2 Residue Number System (RNS)

In computing, numbers are usually represented in a weighted binary system, with larger numbers requiring more bits. Such a system has a single base (the number 2) and the weight of each digit (bit) is determined by its position, i.e. the digits have weight 2^0, 2^1, 2^2 and so forth. The weighted binary system has a lot of advantages, but can become inefficient when multiplying large numbers. Naively, multiplying two numbers of length n requires $\mathcal{O}(n^2)$ operations. Karatsuba multiplication can reduce this to $\mathcal{O}(n^{\log_2 3})$ operations, but this is still exponential with respect to n. However, if we represent numbers in a residue number system (RNS) instead, modular multiplication and addition of large numbers only requires linear time. But this representation also has downsides. For example, general division is more complicated in an RNS representation, than in a weighted binary representation.

4.2.1 Representation

RNS uses a set of bases (or *moduli*) $m = (m_1, m_2, \ldots m_n)$ which are pairwise coprime. A number is then represented by its *residues* modulo these bases (the remainders when dividing by these moduli). For example, using RNS with bases $m = (3, 5, 7)$, the number $x = 22$ would be represented as:

$$x = (22 \bmod 3, 22 \bmod 5, 22 \bmod 7) = (1, 2, 1)$$

The set of bases m is also called the (RNS) base. The product of these bases is known as $M = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ and an RNS can only (uniquely) represent integers from 0 to $M - 1$.

4.2.2 Modular Addition and Multiplication

In RNS representation, two numbers can be added by simply adding their corresponding residues modulo their base. For example, with bases $m = (3, 5, 7)$, we can add $x = 22 = (1, 2, 1)$ and $y = 79 = (1, 4, 2)$ modulo $M = 3 \cdot 5 \cdot 7 = 105$ as follows:

$$x + y = (1, 2, 1) + (1, 4, 2) = (1 + 1 \bmod 3, 2 + 4 \bmod 5, 1 + 2 \bmod 7) = (2, 1, 3)$$

The Chinese Remainder Theorem allows us to compute that $(2, 1, 3)$ represents 101, which is the correct sum of x and y.
If we instead add x and $z = 100 = (1, 0, 2)$, we get:

$$x + z = (1, 2, 1) + (1, 0, 2) = (1 + 1 \bmod 3, 2 + 0 \bmod 5, 1 + 2 \bmod 7) = (2, 2, 3)$$

which represents 17, because $x + z = 22 + 100 = 122 = 17 \mod 105$.
Modular multiplication works similarly: To multiply two numbers in RNS representation, we multiply their corresponding residues modulo their base. For example, with the previous bases $m = (3, 5, 7)$, we can multiply $a = 6 = (0, 1, 6)$ and $b = 9 = (0, 4, 2)$ modulo $M = 3 \cdot 5 \cdot 7 = 105$ as follows:

$$a \cdot b = (0, 1, 6) \cdot (0, 4, 2) = (0 \cdot 0 \bmod 3, 1 \cdot 4 \bmod 5, 6 \cdot 2 \bmod 7) = (0, 4, 5)$$

which correctly represents $54 = 6 \cdot 9$, since $54 \bmod 3 = 0$, $54 \bmod 5 = 4$ and $54 \bmod 7 = 5$.

4.2.3 Choice of Moduli

RNS only requires the set of moduli to be pairwise coprime. However, since we want to combine RNS and NTT, we must choose moduli that are compatible with NTT. This means that moduli must be prime and contain suitable primitive roots of unity. In other words, each RNS modulus must also be a working modulus for an NTT of the polynomial length. Fortunately, we can pre-compute such RNS bases as part of the encryption parameters.

4.2.4 Converting to and from RNS

Converting between RNS and a normal weighted binary system is computationally expensive, which can negate the speed-up that we want to achieve. However, this is rarely a problem in our use-case, since we can typically specify the encryption parameters to be used. If the secret-holding party (i.e. the client) knows that we expect a certain RNS representation, it can perform the key generation, encryption and decryption in this RNS form as well, thus skipping the conversion step or combining it with other expensive operations. This is a common technique and is also how encryption is handled in Microsoft SEAL, which we will compare against later.

4.3 Our Implementation

In this section, we describe our improved implementation, which combines the RNS and NTT techniques described above.

4.3.1 RNS Representation of Polynomials

We want to use RNS representations to improve arithmetic performance, especially multiplication. Consider that our data has the following logical structure:

- A ciphertext consists of at least two polynomials.
- Each polynomial has n coefficients, where n is a power of two, typically between 1024 and 32768.
- These coefficients may have hundreds of bits.

We can use an RNS representation for the polynomials constituting a ciphertext, but there are multiple possibilities for storing the polynomials in this form. In all RNS representations, coefficients are split into multiple residues. The difference is in where these residues are stored in memory:

1. The residues of each coefficient could be stored next to each other and kept in a single polynomial (which is just an array). This results in an **array of tuples of residues** layout.
2. The residues of each coefficient could be separated and stored in different arrays. This results in a **tuple of arrays of residues** layout, where all residues in an array correspond to the same modulus.

Our implementation uses option two, i.e. a **tuple of arrays of residues** layout. We also refer to these arrays of residues of the same modulus as *sub-polynomials*, so in our implementation, a polynomial in RNS representation is a **tuple of sub-polynomials**. This has the following advantages:

- Separating the residues allows for more flexibility in distributing the data between DPUs or DPU threads.
- It simplifies the iteration over residues of the same modulus, because this is an iteration over continuous memory. Option one would require strided access in this case, which is less efficient for buffering.

4.3.2 Splitting Polynomials Between DPUs

In our implementation, each DPU operates on sub-polynomials of a specific modulus, which can be set by the host. As an example, consider two ciphertexts ct and ct' with three primes m_1, m_2, m_3 as RNS moduli. In RNS representation, each polynomial of these ciphertexts consists of three sub-polynomials:

$$\begin{aligned} ct &= (ct_0, ct_1) = ((ct_{0m_1}, ct_{0m_2}, ct_{0m_3}), (ct_{1m_1}, ct_{1m_2}, ct_{1m_3})) \\ ct' &= (ct'_0, ct'_1) = ((ct'_{0m_1}, ct'_{0m_2}, ct'_{0m_3}), (ct'_{1m_1}, ct'_{1m_2}, ct'_{1m_3})) \end{aligned}$$

To add these ciphertexts, one DPU would be configured for modulus m_1 and operate on sub-polynomials ct_{0m_1}, ct_{1m_1}, ct'_{0m_1} and ct'_{1m_1}. Another DPU would be configured for modulus m_2 and operate on sub-polynomials ct_{0m_2}, ct_{1m_2}, ct'_{0m_2} and ct'_{1m_2}. And a third DPU would operate on the remaining sub-polynomials for modulus m_3.
The advantage of this is that each DPU only has to operate on a single modulus and also only needs the support data (like NTT twiddle factors) for this one modulus, which reduces the memory and transfer requirements for the DPUs. Of course, multiple DPUs can also be configured for the same modulus to improve throughput if required.

4.3.3 CPU-DPU Interface

As part of a DPU program, the UPMEM SDK allows defining *symbols*, which denote memory regions in MRAM or WRAM for transferring data between DPUs and the CPU. These symbols are compiled into the DPU binary and the position and size of their regions are static.
This leaves some options for designing the interface between DPUs and the host CPU. We could define a separate symbol for every chunk of data that we need in the interface. For example, different symbols for `input_polynomials`, `output_polynomials`, `commands`, `twiddle_factors`, `modulus_data`, etc.

However, this approach has a few problems: The optimal interface layout depends on the encryption parameters and other dynamic factors. For example, the number of twiddle factors (and thus memory needed for the `twiddle_factors` symbol) depends on the length of the polynomials, and the optimal memory split between `input_polynomials`, `output_polynomials` and `commands` depends on the operations to be performed. Thus, for an optimal interface layout, the DPU program would need to be recompiled whenever one of these parameters change. Additionally, the UPMEM SDK requires the host to perform a separate data transfer for every symbol it wants to access, which could limit performance.
Instead, we use a single MRAM symbol for our interface. The memory at this symbol begins with a header, which we call the *auxiliary data*, followed by the *main data* — an array of DPU words (32-bit integers) which spans the rest of the DPU's MRAM. The header contains information about the rest of the data, like the polynomial length, the modulus and the number of commands. It also contains some pre-computed data, like the modular inverse of the polynomial length. Lastly, it contains offsets into the *main data*, which define where different data sections start, like the twiddle factors, the commands or the sub-polynomials to operate on. This design is very flexible, because it allows the host to dynamically define the positions and sizes of the various data sections. It also allows the host to update multiple of these sections in a single data transfer.

Command Structure

In our implementation, DPUs receive a list of simple commands, which they execute sequentially. But these simple commands can be composed into more complex operations. A command consists of four 16-bit integers: The command type and three arguments `out`, `A` and `B`. The command type is an enumeration of commands like `ADD`, `MUL_POINTWISE`, `FWD_NTT` or `STOP`. The arguments `out`, `A` and `B` specify indices of the input and output polynomials in *main data*. Some commands operate in-place on the `out` polynomial and ignore the `A` and `B` arguments. Special *batching commands* like `INV_NTT_BATCH` take a range of polynomials instead and perform the given operation on all of them. This improves performance for operations using coarse-grained multi-threading (see below).

4.3.4 Multi-Threading

We run 16 threads per DPU. This is to ensure that the DPU's pipeline is always saturated, even when some threads are blocked on DMA operations. This number also simplifies splitting coefficients of a sub-polynomial between threads, because the number of threads and the polynomial length are both a power of two. To ensure data consistency, threads

are synchronised between commands.
We differentiate between fine-grained and coarse-grained multi-threading. In fine-grained multi-threading, multiple threads are simultaneously operating on the same sub-polynomials. This type of multi-threading can improve performance, even when operating on few sub-polynomials. In coarse-grained multi-threading, each thread operates on its own sub-polynomials independently from other threads. For the maximum performance improvement, this type of multi-threading requires operating on many sub-polynomials concurrently.
We use fine-grained multi-threading for element-wise operations, like modular addition or multiplication. These operations are easily parallelisable and require no synchronisation between threads. For more complicated operations, like NTT and iNTT, we use coarse-grained multi-threading instead. This avoids synchronisation overhead, and while it can limit performance when operating on only a few sub-polynomials, in a typical use-case, all incoming data and all results are transformed via NTT and iNTT respectively, which means that situations with few sub-polynomials should be rare.

4.3.5 Improvement over Previous Implementation

The initial version of this improved implementation already outperforms our previous 128-bit implementation. Recall that our previous implementation required ~1000 ms (computation time only) to perform the modular element-wise multiplication test on 40960 polynomials with 4096 109-bit coefficients each. Our improved implementation only requires 353 ms for the equivalent task of performing modular element-wise multiplication on $4 \cdot 40960 = 163840$ sub-polynomials, each with 4096 27-bit coefficients. To perform modular polynomial multiplication (which previously took over an hour), these sub-polynomials can also be transformed using NTT, then multiplied pointwise and finally transformed back using iNTT, which only takes about 7.9 seconds for the above example of 40960 polynomials.
Our improved implementation is also more flexible, since it can easily adapt to different coefficient sizes by using a larger RNS base, i.e. increasing the number of sub-polynomials. For example, to use 150-bit coefficients, our improved implementation can use five moduli with 30 bits each. In contrast, our previous implementation was unable to handle coefficients larger than 128 bits without manual changes. Additionally, since our improved implementation splits each polynomial into multiple sub-polynomials, operations using coarse-grained multi-threading can be parallelised more, even without changing their implementations.
Since we have already significantly improved upon our previous implementation, in the following section, we will compare our improved implementation to Microsoft SEAL in-

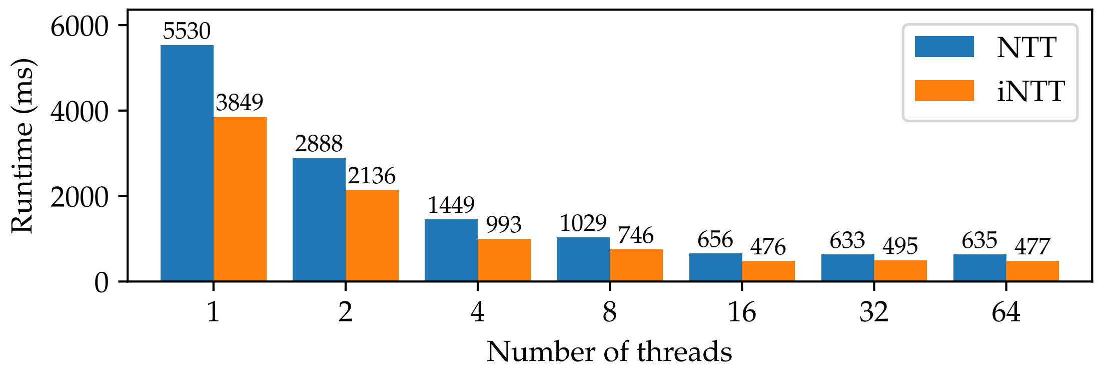


Figure 4.2: Runtime of SEAL for NTT and iNTT using different numbers of threads, tested on 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients (65536 polynomials in total).

stead, which is a popular and optimised HE library.

4.4 Performance Evaluation

We analyse the performance of our implementation and compare it to Microsoft SEAL [SEA23] regarding both runtime and energy efficiency. We test three operations, which are typical for FHE schemes, namely NTT, iNTT and BGV multiplication, which consists of four modular polynomial multiplications and one modular polynomial addition. All of these operations are performed on many ciphertexts and repeated multiple times, to get more accurate results. We first test the basic implementation (as described in the previous section) and then explore additional optimisations.

For BGV multiplication, we assume that the ciphertexts are stored in NTT form (for comparability with Microsoft SEAL). These BGV results can also be added with the NTT (and iNTT) results, to get good approximations of alternate BGV multiplications which include these steps. For PIM, we only consider the actual computation time (as measured by the DPUs' cycle counters) and do not include the time required for transferring data between the CPU and the DPUs. These computation times can be composed together and if required, the transfer times can simply be added. In the following tests with 32768 ciphertexts, transferring them to the DPUs takes about 178 ms. As we have seen previously, transfer times scale linearly with the amount of data and are mostly dominated by the DPU computation times for these complex operations.

4.4.1 Comparing to Microsoft SEAL

Microsoft SEAL [SEA23] is a popular open-source HE library. SEAL supports the BFV, BGV and CKKS[6] HE-schemes and also uses optimisations like RNS and NTT. We use SEAL as an optimised CPU implementation to compare against.
For the following benchmarks, we create the test data using SEAL, by generating ciphertexts which encrypt increasing values. When comparing polynomial operations (like NTT) on some number of polynomials p, we only generate $p/2$ ciphertexts, since freshly encrypted ciphertexts consist of two polynomials each.

Ciphertext Moduli

Because SEAL uses *modulus switching* for both BGV and BFV, the ciphertext moduli are a bit smaller than in the previous chapter. For example, with a polynomial length of 4096, SEAL uses a key level of 109 bits[7] with an RNS base of three moduli (36, 36 and 37 bits), but ciphertexts only use the first two moduli and thus have 72-bit coefficients ($2 \cdot 36$-bit).
For the comparisons, our DPU implementation uses ciphertext moduli of the same size as SEAL, but with different, DPU-compatible RNS bases (moduli of at most 32-bit each). This makes the computations directly comparable, but can be a suboptimal choice for the DPU implementation (see Section 4.4.5).

SEAL Measurements

We use SEAL version 4.1.2 compiled with default CMake options using clang-12. Our test machine has an Intel Xeon Gold 5415+ processor (8 cores, 16 threads, up to 4.1 GHz clock frequency) with 128 GB of RAM. We test the performance of SEAL for NTT, iNTT and BGV multiplication on all generated ciphertexts[8]. We split these tasks between multiple threads using Barak Shoshany's thread pool library [Sho24]. We first perform a test to determine the optimal number of threads for the SEAL implementation. As shown in Figure 4.2, performance scales all the way to the maximum number of concurrent threads in the system (16). As such, in the following benchmarks, our SEAL implementation will use all available threads. Figure 4.3 shows the performance of SEAL for NTT, iNTT and BGV multiplication, which we will compare against.

[6]CKKS is an HE-scheme for arithmetic on approximate numbers (see [CKKS17]).
[7]This is the default setting, which corresponds to a 128-bit security level according to the Homomorphic Encryption Standard [ACC+18].
[8]Specifically, we call `seal::Evaluator::transform_to_ntt_inplace(seal::Ciphertext&)` and `seal::Evaluator::transform_from_ntt_inplace(seal::Ciphertext&)` for NTT/iNTT and `seal::Evaluator::multiply_inplace(seal::Ciphertext&, const seal::Ciphertext&, seal::MemoryPoolHandle)` for BGV multiplication (using thread-local memory pools).

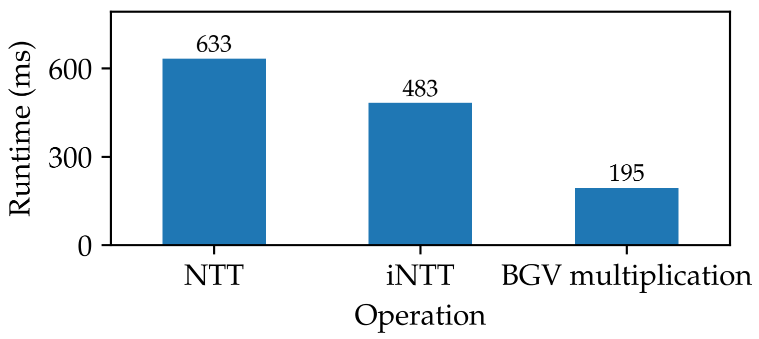


Figure 4.3: Performance of multi-threaded SEAL for NTT, iNTT and BGV multiplication, tested on 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients.

4.4.2 Initial Performance

We test the performance of our initial implementation for the same task, i.e. performing NTT and iNTT transformations as well as BGV multiplications on 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients. Note that, since our moduli are limited to 32 bits, we use a different RNS base (consisting of three 24-bit moduli). The runtime for NTT is 4737 ms, while iNTT takes 5155 ms. This is comparable to the performance we measured for SEAL when using a single thread, but is about 7.5-times and 10.7-times slower than multi-threaded SEAL for NTT and iNTT respectively. Notably, the BGV multiplication tests are only 4.2-times slower than multi-threaded SEAL, at 817 ms.

Even though our initial implementation is a lot slower than SEAL, it is reassuring that we can at least match its single-threaded performance in these tests, especially since SEAL is highly optimised.

4.4.3 Optimisations

In the following, we improve the performance of our implementation by exploring additional optimisations. Figure 4.4 shows the effect of these optimisations on the runtime of our tests. Because we expect the majority of the time to be spend on modular multiplication, we start by improving the performance of multiplication and modular reduction.

Multiplication

In Section 3.1 we tested the DPUs' performance for basic arithmetic on native integers (as implemented by the compiler) and noticed that multiplications become significantly slower for larger integers. Specifically, 64-bit multiplication is over three times slower than 32-bit multiplication.

While the RNS representation limits our inputs to 32-bit values, we still need 64-bit intermediate multiplication results to correctly perform modular reductions. However, the native 32-bit multiplication only produces 32-bit results. Thus, when using native arithmetic, we have to perform full 64-bit multiplications to get correct results, even though our inputs are only 32-bit values.

We can improve multiplication performance by implementing a custom routine, which takes 32-bit inputs and produces a 64-bit result. We take an approach similar to the native 16-bit multiplication (see Section 2.2.1) and construct our result by appropriately shifting and adding the results of 16 smaller (8x8-bit) multiplications. Our custom multiplication achieves a runtime of 37 cycles (including the function call to our routine), which is 97 cycles faster than the native 64-bit multiplication, or a time reduction of 72% and translates to a time reduction of about 13.5% for the NTT, iNTT and BGV multiplication tests.

We can apply the same technique to optimise the 32-bit multiplication that only produces 32-bit results, which is used in Barrett reductions (see below). The result is a runtime of 23 cycles (including the function call), which is up to 20 cycles faster than the native 32-bit multiplication. Recall that the runtime of the native implementation depends on the value of its inputs (see Section 2.2.1), which is because the native implementation makes a compromise: It is optimal for small inputs, which are typical in many applications, but is relatively slow in the worst case. Our custom multiplication routine has constant runtime instead. The result is that our implementation is faster when both factors are longer than 12 bits, but is otherwise slower than the native implementation. However, this is still an improvement for our use-case. Since we are operating on encrypted data, the input values are seemingly random (uniformly spread between zero and the specific modulus) and thus for a typical modulus size of ~27 bits, almost all inputs will be longer than 12 bits, which benefits our custom multiplication.

Barrett Reduction

The modular reduction after a multiplication is another slow operation, since it naively requires a 64-bit by 32-bit division. As an improvement, we can use *Barrett reduction* instead, which is optimised for repeated reductions by the same modulus and was first introduced by Barrett [Bar86] for a fast RSA implementation. We will briefly explain how it works:

Given an integer v and a modulus m, we want to find the remainder $x = v \bmod m = v - m \cdot \lfloor v/m \rfloor$. The idea of Barrett reduction is to pre-compute the reciprocal $r = m^{-1}$ of the modulus and to compute the remainder as $x = v - m \cdot \lfloor vr \rfloor$ instead. Since r will be a fractional number less than one and we do not have fast floating point arithmetic, we need

to scale and then round off r, to represent it as an integer R. This means that instead of r, we pre-compute $R = \lfloor 2^n/m \rfloor$ for some scale 2^n. The remainder can then be approximated as $x \approx v - m \cdot \lfloor vR/2^n \rfloor$. The scale 2^n must be a power of two, so that $\lfloor vR/2^n \rfloor$ can be computed using a simple bit-shift. The result is that we exchanged the original division for a multiplication and a bit-shift, which can be computed much faster. Note however, that $x \approx v - m \cdot \lfloor vR/2^n \rfloor$ only approximates the remainder, because of the rounding in $R = \lfloor 2^n/m \rfloor$. But if the scale 2^n is large enough[9], then $\lfloor vR/2^n \rfloor$ will differ from the correct value of $\lfloor v/m \rfloor$ by at most one. Thus, the resulting x will be in the range $[0, 2m)$ and can be correctly reduced modulo m with a final conditional subtraction.
In our implementation, the *Barrett factor* R of every modulus is pre-computed by the host and transferred to the appropriate DPUs as part of the auxiliary data. In the NTT, iNTT and BGV multiplication tests, Barrett reduction reduces the runtime by ~50% compared to our initial implementation. By combining it with the optimised multiplications described above, we achieve a total time reduction of ~64%, ~67% and ~71.5% for the NTT, iNTT and BGV multiplication tests respectively (compared to our initial implementation).

MRAM Buffering

In Section 3.2.4 we described a way to reduce the overhead associated with MRAM accesses by increasing the access size and buffering additional data. We can apply the same technique to our improved implementation. Since the improved implementation uses RNS to limit input values to 32-bit, it requires less stack space than the previous 128-bit implementation. We use the additional stack space to increase the buffer size of each thread up to 768 bytes. Combining this MRAM buffering with the previous optimisations yields an additional time reduction of ~36%, ~31% and ~46% for the NTT, iNTT and BGV multiplication tests respectively.

Modular Addition and Subtraction

We can also incorporate some of the optimised modular addition and subtraction techniques described in Section 3.2.4. This achieves an additional time reduction of ~1.8%, ~2.5% and ~0.9% for the NTT, iNTT and BGV multiplication tests respectively.

Scrambled Twiddle Factors and Better Buffering

By combining all of the above optimisations, we are already faster than our multi-threaded SEAL system for the BGV multiplication tests. However for the NTT and iNTT tests, our

[9]The scale should be a power of two larger than the square of the modulus. Since our moduli are limited to 32 bits, we can always use 2^{64}.

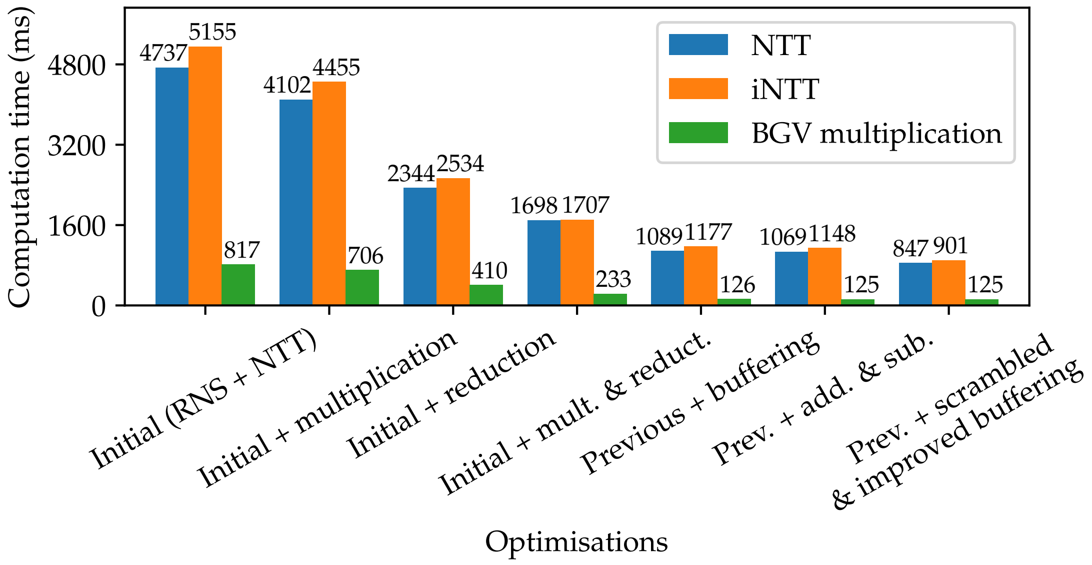


Figure 4.4: Overview of the impact of DPU optimisations on the runtime of NTT, iNTT and BGV multiplication tests for 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients.

DPU implementation is still 1.69-times slower and 2.38-times slower than SEAL respectively. Thus, we will now look more closely at our NTT and iNTT implementation and perform additional improvements.

Figure 4.5 shows the butterfly stages in our NTT and iNTT implementation. Note that the iNTT butterflies (bottom) differ slightly from the NTT Cooley-Tukey butterflies (top), which we described in Section 4.1.4. But more importantly, the order in which the twiddle factors are used, differs for NTT and iNTT. In our current implementation, iNTT iterates backwards through the polynomials (bottom to top in our illustration), to match the twiddle factors in reverse order. However, we can also reorder the twiddle factors instead, which allows iNTT to iterate forwards through memory. This results in a scrambled order of the twiddle factors, which is different from bit-reversed order. To be precise, the i-th inverse root of unity power ψ^{-i} is stored in position $1 + \text{bit\_reverse}(i - 1)$.

On its own, this only improves iNTT performance by ~1%, but it allows us to further improve MRAM buffering. Previously, we only buffered the coefficients within each butterfly group sharing a twiddle factor. This works great for butterfly stages which consist of few large groups, like the early stages of NTT, or the final stages of iNTT. But it is suboptimal for stages consisting of smaller groups, since the number of buffered coefficients is limited by the group size. For example, in the second stage of an iNTT, only four coefficients can be buffered with our previous implementation.

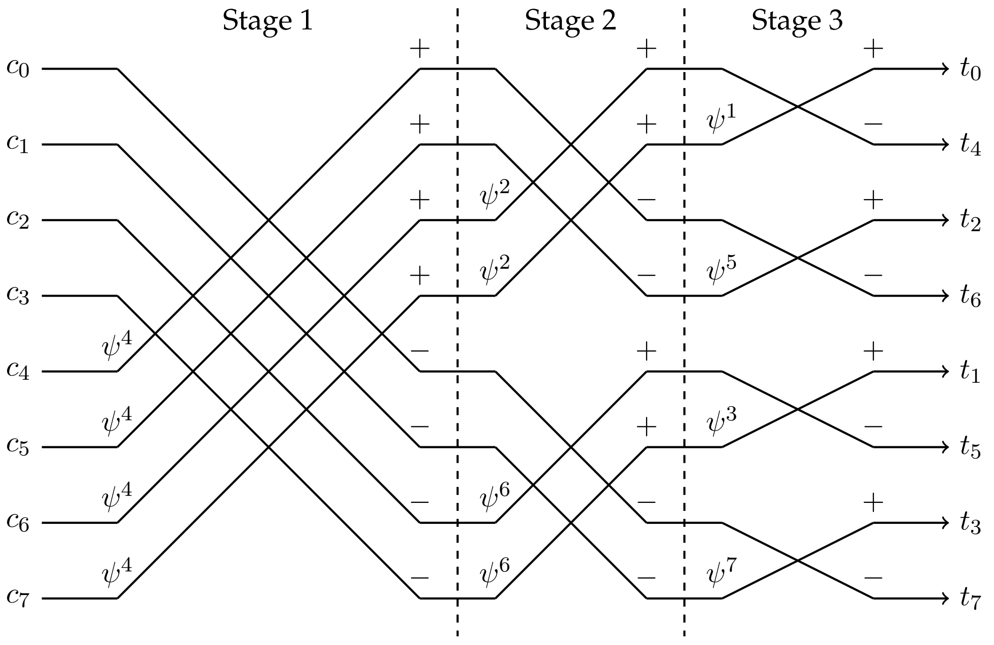
Our improved implementation can instead utilise the whole buffer space in all stages. This results in an additional ~20.7% time reduction for the NTT tests. Due to the scrambled twiddle factors, we can also apply this better buffering to iNTT, for an additional ~21.5% time reduction.

Resulting Performance

Figure 4.4 shows the impact of the optimisations described above on the NTT, iNTT and BGV multiplication tests. Compared to our initial implementation, we achieved a total time reduction of ~82.3% for NTT and iNTT, and a time reduction of ~84.7% for BGV multiplication. Thus, our optimised implementation is only 1.34-times slower and 1.87-times slower than multi-threaded SEAL for NTT and iNTT respectively. For the BGV multiplication tests, our optimised implementation is actually faster than SEAL, requiring about 35.9% less time.

Even though FHE is memory bound on conventional architectures, our DPU implementation is still constrained by computational performance, which is mainly due to the slow multiplication on DPUs for integers larger than 8 bits. We confirmed this by running additional tests, in which we replaced our multiplication routines with dummy versions, which take ~83% less time to execute. This resulted in an additional time reduction of ~68% for BGV multiplication and ~62.5% for NTT and iNTT compared to our optimised implementation. Of course, these are not “real” results, as the dummy methods do not actually perform multiplications and thus the resulting ciphertexts are wrong, but they illustrate the impact that faster multiplications would have on the performance of our DPU implementation.

As it stands, our optimised DPU implementation is slightly faster than SEAL for BGV multiplication only, but slower than SEAL for NTT and iNTT. In theory, many such multiplications can be performed after a single NTT operation, before transforming back using iNTT. Thus, the speed advantage of the PIM-based multiplication could accumulate and offset the slower NTT operations. However, this would require roughly nine multiplications between an NTT/iNTT pair just to break even, which is unrealistic, because relinearisation (and modulus switching) would have to be performed in between. In our implementation, this also requires inter-DPU communication, which is relatively slow. If however, the data is already transformed, e.g. because it was generated/sent in NTT form by the client, then the NTT performance of our implementation is less important and it could be possible to benefit from the faster BGV multiplication. Unfortunately, this would not be a very significant advantage, especially since our SEAL test system does not use the fastest and most recent CPU model, which would likely improve its performance and negate the slight benefit of our DPU implementation. This comparison also does not



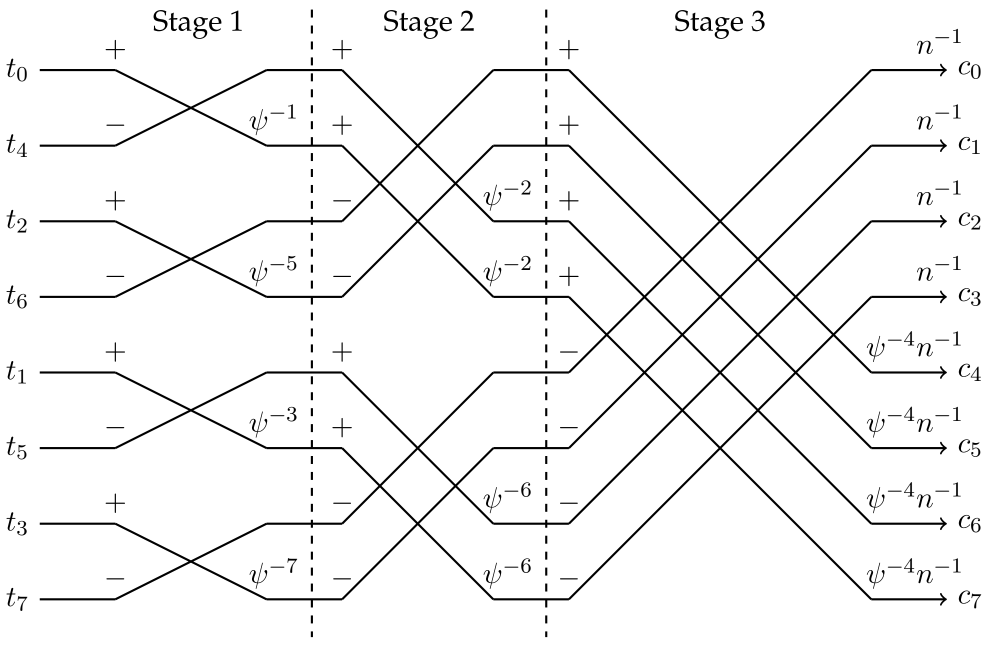


Figure 4.5: Illustration of the butterfly stages for NTT (top) and iNTT (bottom). Shown here for $n = 8$ with $\log_2(n) = 3$ stages. The bit-reversed order of the root-of-unity-powers[a] is $\psi^4, \psi^2, \psi^6, \psi^1, \psi^5, \psi^3, \psi^7$. Note that this perfectly matches the order of the NTT butterflies (top to bottom, left to right), while the iNTT butterflies must be evaluated in a different order (bottom to top, left to right) to match the twiddle factors in reverse. This can result in sub-optimal memory access patterns.

[a]For iNTT, these are powers of the inverse root of unity ψ^{-1}, i.e. $\psi^{-4}, \psi^{-2}, \psi^{-6}$ and so on.

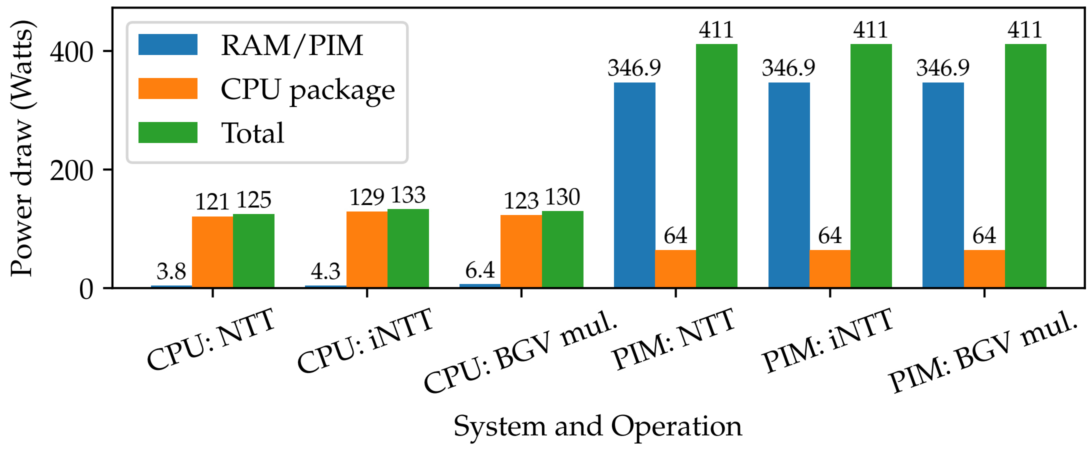


Figure 4.6: Power draw of the SEAL and DPU implementations while looping operations on 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients.

consider the energy efficiency, which we will evaluate next.

4.4.4 Energy Efficiency

For comparing energy efficiency against Microsoft SEAL, we test our most optimised DPU implementation. We use the Linux `perf` tool to measure the power consumption of the test systems. Specifically, we monitor the `/power/energy-pkg/` and `/power/energy-ram/` events. This uses the Intel RAPL (Running Average Power Limit) interface, which allows separately measuring the power of the *package domain* and the *memory domain*, which correspond to the processor die (CPU) and the attached DRAM respectively [Int24].

The CPU power measurement works on both test systems. However, the memory power measurement is only valid for the CPU system. For the PIM system, it is not currently possible to measure the power consumption of the PIM DIMMs in software. We also do not have access to the cloud servers, to perform hardware measurements. Thus, we have to estimate the power consumption of the PIM DIMMs, which according to UPMEM, consume about 20 Watts when fully loaded.

Our PIM system has 20 PIM DIMMs, which contain 2560 DPUs in total. However, we can only use 2220 of these DPUs, since some of them are disabled. Assuming that this ~86.7% utilisation transfers linearly to the power consumption, we expect the PIM DIMMs to draw 346.9 Watts during our tests.

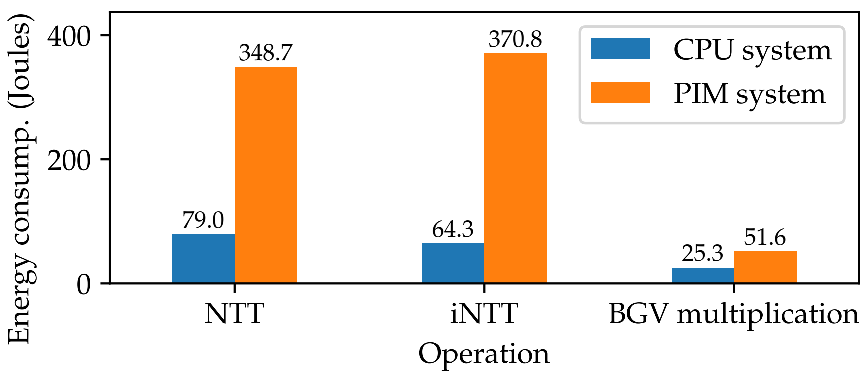


Figure 4.7: Energy consumption of the SEAL and DPU implementations for operations on 32768 ciphertexts with polynomials of length 4096 and 72-bit coefficients.

Power Draw

Figure 4.6 shows the power draw of our CPU and PIM systems while looping NTT, iNTT and BGV multiplication tasks. We can see that in the CPU system, the majority of the power is used by the CPU, while in the PIM system, most of the power is used by the PIM DIMMs. In total, the PIM system consumes more power than the CPU system in these tests. While the CPU of the PIM system is mostly idle during the tests, it still consumes a considerable amount of power. The system seems to continuously saturate 1–2 CPU threads, while the DPUs are running, which might prevent the CPU from entering a lower power state. We suspect that this is due to some internal bookkeeping or synchronisation in the UPMEM SDK. Note however, that the idle CPU threads can also be used to perform other tasks, which could potentially reduce the effective power consumption of our DPU implementation, if the CPU is already in a high power state.

Energy Consumption

By combining the results of the power draw tests and the earlier runtime tests, we can calculate the energy requirements for these operations on the CPU and PIM systems. This is shown in Figure 4.7. We can see that the PIM system requires more energy than the CPU system for all tested operations. Even tough our DPU implementation is faster than SEAL for BGV multiplication, this advantage is offset by the higher power consumption.

4.4.5 Ciphertext Moduli

When choosing a ciphertext modulus, the individual RNS moduli should be as large as possible, while still fitting into a machine word, as this minimises the number of moduli

and thus the number of sub-polynomials which need to be manipulated. Additionally, it maximises the efficiency of operations on these machine words, because the operations are typically constant-time, regardless of how "full" their input words are. However, because DPUs have 32-bit words and SEAL uses 64-bit words, their optimal RNS bases are different and using ciphertext moduli of the same size as SEAL can be a suboptimal choice for our DPU implementation.
In the above tests, SEAL uses an RNS base with two 36-bit moduli for ciphertexts, which corresponds to three 24-bit moduli on DPUs. When choosing such an RNS base for DPUs directly, we could have used three 27-bit moduli instead, which would increase the ciphertext modulus and potentially allow for more homomorphic operations before requiring bootstrapping. However, in our benchmarks, we only care about the runtime of these operations and since our optimised multiplication and Barrett reduction are constant-time, their runtime is the same for 24-bit and 27-bit moduli. Thus, only our unoptimised implementations are affected by the moduli discrepancy described above.

4.4.6 Unsuitable Optimisations

In this section, we discuss some additional optimisations which we considered, but which were unsuitable for our use case. We focus on optimising multiplication, since our implementation is mostly constrained by its multiplication performance.

Smaller RNS Moduli

RNS allows us to operate on smaller values and we use it to split large coefficients into groups of 32-bit integers. Since multiplication of 16-bit integers is even faster, one could ask why we do not split our coefficients into 16-bit or even 8-bit integers instead. The reason is that we still need to perform NTT on the resulting sub-polynomials. As stated in Section 4.2.3, each RNS moduli must also be a working modulus for an NTT of the polynomial length. For a typical polynomial length of 4096, this means that all RNS moduli must be one higher than a multiple of 8192. In the numbers up to 2^{16}, we could thus find at most 7 such moduli (even fewer in practice), which is insufficient for almost all use cases of FHE. We chose 32-bit moduli for our implementation, because this is the smallest (and thus fastest) common bit size, which is still practical for our use case.

Karatsuba Multiplication

As described in Section 4.4.3, we perform 32-bit multiplication by appropriately shifting and adding the results of $4^2 = 16$ smaller (8x8-bit) multiplications. It seems obvious to use Karatsuba multiplication instead, which would only require $3^2 = 9$ multiplications.

However, this is not actually faster, since the additional bookkeeping, like additions, subtractions and register shuffling, offset the potential savings. It is easy to see that the native 16-bit multiplication can not be sped-up using Karatsuba multiplication, as its constituent 8-bit multiplications require just one cycle each. Thus, Karatsuba multiplication could only be useful for reducing the number of 16-bit multiplications that are part of a 32-bit multiplication. Consider two 32-bit numbers $a = a_0a_1$ and $b = b_0b_1$ split into the 16-bit parts a_0, a_1, b_0 and b_1. The classical Karatsuba multiplication requires computing $(a_0 + a_1) \cdot (b_0 + b_1)$. However, since the parts are 16-bit each, $a_0 + a_1$ and $b_0 + b_1$ can be 17 bits long and $(a_0 + a_1) \cdot (b_0 + b_1)$ must then be computed as a 17x17-bit multiplication. This is difficult to implement and slower than the naive version. There is an alternative Karatsuba implementation, which requires computing $|a_0 - a_1| \cdot |b_0 + b_1|$ instead and then applying the expected sign to the result. This stores the 17-th bits separately (as the expected sign) and thus only requires 16x16-bit multiplications. However, the additional comparisons (for computing the absolute differences $|a_0 - a_1|$ and $|b_0 + b_1|$) and the conditional negation of the result, together with the aforementioned bookkeeping are more expensive than just performing an additional multiplication. Even with a hand optimised assembly implementation, we could only achieve a runtime of 41 cycles, which is slower than our multiplication routine described in Section 4.4.3, which takes 37 cycles. Note however, that Karatsuba multiplication is beneficial for 64-bit and 128-bit multiplications, as used in chapter 3.

5 Conclusions

In this chapter, we summarise our results and provide an outlook into possible further research and future advancements.

5.1 Summary

We presented a detailed evaluation on the suitability of the UPMEM PIM system for accelerating FHE operations. As part of this evaluation, we explored the DPU architecture and its performance characteristics, devised and implemented many optimisations and tested their impact on different operations which are typical for FHE.

We analysed the implementation and performance of the native arithmetic operations on DPUs, showing that DPUs are constrained by multiplication performance. We explained the RNS and NTT techniques for optimising multiplication, as well as their parameter requirements. Using these techniques, we developed an improved implementation of polynomial operations on DPUs, which can be combined into more complex FHE operations. We also designed a suitable interface for communicating with DPUs and described our threading models for different operations. We further improved the performance of our implementation by applying additional optimisations, including a custom multiplication routine optimised for the expected input values of our use case. Additionally, we described our implementation of Barrett reduction, which replaces the division-based reduction algorithm and greatly improves the performance of modular reduction. With these optimisations, we significantly improved upon prior results [GKG$^+$23] for HE performance on an UPMEM PIM system. Furthermore, we compared the performance of our implementation with Microsoft SEAL, which is a popular CPU implementation for FHE. We showed that our optimised BGV multiplication using DPUs is slightly faster than a comparison system running SEAL, but that NTT and iNTT operations remain slower and our DPU implementation is still limited by multiplication performance. Finally, we calculated the power draw and energy efficiency of the PIM system and compared it against the CPU system running SEAL. The results show that the higher power draw of the PIM system negates its speed advantage for BGV multiplication and results in higher power consumption than the CPU system for all tested operations.

5.2 Answering the Research Question

We evaluated whether the UPMEM PIM system is suitable for improving the throughput or energy efficiency of FHE operations compared to other optimised implementations. Based on our results, we come to the conclusion that UPMEM PIM (at least in its current version) is not suitable for improving the energy efficiency of FHE operations or for improving their throughput in a significant way compared to other optimised implementations.

Although we greatly improve upon previous results, FHE operations on UPMEM PIM remain bound by multiplication performance. This is caused by the limited hardware multiplication support in current UPMEM DPUs. Additionally, the higher power consumption of the PIM system results in unfavourable energy efficiency.

5.3 Outlook

While CPUs have been developed and improved upon for decades, UPMEM's DPUs are relatively new and the future might still hold many advancements for the technology, which could improve its performance and open up new use cases. We would especially like to see some sort of *hybrid computing*, in which the CPU and the DPUs can operate on the same data in memory. This could allow each component to focus on the (sub-)tasks, for which it is most performant. For example, in our case, the CPU might perform the NTT and iNTT transformations, but hand-off the BGV multiplication to the DPUs. In current UPMEM PIM systems, this is not possible, since the data must first be copied into the memory of the DPUs and must later be retrieved. The resulting data transfer overhead reduces the opportunities for such hand-offs.

As possible future work, one might evaluate FHE acceleration using different PIM approaches or products, like Samsung's AxDIMMs, which have a DIMM-compatible interface like UPMEM PIM, but use a programmable FPGA fabric instead of DPUs [KZS+22]. One might also evaluate the suitability of PIM systems for other FHE operations like relinearisation and modulus switching, or multiplying ciphertexts with plaintexts.

References

[ACC$^+$18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BGK$^+$18] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 316–331. ACM, 2018.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[BVL$^+$21] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Xiao Nan, Kazuaki Matsumura, and Khin Mi Mi Aung. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Trans. Parallel Distributed Syst.*, 32(2):379–391, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CRS17] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. Designing an FPGA-accelerated homomorphic encryption co-processor. *IEEE Trans. Emerg. Top. Comput.*, 5(2):193–206, 2017.

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

[dCAY$^+$21] Leo de Castro, Rashmi Agrawal, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. Does fully homomorphic encryption need compute acceleration? *IACR Cryptol. ePrint Arch.*, page 1636, 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[GHF$^+$21] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *CoRR*, abs/2105.03814, 2021.

[GKG$^+$23] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. Evaluating homomorphic operations on a real-world processing-in-memory system. In *IEEE International Symposium on Workload Characterization, IISWC 2023, Ghent, Belgium, October 1-3, 2023*, pages 211–215. IEEE, 2023.

[Int24] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Volume 3B)*, October 2024. Available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):114–148, 2021.

[KZS+22] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shinhaeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, IlKwon Yun, Sung Joo Park, Hyunsun Park, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with AxDIMM. *IEEE Micro*, 42(1):116–127, 2022.

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

[NGI+20] Hamid Nejatollahi, Saransh Gupta, Mohsen Imani, Tajana Simunic Rosing, Rosario Cammarota, and Nikil D. Dutt. CryptoPIM: In-memory acceleration for lattice-based cryptographic hardware. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 143–163. Springer, 2015.

[SEA23] Microsoft SEAL (release 4.1). `https://github.com/Microsoft/SEAL`, January 2023. Microsoft Research, Redmond, WA.

[Sho24] Barak Shoshany. A C++17 thread pool for high-performance scientific computing. *SoftwareX*, 26:101687, 2024.

[UPM23] UPMEM website: Use cases. `https://www.upmem.com/use-cases/`, 2023.