Accelerating Homomorphic Evaluation on Reconfigurable Hardware

Thomas Pöppelmann[1](✉), Michael Naehrig[2], Andrew Putnam[2], and Adrian Macias[3]

[1] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Bochum, Germany
thomas.poeppelmann@rub.de

[2] Microsoft Research, Redmond, WA, USA
{mnaehrig,anputnam}@microsoft.com

[3] Altera Corporation, San Diego, CA, USA
amacias@altera.com

Abstract. Homomorphic encryption allows computation on encrypted data and makes it possible to securely outsource computational tasks to untrusted environments. However, all proposed schemes are quite inefficient and homomorphic evaluation of ciphertexts usually takes several seconds on high-end CPUs, even for evaluating simple functions. In this work we investigate the potential of FPGAs for speeding up those evaluation operations. We propose an architecture to accelerate schemes based on the ring learning with errors (RLWE) problem and specifically implemented the somewhat homomorphic encryption scheme YASHE, which was proposed by Bos, Lauter, Loftus, and Naehrig in 2013. Due to the large size of ciphertexts and evaluation keys, on-chip storage of all data is not possible and external memory is required. For efficient utilization of the external memory we propose an efficient double-buffered memory access scheme and a polynomial multiplier based on the number theoretic transform (NTT). For the parameter set ($n = 16384, \lceil \log_2 q \rceil = 512$) capable of evaluating 9 levels of multiplications, we can perform a homomorphic addition in 0.94 ms and a homomorphic multiplication in 48.67 ms.

Keywords: Homomorphic encryption · Ring learning with errors · FPGA · Reconfigurable computing

1 Introduction

A homomorphic encryption scheme enables a third party to perform meaningful computation on encrypted data and a prime example for an application is the outsourcing of a computational task into an untrusted cloud environment (see, e.g., [5,12,13,28]). Such schemes come in different flavors, the most versatile being a fully homomorphic encryption (FHE) scheme, which allows an

T. Pöppelmann—This work was mainly carried out when the author was an intern in the Cryptography Research group at Microsoft Research, Redmond.

unlimited number of operations. The first FHE scheme was proposed by Gentry in 2009 [23] and led to many new schemes optimized for better efficiency or security (e.g., [6,8,16,24,26,30,39]). FHE schemes usually consist of a so-called somewhat or leveled homomorphic scheme with limited functionality together with a procedure to bootstrap its capabilities to an arbitrary number of operations. The somewhat homomorphic encryption (SHE) schemes are usually a lot more efficient than their corresponding FHE counterparts because bootstrapping imposes a significant overhead. Examples of SHE schemes are the BGV [8] and LTV [30] schemes and the subsequent YASHE [4] scheme, which are relatively straightforward and conceptually simple as they mainly require polynomial multiplication and (bit level) manipulation of polynomial coefficients for evaluation of ciphertexts (i.e., mul, add). But even limited SHE schemes are still slow and especially for relatively complex computations, evaluation operations can take several hours, even on high-end CPUs [25,29]. A natural question concerning FHE and SHE is whether reconfigurable hardware can be used to accelerate the computation. However, as ciphertexts and keys are large and require several megabytes or even gigabytes of storage for meaningful parameter sets, the internal memory of FPGAs is quickly exhausted, and required data transfers between host and FPGA might degrade the achievable performance.

These may be reasons that previous work mainly focuses on using GPUs [19,40,41] and ASICs [21,44], and that FPGA implementations either work only with small parameters and on-chip memory [11] or explicitly do not take into account the complexity of transferring data between an FPGA and a host [9,35]. For our implementation we use the Catapult data center acceleration platform [34], which provides a Stratix V FPGA on a PCB with two 4 GB memory modules inserted into the expansion slot of a cloud server. This fits nicely into the obvious scenario in which homomorphic evaluation operations are carried out on encrypted data stored in the cloud. Since future data centers might be equipped with such accelerators, it makes sense to consider the Catapult architecture as a natural platform for evaluating functions with homomorphic encryption.

Our Contribution. To our knowledge, we provide the first fully functional FPGA implementation of the homomorphic evaluation operations of an RLWE-based SHE scheme. Our main contribution is an efficient architecture for performing number theoretic transforms, which is used to implement the SHE scheme YASHE. Compared to previous FPGA implementations of integer-based FHE schemes (e.g., [9]) we especially take into account the complexity of using off-chip memory. Thus we propose and evaluate the usage of the cached-NTT [2,3] for bandwidth-efficient computations of products of large polynomials in $\mathbb{Z}_q[X]/(X^n+1)$ and the YASHE specific parts of the KeySwitch and Mult algorithms. The main computational burden is handled by a large integer multiplier built out of DSP blocks and modular reduction using Solinas primes. An implementation of the parameter set ($n = 16384, \lceil \log_2 q \rceil = 512$) that can handle computations on the encrypted data of multiplicative depth up to $L = 9$ levels (for $t = 1024$) roughly matches the performance of a software implementation of the parameter set ($n = 4096$, $\lceil \log_2 q \rceil = 128$) supporting just one level [29]. With only 48.67 ms for a homomorphic multiplication (instead of several seconds in software) we provide evidence that hardware-accelerated somewhat homomorphic cryptography can be made practical for certain application scenarios.

2 Background

2.1 Somewhat Homomorphic Scheme YASHE

The homomorphic encryption scheme YASHE [4] is based on the multi-key FHE scheme from [30] and the modified, provably-secure version of NTRU in [38]. In [4], two versions of YASHE are presented. We use the more efficient variant.

The system parameters are fixed as follows: a positive integer $m = 2^k$ that determines the ring $R = \mathbb{Z}[X]/(X^n + 1)$ and its dimension $n = \varphi(m) = m/2$, two moduli q and t with $1 < t < q$, discrete probability distributions $\chi_{\text{key}}, \chi_{\text{err}}$ on R, and an integer base $w > 1$. We view R to be the ring of polynomials with integer coefficients taken modulo the m-th cyclotomic polynomial $X^n + 1$. Let $R_q = R/qR \cong \mathbb{Z}_q[X]/(X^n + 1)$ be defined by reducing the elements in R modulo q, similarly we define R_t. A polynomial $\mathbf{a} \in R_q$ can be decomposed using base w as $\mathbf{a} = \sum_{i=0}^{\ell_{w,q}-1} \mathbf{a}_i w^i$, where the $\mathbf{a}_i \in R$ have coefficients in $(-w/2, w/2]$. The scheme YASHE makes use of the functions $\mathsf{Dec}_{w,q}(\mathbf{a}) = ([\mathbf{a}_i]_w)_{i=0}^{\ell_{w,q}-1}$ and $\mathsf{Pow}_{w,q}(\mathbf{a}) = ([\mathbf{a}w^i]_q)_{i=0}^{\ell_{w,q}-1}$, where $\ell_{w,q} = \lfloor \log_w(q) \rfloor + 1$. Both functions take a polynomial and map it to a vector of polynomials in $R^{\ell_{w,q}}$. They satisfy the scalar product property $\langle \mathsf{Dec}_{w,q}(\mathbf{a}), \mathsf{Pow}_{w,q}(\mathbf{b}) \rangle = \mathbf{ab} \pmod q$.

YASHE consists of the following algorithms. Note that homomorphic multiplication Mult consists of two parts, the rounded multiplication RMult and the key switching step KeySwitch.

$\mathsf{KeyGen}(d, q, t, \chi_{\text{key}}, \chi_{\text{err}}, w)$: Sample $\mathbf{f}' \leftarrow \chi_{\text{key}}$ until $\mathbf{f} = [\mathbf{t}\mathbf{f}' + 1]_q$ is invertible modulo q. Compute the inverse $\mathbf{f}^{-1} \in R$ of $\mathbf{f}$ modulo q, sample $\mathbf{g} \leftarrow \chi_{\text{key}}$ and set $\mathbf{h} = [\mathbf{t}\mathbf{g}\mathbf{f}^{-1}]_q$. Sample $\boldsymbol{e}, \boldsymbol{s} \leftarrow \chi_{err}^{\ell_{w,q}}$, compute $\boldsymbol{\gamma} = [\mathsf{Pow}_{w,q}(\mathbf{f}) + \mathbf{e} + \mathbf{h} \cdot \boldsymbol{s}]_q \in R^{\ell_{w,q}}$ and output $(\mathsf{pk}, \mathsf{sk}, \mathsf{evk}) = (\mathbf{h}, \mathbf{f}, \boldsymbol{\gamma})$.

$\mathsf{Encrypt}(\mathbf{h}, \mathbf{m})$: For a message $\mathbf{m} \in R/tR$, sample $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\text{err}}$, scale $[\mathbf{m}]_t$ by the value $\lfloor q/t \rfloor$, and output $\mathbf{c} = \left[\left\lfloor \frac{q}{t} \right\rfloor [\mathbf{m}]_t + \mathbf{e} + \mathbf{hs} \right]_q \in R$.

$\mathsf{Decrypt}(\mathbf{f}, \mathbf{c})$: Compute $[\mathbf{fc}]_q$ modulo q, scale it down by t/q over the rational numbers, round it and reduce it modulo t, i.e. output $\mathbf{m} = \left[\left\lfloor \frac{t}{q} [\mathbf{fc}]_q \right\rceil \right]_t \in R$.

$\mathsf{Add}(\mathbf{c}_1, \mathbf{c}_2)$: Add the two ciphertexts modulo q, i.e. output $\mathbf{c}_{\text{add}} = [\mathbf{c}_1 + \mathbf{c}_2]_q$.

$\mathsf{RMult}(\mathbf{c}_1, \mathbf{c}_2)$: Compute $\mathbf{c}_1\mathbf{c}_2$ without reduction modulo q over the integers, scale by t/q, round the result and reduce modulo q to output $\tilde{\mathbf{c}}_{\text{mult}} = \left[\left\lfloor \frac{t}{q} \mathbf{c}_1 \mathbf{c}_2 \right\rceil \right]_q$.

$\mathsf{KeySwitch}(\tilde{\mathbf{c}}_{\text{mult}}, \mathsf{evk})$: Compute the w-decomposition vector of $\tilde{\mathbf{c}}_{\text{mult}}$ and output the scalar product with evk modulo q: $\mathbf{c}_{\text{mult}} = [\langle \mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}), \mathsf{evk} \rangle]_q$.

Table 1. YASHE parameter sets and supported number of multiplicative levels for different plaintext moduli t, using discrete Gaussian error parameter $s = 8$.

Set	n	q	q'	$\ell_{w,q}$	Levels			
					$t = 2^{20}$	$t = 2^{10}$	$t = 2^5$	$t = 2$
I	4096	$2^{124} - 2^{64} + 1$	$2^{262} - 2^{56} + 1$	2	0	1	1	1
II	16384	$2^{512} - 2^{32} + 1$	$2^{1040} - 2^{32} + 1$	8	6	9	11	14

$\mathsf{Mult}(\mathbf{c}_1, \mathbf{c}_2, \mathsf{evk})$: First apply RMult to $\mathbf{c}_1$ and $\mathbf{c}_2$ and then $\mathsf{KeySwitch}$ to the result. Output the ciphertext $\mathbf{c}_{\text{mult}} = \mathsf{KeySwitch}(\mathsf{RMult}(\mathbf{c}_1, \mathbf{c}_2), \mathsf{evk})$.

In Table 1, we provide the implemented parameter sets and their number of supported multiplicative levels determined by the worst case bounds given in [4]. The plaintext modulus in our implementation is $t = 1024$ for both parameter sets. Since changing t is relatively easy, we also give the number of multiplicative levels for various other choices to illustrate the dependence on t and possible trade-offs. According to the analysis in [29], moduli stay below the maximal bound to achieve 80 bits of security against the distinguishing attack with advantage 2^{-80} as discussed there. The error distribution χ_{err} is the n-dimensional discrete Gaussian with parameter $s = 8$ and the key distribution samples polynomials with uniform random coefficients in $\{-1, 0, 1\}$. Note that one ciphertext requires $n\lceil\log_2(q)\rceil$ bits (1 MiB for Set II) and the evaluation key is $(\ell_{w,q})n\lceil\log_2(q)\rceil$ bits large (8 MiB for parameter Set II).

2.2 Number Theoretic Transform

Polynomial multiplication can be performed with $\mathcal{O}(n \log n)$ operations in $\mathbb{Z}_q$ using the number theoretic transform (NTT), which is basically an FFT defined over a finite field or ring. Given a primitive n-th root of unity ω the forward transformation $\text{NTT}_q(\mathbf{a})$ of a length-n sequence $(\mathbf{a}[0], .., \mathbf{a}[n-1])$ with elements in $\mathbb{Z}_q$ is defined as $\mathbf{A}[i] = \sum_{j=0}^{n-1} \mathbf{a}[j]\omega^{ij} \bmod q$ and the inverse transformation $\text{INTT}_q(\mathbf{A})$ as $\mathbf{a}[i] = n^{-1}\sum_{j=0}^{n-1} \mathbf{A}[j]\omega^{-ij} \bmod q$ for $i = 0, 1, ..., n-1$ (see [18,31,45] for more information on the NTT). For efficient multiplication of polynomials in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, one can use the negative wrapped convolution, which removes the need for zero padding of input polynomials. Let ω be a primitive n-th root of unity in $\mathbb{Z}_q$ and $\psi^2 = \omega$. For two polynomials $\mathbf{a} = \mathbf{a}[0] + \mathbf{a}[1]X + \cdots + \mathbf{a}[n-1]X^{n-1}$ and $\mathbf{b} = \mathbf{b}[0] + \mathbf{b}[1]X + \cdots + \mathbf{b}[n-1]X^{n-1}$ of degree at most $n-1$ with elements in $\mathbb{Z}_q$, we define $\mathbf{d} = \mathbf{d}[0] + \cdots + \mathbf{d}[n-1]X^{n-1}$ as the negative wrapped convolution of $\mathbf{a}$ and $\mathbf{b}$ so that $\mathbf{d} = \mathbf{a} * \mathbf{b} \bmod (X^n + 1)$. We further define the representation $\hat{\mathbf{y}} = \mathbf{y}[0] + \psi\mathbf{y}[1]X + \cdots + \psi^{n-1}\mathbf{y}[n-1]X^{n-1}$ and use it as $\hat{\mathbf{a}}, \hat{\mathbf{b}}$ and $\hat{\mathbf{d}}$. In this case it holds that $\hat{\mathbf{d}} = \text{INTT}_q(\text{NTT}_q(\hat{\mathbf{a}}) \circ \text{NTT}_q(\hat{\mathbf{b}}))$, where $\circ$ means coefficient-wise multiplication [18,45].

Various algorithms that implement the FFT efficiently and which are directly applicable for the NTT are reviewed in [14]. A popular choice is a radix-2, in-place, decimation-in-time (DIT) [15] or decimation-in-frequency (DIF) [22] algorithm that requires roughly $\frac{n}{2}\log_2(n)$ multiplications in $\mathbb{Z}_q$ (see [1,33,36] for implementation results). Note that in the FFT context precomputed powers of the primitive root of unity ω are often referred to as *twiddle factors*.

The primes q and q' we use in our implementation are Solinas primes of the form $q = 2^y - 2^z + 1$, $y > z$ such that $q \equiv 1 \pmod{2n}$. In order to find a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$ that is needed in the NTT transforms as mentioned above, we simply chose random non-zero elements in $a \in \mathbb{Z}_q$, until $a^{(q-1)/2n} \neq 1$ and $a^{(q-1)/2} = -1$ and then set $\psi = a^{(q-1)/2n}$.

2.3 Cached-FFT

The general idea of the cached-FFT algorithm [2,3], as visualized in Fig. 1, is to divide the FFT computation into epochs (E) after which an out-of-place reordering step becomes necessary. In an epoch itself the data is split into groups (G) consisting of $C = n/G$ coefficients and computations require only access to members of a group but do not interfere with or require values from other groups. The required computation on a group is just a standard Cooley-Tukey, radix-2, in-place, DIT FFT/NTT [14,15], denoted as C-NTT and the number of stages or passes (recursive divisions into sub-problems) of the C-NTT is $P = \log_2(n/G)$. Thus one C-NTT on a group requires $\frac{Pn}{2G}$ multiplications in $\mathbb{Z}_q$. As a consequence, during the computation of an NTT/FFT on a group, this group can be stored in a small cache or local memory that supports fast access to coefficients.

For the actual details of the implementation of address generation we refer to the description in [2,3]. However, referring to the $E = 2$ case displayed in Fig. 1, it is easy to see that, with a hardware implementation in mind, it is necessary to read $2n$ coefficients from the main memory and to write $2n$ coefficients back to the main memory to compute the FFT. However, only two of these reads/writes are non-consecutive (i.e., the reordering) while two read/writes are in order.

2.4 Catapult Architecture/Target Hardware

Because a primary application of homomorphic encryption is use in untrusted clouds, we chose to implement YASHE using a previously proposed FPGA-based datacenter accelerator infrastructure called Catapult [34]. Catapult augments a conventional server with an FPGA card attached via PCIe that features a medium size Stratix V GS D5 (5GSMD5) FPGA, two 4 GB DDR3-1333 SO-DIMM (small outline dual inline memory module) memory modules, and a private inter-FPGA 2-D torus network. In the original work, Catapult was used to accelerate parts of the Bing search engine, and a prototype consisting of 1,632 servers was deployed. The two DRAM controllers on the board can be used either independently or combined in a unified interface. When used independently the DIMM modules are clocked with 667 MHz. The Catapult shell [34, Sect. 3.2.] provides a simple interface to access the DRAM and to communicate with the host server. It uses roughly 23 % of the available device resources, depending on the used functionality like DRAM, PCIe, or 2-D torus network. Application logic is implemented as a role.

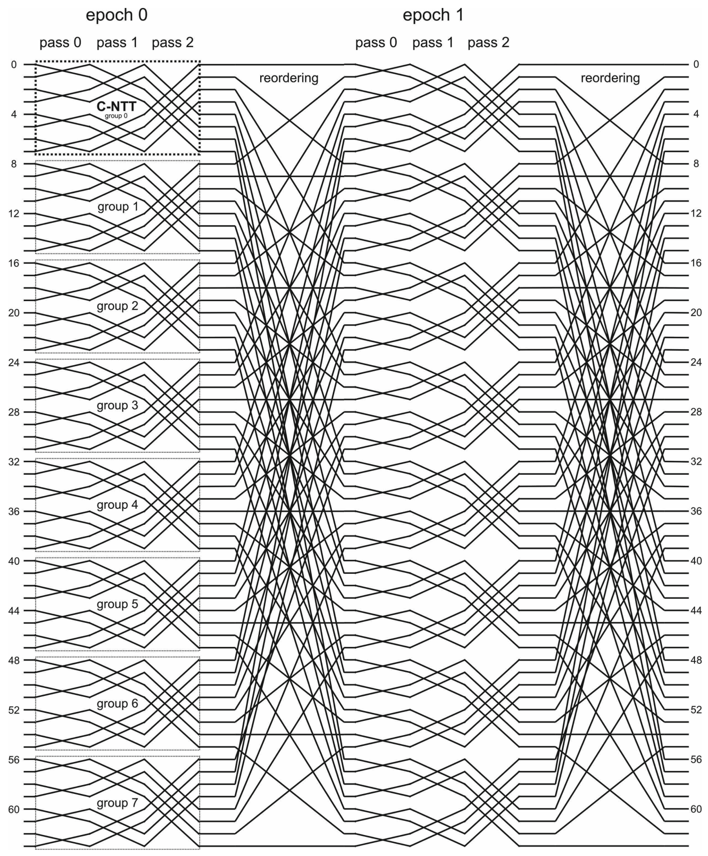

Fig. 1. Dataflow diagram of a 64-point cached-FFT split into two epochs with eight coefficients in each group/cache parameterized as ($n = 64$, $E = 2$, $G = 8$, $P = 3$, $C = 8$). This figure is based on [3, Fig. 3].

For our design, we restrict the accelerator to only a single FPGA card per server. Spanning multiple FPGAs is a promising avenue for improving performance, but is left for future work. Note also that none of the work presented here is exclusive to Catapult and that any FPGA board with two DRAM channels, a sufficiently large FPGA, and fast connection to a host server will suffice. However, Catapult is specifically designed for datacenter workloads, so it presents realistic constraints on cost, area, and power for our accelerator.

3 High Level Description

The goal of our implementation is to accelerate the (cloud) server-based evaluation operations Mult and Add of YASHE (and polynomial multiplication in general) without interaction with the host server using the Catapult infrastructure. Key generation, encryption, and decryption are assumed to be performed on a client and are not in the scope of this work. However, we would like to note that except for a Gaussian sampler, most components required for key generation, encryption, and decryption are already present in our design.

Our main building block is a scalable NTT-based polynomial multiplier that supports the two moduli q and q'. The computation of the NTT is by far the most expensive operation and necessary for the polynomial multiplications in RMult and $\mathsf{KeySwitch}$, which are called during a Mult operation. Other computations like polynomial addition or pointwise multiplication are realized using the hardware building blocks from the NTT multiplier. The modulus $q' > nq^2$ is used to compute

$$\mathbf{c}_1\mathbf{c}_2 = \mathrm{INTT}_{q'}(\mathrm{NTT}_{q'}(\mathbf{c}_1)\circ\mathrm{NTT}_{q'}(\mathbf{c}_2))$$

in RMult exactly without modular reduction as each coefficient of $\mathbf{c}_1$ and $\mathbf{c}_2$ is smaller than q and thus each coefficient of the result is guaranteed to be smaller than nq^2. Reductions modulo q are required for the computation of the scalar product $\mathbf{c}_{\text{mult}} = [\langle \mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}), \mathsf{evk}\rangle]_q$ in $\mathsf{KeySwitch}$ and the polynomial addition in Add. A naive implementation of $\mathsf{KeySwitch}$ would require $\ell_{w,q}$ polynomial multiplications and $\ell_{w,q} - 1$ polynomial additions. By using the NTT and its linearity we just compute

$$\mathsf{KeySwitch}(\tilde{\mathbf{c}}_{\text{mult}}, \overline{\mathsf{evk}}) = \mathrm{INTT}_q\left(\sum_{i=0}^{\ell_{w,q}-1} \mathrm{NTT}_q\left([(\mathbf{c}_{\text{mult}})_i]_w\right) \circ \overline{\mathsf{evk}}_i\right) \tag{1}$$

and store the evaluation keys evk_i in NTT form as $\overline{\mathsf{evk}}_i = \mathrm{NTT}_q(\mathsf{evk}_i)$ for $i \in [0, \ell_{w,q} - 1]$ (similar to [19, Algorithm 2]). To deal with the limited internal memory when computing the NTT we use the aforementioned cached-FFT algorithm [2,3]. This enables us to exploit the memory hierarchy on Catapult where we have access to fast but small FPGA-internal memory (≈4.9 MiB) and large but slow external DRAM (two times 4 GB). We also incorporate some of the optimizations to the NTT proposed in [36]. By merging the multiplication by powers of ψ into the twiddle factors of the main NTT computation we not only save n multiplications but also eliminate expensive read and write operations. To optimally utilize the burst read/write capabilities of the DRAM[1] we have designed our core in a way that we balance non-continuous reorderings and continuous reads or writes. While we only implemented two main parameter sets, our approach is scalable and could be extended to even larger parameter sets and is also generally applicable as we basically implement polynomial multiplication, which is common in most RLWE-based homomorphic encryption schemes.

The general architecture of our `HomomorphicCore` design is shown in Fig. 2. We have divided our implementation into a memory management unit (`NTTMemMgr`) and an NTT computation unit (`NttCore`). The `NTTMemMgr` component loads or stored groups while `NttCore` is responsible for the computation of the $\mathsf{C\text{-}NTT}$ on the cache. Both components have access to the memories

[1] The throughput of the DRAM is drastically increased if large continuous areas of the memory are read at once using the so called *burst mode*.

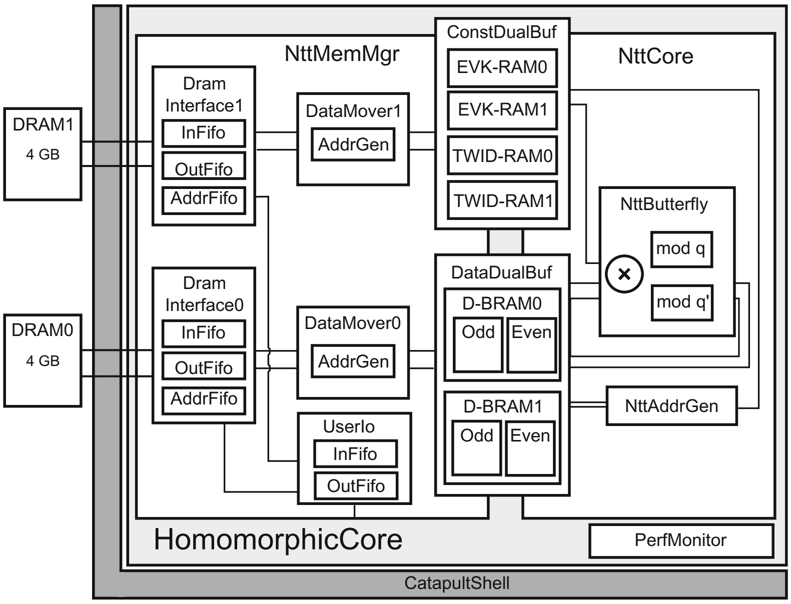

Fig. 2. Block diagram of our HomomorphicCore core used to implement YASHE. The design is controlled by a host server using the `CatapultShell` and has access to two 4 GB DDR3-1333 DRAMs.

`ConstDualBuf` and `DataDualBuf`. The `DataDualBuf` buffer contains a configurable number of groups of a polynomial and the `ConstDualBuf` buffer contains the constants (e.g., twiddle factors or evaluation keys) that correspond to the groups in `DataDualBuf`. To the `NttCore` it does not matter which subset of the cached-NTT has to be computed as this is only determined by the loaded data and twiddle factors. This makes the design simpler and also easier to test. To support moduli q and q' we implemented two butterfly units that share one large integer multiplier. Both buffers are double-buffered so that the `NttCore` component can compute on one subset of the data while the `NTTMemMgr` component can load or store a new subset from or into the other buffer. Ciphertexts, NTT constants, and keys are held in one of the two DRAMs (`Dram0` or `Dram1`) and are provided to the core from the outside over the `UserIo` and `CatapultShell` components. The `CatapultShell` component implements a simple PCI Express (PCIe) interface that allows the host server to issue commands (e.g., Add, or Mult) and to transfer data. Evaluated ciphertexts are also stored in the DRAM and can be read by the host after a computation is finished.

4 Hardware Architecture

In this section we describe our hardware architecture with an emphasis on the memory bandwidth-friendly cached-NTT polynomial multiplier.

4.1 Implementation of the Cached-NTT and Memory Addressing

A crucial aspect when implementing the cached-NTT is efficient access to the main memory (i.e., DRAM) and the use of burst transfers. In this section we describe how data is transferred between the main memory (`Dram0` and `Dram1`) and the cache memory (`DataDualBuf` and `ConstDualBuf`) and how these transfers are optimized.

General Idea. The cached-FFT has been designed for systems with a small cache that supports fast access to coefficients during the computation of a C-NTT on a group. For our core we do not have a transparent cache, like on a CPU, but implement the fast directly addressable internal on-chip memories `DataDualBuf` and `ConstDualBuf` using BRAMs. As we know exactly which values are required at which time, we explicitly load a group into the internal memory before and write it back after a C-NTT computation. The necessary reordering (see Fig. 1) is either performed before or after a computation on a group and done when reading from or writing data into the DRAM. As the DRAM is large enough, plenty of memory is available for temporary storage, but one epoch has to be computed completely and the reordering has to be finished before the next epoch can be computed. In general, it would be sufficient to just store one group consisting of $C = n/G$ coefficients in each buffer of `DataDualBuf`. However, we allow the storage and computation on K groups/caches (configurable as generic during synthesis) in `D-BRAM0` and `D-BRAM1` at the same time (when computing modulo q). One reason is that for relatively small groups we can then avoid frequent waiting for the pipeline to clear after a C-NTT has been computed. Additionally, storing of multiple groups allows more efficient usage of burst reads and writes.

For efficiency (due to less memory transfers) and simplicity we restrict our implementation to a cached-NTT with two epochs[2]. We thus support only dimensions $n = 2^{2n'}$ for $n' \in \mathbb{N}$. For Set I we use $(n = 4096, E = 2, G = 64, P = 6)$ and for Set II $(n = 16384, E = 2, G = 128, P = 7)$.

Supported Commands. To simplify the implementation of homomorphic evaluation algorithms (see Sect. 5) and to abstract away implementation details we support a specific set of instructions to store or load groups or constants and to compute the C-NTT on such stored groups. A complete set of available commands is provided in Table 2. These commands could also be used to implement other homomorphic schemes and they can be directly used to realize polynomial multiplication in $\mathbb{Z}_q[X]/(X^n + 1)$ and $\mathbb{Z}_{q'}[X]/(X^n + 1)$.

Each command consists of a name, which is mapped to an opcode, and zero, one, or two parameters that define the source or destination of data to be transferred or the buffer on which a computation should be performed. A command either blocks `Dram0`, `Dram1`, or `NttCore` and commands can be executed in parallel, in case no resource conflict happens. Memory transfer and computation

[2] With only one epoch the cached-NTT becomes the standard Cooley-Tukey NTT and the cache contains all n coefficients.

commands do not interfere due to the dual-buffering. Additionally, commands can be configured for specific situations. For commands operating on `Dram0` or `Dram1` the configuration describes how a storage operation should be performed. Supported modes are a continuous burst transfer ([burst]), or bit-reversal of coefficients ([bitrev]), and/or cached-NTT reordering ([ro]) during a write or read operation. The [q] and [q'] configuration determines whether transfers operate on polynomials modulo q or polynomials modulo q'. When a homomorphic operation has to be performed the top-level state machine also has to provide the base address of the inputs and the base address of the result memory block. Each command also supports a specific maximum burst transfer size.

The commands itself are described in Table 2. As an example, the load-group[burst]($\mathbf{t}$,x) command loads groups x to $x+K-1$ from the DRAM at base-address of $\mathbf{t}$ into a buffer using the DRAM's fast burst mode. The store-group[ro, bitrev]($\mathbf{t}$,x) command stores the groups x to $x+K-1$ in the DRAM at base-address of $\mathbf{t}$ but performs the reordering of the cached-NTT and also a bit-reversal. A command used to load constants is the load-twiddles[fwd,q](x, y) command that loads the twiddle factors required to compute groups x to $x+K-1$ in epoch y using burst mode. While the previous commands can be used to implement general polynomial multiplication, we also provide the YASHE specific load-group-expand, load-chunks, and store-chunks commands. The reason is that the KeySwitch algorithm requires the expansion of one polynomial into $\ell_{w,q}$ polynomials (from now on also referred to as *chunks*). For efficiency reasons, the computations are thus performed in parallel on all decomposed polynomials and the larger amount of data to be transferred is handled by the previously mentioned commands. The width of the data ports of `DataDualBuf` and `ConstDualBuf` is $\frac{q'}{2}$ bits so that we can either store one coefficient modulo q in one position or half of a coefficient modulo q'. As a consequence, the minimal size of `D-BRAM0` and `D-BRAM1` is $\frac{\lceil \log_2 q' \rceil \cdot K \cdot n \cdot \ell_{w,q}}{2G}$ bits.

Usage of Burst Transfers. A significant advantage of storing multiple groups is that this allows the usage of the DRAM's burst mode. In case memory is written or read continuously ([burst]) it is straightforward to see that $K \cdot C$ coefficients can be handled in one burst transfer. But also when performing the cached-NTT reordering ([ro]) the simultaneous reordering of multiple groups allows better utilization of burst operations[3]. By iterating over the groups and then over the addresses we can write K coefficients using burst mode and thus reduce memory transfer times significantly. Note that the non-continuous access to memory in `D-BRAM0` or `D-BRAM1` does not introduce a performance bottleneck as the memory is implemented using BRAMs that do not cause a performance penalty when being accessed non-continuously.

[3] In the following we only discuss the case of writing coefficients from the FPGA (BRAM) into the external memory (DRAM) in reordered or reordered and bit-reversed fashion. However, the same ideas can be also applied for loading from the DRAM and writing into the BRAM on the FPGA.

Table 2. Commands that are used to implement YASHE with `HomomorphicCore`. Depending on the configuration of each memory transfer command, different burst widths can be realized.

Command	Param. p_1	Param. p_2	Resource	Configuration
load-group-expand	DRAM address	group	`Dram0`	[burst]
Loads groups p_2 to $p_2 + K - 1$ using p_1 as base address, performs the decomposition $\mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}) = ([(\tilde{\mathbf{c}}_{\text{mult}})_i]_w)_{i=0}^{\ell_{w,q}-1}$ into $\ell_{w,q}$ polynomials, and stores the decomposed polynomials in the `DataDualBuf` buffer.				
store-chunks	DRAM address	group	`Dram0`	[burst,q], [burst,q']
Saves groups p_2 to p_2+K-1 of all $\ell_{w,q}$ decomposed polynomials ([q]) or spitted coefficients modulo q' ([q']) stored in `DataDualBuf` at base address p_1.				
load-chunks	DRAM address	group	`Dram0`	[burst,q'], [ro,q'] [ro,bitrev,q'], [ro,q]
Equivalent to store-chunks.				
store-group	DRAM address	group	`Dram0`	[burst], [ro,bitrev], [ro]
Saves groups p_2 to p_2+K-1 of the polynomial stored in `DataDualBuf` at base address p_1.				
load-group	DRAM address	group	`Dram0`	[burst], [bitrev]
Equivalent to store-group.				
load-twiddles	group G	epoch E	`Dram1`	[(fwd|inv),q], [(fwd|inv),q']
Loads the precomputed forward or inverse twiddle factors for modulus q or q' for groups p_1 to $p_1 + K$ and epoch $E = p_2$ into `ConstDualBuf` using burst read.				
load-psis	group G	-	`Dram1`	[q], [q']
Loads the powers of ψ^{-1} for groups p_1 to $p_1 + K - 1$ and moduli q or modulus q' from DRAM using burst read and saves them in `ConstDualBuf`.				
load-evks	DRAM address	group	`Dram1`	-
Loads the $\ell_{w,q}$ different evaluation key parts for groups p_2 to $p_2 + K - 1$ stored at base address p_1 into `ConstDualBuf` using burst read.				
ntt-on-buffer	chunk	-	`NttCore`	[q], [q']
Computes the C-NTT on chunk p_1 stored in `DataDualBuf` using either modulus q or modulus q' and requiring $\frac{Pn}{2G}$ multiply accumulate (MAC) operations.				
mul-psi	chunk	-	`NttCore`	[(q|q'),round]]
Multiplies chunk p_1 stored in `DataDualBuf` by powers of ψ^{-1} stored in `ConstDualBuf`. If configured with [round] the YASHE rounding operation is performed after the NTT.				
mul-evk	chunk	-	`NttCore`	[q]
Multiplies chunk p_1 in `DataDualBuf` by the evaluation keys stored in `ConstDualBuf`.				
accumulate	chunk	-	`NttCore`	-
Adds chunks p_1 to chunk 0 stored in `DataDualBuf`.				
mul-point-wise	-	-	`NttCore`	[q], [q']
Performs point-wise multiplication.				

Another improvement is achieved by the combination of the bit-reversal with the reordering procedure of the cached-NTT ([ro,bitrev]) in which case it is possible to write a whole group ($C = n/G$ coefficients) using burst mode. As a consequence, it is even preferable to compute the reordering together with the bit-reversal instead of only the reordering, as the size of the burst write is even larger in this case for relevant parameters (i.e., n/G instead of G).

4.2 Computation of the C-NTT on the Cache

The C-NTT is computed on each group in the cache (see the dotted box in Fig. 1) and requires arithmetic operations that dominate the area costs of our implementation. Each C-NTT on a group requires $\frac{Pn}{2G}$ multiplications in $\mathbb{Z}_q$ (or $\mathbb{Z}_{q'}$) and the whole cached-NTT requires $EG\frac{Pn}{2G} = \frac{n\log_2(n)}{2}$ multiplications in $\mathbb{Z}_q$ (or $\mathbb{Z}_{q'}$). The address generation in `NttCore`, which implements the C-NTT, is independent of the group or epoch that is processed. This allows a simple data-path and also testability independently of the memory transfer commands. To saturate the pipelined butterfly unit of the NTT, two reads and two writes are required per cycle and we use the well-known fact that the buffer can be split into two memories, one for even and one for odd addresses (see [32]). While this approach might lead to wasted space in block memories if small polynomials do not fill a whole block RAM, as in [33] and optimized in [1,36], it is not a concern for the large parameter sets we are dealing with. The only input to the NTT, besides the actual polynomial coefficients, that depend on the current group or epoch are the constants like twiddle factors, powers of ψ^{-1}, or the evaluation key evk. We decided to store each constant in a continuous memory region and load them into the `TWID-RAM` or `EVK-RAM` buffers depending on the current group or epoch. While it would also be possible to compute the twiddle factors on-the-fly (as in [36]) this approach would require an additional expensive $q' \times q'$ multiplier and modulo unit. Additionally, we do not exploit redundancies in twiddle factors or other tricks so that we are able to load constants using the fast burst mode. The only important observation is that when $E = 2$ the same set of twiddle factors is used for the computation of all groups of the first epoch of the NTT.

For best performance of the NTT_q our architecture requires a pipelined NTT butterfly that is able to compute a $\log_2(q) \times \log_2(q)$ multiplication, modular reduction, and two accumulations per cycle. For the butterfly of the $\text{NTT}_{q'}$, execution in one clock cycle is not necessary as the maximum data width of the `ConstDualBuf` and `DataDualBuf` components is $\frac{q'}{2}$. Thus, at least two cycles are needed to load a coefficient from the buffer in which one coefficient modulo q' is split into chunk 0 and chunk 1.

To instantiate the multiplier we used a traditional RTL design that uses four pipelined 72×72-bit multipliers generated using the Altera MegaWizard to instantiate a 144×144-bit multiplier. The instantiation of four 144×144-bit multipliers yields a 288×288-bit multiplier and finally a pipelined 576×576-bit multiplier. For modular reduction we restrict the moduli q and q' to Solinas primes [37] of the form $2^y - 2^z + 1$ for $y, z \in \mathbb{Z}$ and $y > z$. A modular reduction circuit can then be configured by providing the input bit width and the values y and z as generics/parameters. The implementation only requires a few shifts and few additions/subtractions to perform a modular reduction.

5 Configuration of Our Core for YASHE

For our prototype we have implemented YASHE's homomorphic evaluation operations Add and Mult using the architecture described in Sect. 4. As space is limited we only cover the RMult and KeySwitch functions in detail, which are essential for the implementation of Mult. All homomorphic evaluation operations use the hardware architecture described in Sect. 4 and the commands provided in Table 2. The commands are executed by a large state machine implemented in HomomorphicCore, which is also responsible for interaction with the Catapult shell and host PC.

5.1 Implementation of RMult

For RMult, a standard integer polynomial multiplication in $\mathbb{Z}_{q'}[X]/(X^n + 1)$ is required after which the result is rounded and reduced modulo q. Selecting $q' > nq^2$ guarantees that the product $\mathbf{c}_1\mathbf{c}_2$ of two polynomials $\mathbf{c}_1, \mathbf{c}_2 \in \mathbb{Z}_q[X]/(X^n+1)$ is computed over the integers and not being reduced before it is rounded. Instead of using a single routine for RMult, the host server can make separate calls to a single forward transformation $\bar{\mathbf{c}}_i = \mathsf{RMultFwd}(\mathbf{c}_i)$ so that polynomials to be multiplied with multiple other polynomials have to be transformed only once into the NTT domain. The $\tilde{\mathbf{c}}_{\text{mult}} = \mathsf{RMultInv}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2)$ routine then takes two transformed polynomials $\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2$ as input and computes the product by performing point-wise multiplication, the inverse NTT, and rounding of the result. While we give up some efficiency (e.g., merging of forward transformation and point-wise multiplication) by this approach, it seems beneficial to provide this additional flexibility when computing homomorphic circuits.

The (simplified) sequence of executed commands for RMultFwd is provided in Algorithm 1, but for the actual implementation load/store operations and NTT computations are executed in parallel to make use of the double-buffer capability of the DataDualBuf and ConstDualBuf components. In step 5 of RMultFwd the input polynomial is expected to be saved in bitreversed order already. This is either ensured by the user when the polynomial is initially transferred to the device or by our implementation in the last step of KeySwitch. The only execution of a reordering load operation is performed in step 11 and all other loads or stores use the burst mode. Thus the second reordering is delayed till the pointwise multiplication in RMultInv which is given in Algorithm 2. In RMultInv the first block of operations (steps 3 to 7) is responsible for the pointwise multiplication. Note that the Add operation of YASHE is basically this loop but mul-pointwise is exchanged by a command for addition in $\mathbb{Z}_q$. The first NTT-related load is performed in step 11 in which the final reordering of the forward transform together with the bitreversal step is performed. The final rounding operation $\left[\left\lfloor \frac{t}{q}\mathbf{t}_2 \right\rceil\right]_q$ is included into the mul-psi[q', round] command. After that the result $\tilde{\mathbf{c}}_{\text{mult}}$ is in $\mathbb{Z}_q[X]/(X^n+1)$. Note that it is not possible to merge the multiplication by powers of ψ^{-1} into the NTT twiddle factors for the inverse transformation [36] as we use the Cooley-Tukey butterfly. The multiplication by powers of ψ^{-1} is performed by the mul-psi command and the constants are loaded into the memory space reserved for the evaluation key during the forward transformation by load-psis. The multiplication by the scalar n^{-1} is merged into the ψ^{-1} values.

Algorithm 1. Forward transformation of an input polynomial in RMult

```
1: function RMultFwd($\mathbf{c}_i$)
2:     //Epoch 0
3:     load-twiddles[fwd,$q'$]$(0, 0)$
4:     forall groups $x \in 0 \ldots G/K - 1$:
5:         load-group[burst]($\mathbf{c}_i$, $Kx$)
6:         ntt-on-buffer[$q'$](0)
7:         store-chunks[burst,$q'$]($\mathbf{t}$, $Kx$)
8:     //Epoch 1
9:     forall groups $x \in 0 \ldots G/K - 1$:
10:        load-twiddles[fwd,$q'$]$(Kx, 1)$
11:        load-chunks[ro,$q'$]($\mathbf{t}$, $Kx$)
12:        ntt-on-buffer[$q'$](0)
13:        store-chunks[burst,$q'$]($\bar{\mathbf{c}}_i$, $Kx$)
14:    return $\bar{\mathbf{c}}_i$
15: end function
```

Algorithm 2. Pointwise multiplication and inv. transformation in RMult

```
1: function RMultInv($\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2$)
2:     //Pointwise multiplication
3:     forall groups $x \in 0 \ldots G/K - 1$:
4:         load-chunks[burst,$q'$]($\mathbf{c}_1$, $Kx$)
5:         load-chunks[burst,$q'$]($\mathbf{c}_2$, $Kx$)
6:         mul-point-wise[$q'$]()
7:         store-chunks[burst,$q'$]($\mathbf{t}_1$, $Kx$)
8:     //Epoch 0
9:     load-twiddles[inv,$q'$]$(0, 0)$
10:    forall groups $x \in 0 \ldots G/K - 1$:
11:        load-chunks[ro,bitrev,$q'$]($\mathbf{t}_1$, $Kx$)
12:        ntt-on-buffer[$q'$](0)
13:        store-chunks[burst,$q'$]($\mathbf{t}_2$, $Kx$)
14:    //Epoch 1
15:    forall groups $x \in 0 \ldots G/K - 1$:
16:        load-twiddles[inv,$q'$]$(Kx, 1)$
17:        load-psis[$q'$]$(Kx)$
18:        load-chunks[ro,$q'$]($\mathbf{t}_2$, $Kx$)
19:        ntt-on-buffer[$q'$](0)
20:        mul-psi[$q'$, round](0)
21:        store-group[ro,bitrev]($\tilde{\mathbf{c}}_{\text{mult}}$, $Kx$)
22:    return $\tilde{\mathbf{c}}_{\text{mult}}$
23: end function
```

5.2 Implementation of KeySwitch

The control-flow used to implement KeySwitch based on the commands introduced in Sect. 4 and Eq. 1 is given in Algorithm 3. For the forward transformation (step 2 to step 19) the coefficients of the input polynomial $\tilde{\mathbf{c}}_{\text{mult}}$ can be loaded using the burst mode as they have already been stored in bitreversed representation in RMultInv. The decomposition $\mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}) = ([(\tilde{\mathbf{c}}_{\text{mult}})_i]_w)_{i=0}^{\ell_{w,q}-1}$ is performed on-the-fly inside the FPGA using the load-group-expand[burst] command. The NTT is then performed on all $\ell_{w,q}$ decomposed polynomials in the buffer. As the twiddle factors are the same for each polynomial we only have to load and store K sets of twiddle factors into the `ConstDualBuf` component (each set containing $P \cdot \ell_{w,q}/2$ coefficients). During the NTT computation on all polynomials the results are accumulated (step 18) and then stored (step 19). The relatively slow reordering operation load-chunks[ro, q] is performed at the beginning of the second epoch and not after the first epoch as the accumulation and multiplication with the evaluation keys takes additional time so that we can balance the time required for memory transfers and computation. As the forward transformed polynomials are already stored in the correct order, we just have to perform a burst read at the beginning of the inverse transformation in step 24. Additionally, the computation is much less involved as we only have to compute one INTT_q and not $\ell_{w,q}$ computations of NTT_q caused by the decomposition.

Algorithm 3. Key switching in YASHE

```
 1: function KeySwitch($\tilde{\mathbf{c}}_{\mathrm{mult}}, \bar{\mathsf{evk}}$)
 2:     //Fwd. transform and accumulation:
 3:     load-twiddles[fwd,$q$]$(0, 0)$
 4:     //Epoch 0
 5:     forall groups $x \in 0 \ldots G/K - 1$:
 6:         load-group-expand[burst]($\tilde{\mathbf{c}}_{\mathrm{mult}}, Kx$)
 7:         forall chunks $y \in 0 \ldots \ell_{w,q} - 1$:
 8:             ntt-on-buffer[$q$]($y$)
 9:         store-chunks[burst,$q$]($\mathbf{t}_1, Kx$)
10:     //Epoch 1
11:     forall groups $x \in 0 \ldots G/K - 1$:
12:         load-twiddles[fwd,$q$]($Kx, 1$)
13:         load-evk($\bar{\mathsf{evk}}, Kx$)
14:         load-chunks[ro, $q$]($\mathbf{t}_1, Kx$)
15:         forall chunks $y \in 0 \ldots \ell_{w,q}$:
16:             ntt-on-buffer[$q$]($y$)
17:             mul-evk[$q$]($y$)
18:             accumulate($y$)
19:         store-group[ro, bitrev]($\mathbf{t}_2, Kx$)
20:     //Inverse transform:
21:     load-twiddles[inv,$q$]$(0, 0)$
22:     //Epoch 0
23:     forall groups $x \in 0 \ldots G/K - 1$:
24:         load-group[burst]($\mathbf{t}_2, Kx$)
25:         ntt-on-buffer[$q$](0)
26:         store-group[ro]($\mathbf{t}_1, Kx$)
27:     //Epoch 1
28:     forall groups $x \in 0 \ldots G/K - 1$:
29:         load-twiddles[inv,$q$]($Kx, 1$)
30:         load-psis[$q$]($Kx$)
31:         load-group[burst]($\mathbf{t}_1, Kx$)
32:         ntt-on-buffer[$q$](0)
33:         mul-psi[$q$](0)
34:         store-group[ro, bitrev]($\mathbf{c}_{\mathrm{mult}}, Kx$)
35:     return $\mathbf{c}_{\mathrm{mult}}$
36: end function
```

6 Results and Comparison

In this section we provide post place-and-route (post-PAR) results and performance measurements of our implementation on the Catapult board [34] equipped with an Altera Stratix V (5GSND5H) FPGA and two 4 GB DRAMs.

6.1 Resource Consumption and Performance

The resource consumption of our implementation is reported in Table 3. Achieving a high clock frequency for parameter Set II is challenging. One reason seems to be that, due to our design choices, we have to deal with extremely large structures like several thousand bit wide adders and a large integer multiplier. Such structures are tedious to manually optimize and it is hard to determine an optimal pipeline length. Another reason is that the design is congested and that placement and fitting have to satisfy strict constraints imposed by the PCIe and DRAM controllers in the Catapult shell. Still, switching to larger devices to reduce congestion would also increase costs.

Table 3. Resource consumption of our implementation (including communication).

Implementation	ALM	FF	DSP	BRAM Bits	MHz
Set I (n=4096, K= 8)	69,058 (40 %)	144,747	144 (9 %)	8,031,568 (19 %)	100
Set II (n=16384, K=4)	141,090 (82 %)	391,773	577 (36 %)	17,626,400 (43 %)	66

Table 4. Cycle counts and runtimes for the different evaluation algorithms of YASHE measured on the Catapult board.

Implementation		Mult	Add	KeySwitch	RMult	RMultFwd	RMultInv
Set I (n=4096) 100 MHz (K=8)	cycles	675,326	19,057	478,911	196,415	160,693	157,525
	time	6.75 ms	0.19 ms	4.79 ms	1.96 ms	1.61 ms	1.58 ms
Set II (n=16384) 66 MHz (K=4)	cycles	3,212,506	61,775	1,372,519	1,839,987	587,664	664,659
	time	48.67 ms	0.94 ms	20.80 ms	27.88 ms	8.90 ms	10.07 ms

Cycle counts for evaluation operations are given in Table 4 and are obtained using the `PerfMonitor` component that logs cycle counts and transfers them to the host server over PCIe, if requested. The usual approach of obtaining cycle counts from simulation is not possible as we are using an external DRAM without a cycle accurate simulation model. Note that the Mult operation requires to execute RMult and KeySwitch. Also note that the runtime does not simply scale for higher clock frequencies as the DDR memory interface is running in its own clock domain and thus the memory bandwidth is not significantly increased by higher clock frequencies of the `HomomorphicCore` component.

A good indicator for the efficiency of our memory addressing is the saturation of the $\log(q) \times \log(q)$ modular multiplier. One NTT requires $\frac{n}{2}\log_2(n)$ multiply-accumulate (MAC) operations so that KeySwitch takes at least $C_{KS}(\ell_{w,q}, n) = (\ell_{w,q} + 1)(\frac{n}{2}\log_2(n) + n)$ cycles assuming one clock cycle per MAC ($\ell_{w,q}$ forward and one inverse NTT, see Eq. 1). For parameter Set II we get $C_{KS}(8{,}16384) = 1{,}179{,}648$ as lower bound on the number of cycles for KeySwitch which is close to the measured 1,372,519 cycles. For RMult approx. $C_{\mathrm{RM}}(n) = 3(4\frac{n}{2}\log_2 n) + 2(4n)$ cycles are required (three transformations, pointwise and ψ^{-1} multiplication; four cycles per MAC) and the saturation of the MAC unit is $\frac{C_{\mathrm{RM}}(16384)}{1{,}839{,}987} = 0.82$.

6.2 Comparison with Previous Work

Cao et al. [9] describe an implementation of the integer-based FHE scheme in [16] on a Virtex-7 FPGA (XC7VX980T) but explicitly do not take into account the bottleneck that may be caused by accessing off-chip memory. Their implementation achieves a speed up factor of 11.25 compared to a software implementation but for large parameter sets, which might promise some performance gains, the design does not fit on current FPGAs. An FPGA implementation of an integer multiplier for the Gentry-Halevi [24] FHE scheme is proposed in [43]. The architecture requires about 462,983 ALUs, and 720 DSPs on a Stratix-V

(55GSMD8N3F45I4) and allows 768K-bit multiplications. It is reported to be about two times faster than a similar implementation on an NVIDA C2050 GPU. Another 768K-bit multiplication architecture is proposed by Wang et al. in [44] targeting ASICs and FPGAs. An outline of an implementation of a homomorphic encryption scheme is given in [17] using Matlab/Simulink and the Mathwork HDL coder. The used tools limit the available basic multiplier width to 128 bits and the design requires multiple FPGAs to deal with long vectors.

An ASIC implementation of a million-bit multiplier for integer-based FHE schemes is presented by Doröz et al. in [21]. The computation of the product of two 1,179,648-bit integers takes 5.16 million clock cycles. Synthesis results for a chip using the TSMC 90 nm cell library show a maximum clock frequency of 666 MHz and thus a runtime of 7.74 ms for this operation, equivalent to that of a software implementation. This shows, similar to our result, that the biggest challenges in the implementation of homomorphic cryptography in hardware are the large ciphertext sizes that do not fit into block RAMs (our case) or caches instantiated with the standard library (Doröz et al. [21]).

Wang et al. [42] present the first GPU implementation of an FHE scheme and provide results for the Gentry-Halevi [24] scheme on an NVIDIA C2050 GPU. The results were subsequently improved in [40]. A GPU implementation of the leveled FHE scheme by Brakerski et al. [8] is given in [41]. In [19] Dai et al. provide an implementation of the DHS [20] FHE scheme that is based on the scheme in [30]. For the parameters ($n = 16384, \log_2(q) = 575$), they require 0.063 s for multiplication and 0.89 s for relinearization (key switching) on a 2.5 GHz Xeon E5-2609 equipped with an NVIDIA GeForce GTX 690.

A software library that implements the Brakerski-Gentry-Vaikuntanathan (BGV) [7,8] scheme is described in [27]. In [29], a software implementation of YASHE is reported which for the parameter set ($n = 4096, q = 2^{127} - 1, w = 2^{32}$) executes Add in 0.7 ms, RMult in 18 ms, and KeySwitch in 31 ms on an Intel Core i7-2600 running at 3.4 GHz. So our hardware implementation can evaluate Mult on a parameter set supporting 9 levels in 48.67 ms while the software requires 49 ms for parameters supporting only 1 multiplicative level.

Roy et al. [35] proposed an implementation of YASHE with $n = 2^{15}$ and a modulus of $\log_2(q) = 1228$ bits. They use a much larger next generation FPGA (Virtex-7 XC7V1140T) from a different vendor so that a comparison with our work is naturally hard - especially regarding the economical benefits of using FPGAs. We see the biggest contribution of the work by Roy et al. in their efficient implementation of independent processors that use the CRT to decompose polynomials. This approach avoids large integer multipliers and simplifies routing and performance tuning. When we designed our core, the added complexity and the need to lift polynomials from CRT to natural representations in hardware appeared to be too expensive. However, the authors of [35] do not consider the costs of moving data between external memory and the FPGA but just assume unlimited memory bandwidth. This naturally simplifies the design and placement but does not appear to be a realistic assumption. In our work a considerable amount of time was spent to implement efficient memory transfers and to optimize the algorithms in this regard. However, we see our work and the work of Roy et al. as a first step towards an efficient accelerator.

7 Future Work

While implementing the scheme we encountered several challenges that might also be a good start for future work. A big issue was verification and simulation time due to the large problem sizes. Different design or simulation approaches are probably needed for larger parameter sets. Another area of future work is the design of a more efficient and easier to synthesize large-integer modular multiplier and further design space exploration and implementation of larger parameter sets. Additionally, it might also make sense to investigate the applicability of the Chinese remainder theorem (CRT) in combination with the cached-NTT.

References

1. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, 2–3 June 2013, pp. 81–86. IEEE Computer Society (2013). 5, 12
2. Baas, B.M.: An approach to low-power, high performance, fast fourier transform processor design. Ph.D. thesis, Stanford University, Stanford, CA, USA (1999). 2, 5, 7
3. Baas, B.M.: A generalized cached-FFT algorithm. In: 2005 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2005, Philadelphia, Pennsylvania, USA, 18–23 March 2005, pp. 89–92. IEEE (2005). 2, 5, 6, 7
4. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) IMACC 2013. LNCS, vol. 8308, pp. 45–64. Springer, Heidelberg (2013). 2, 3, 4
5. Bos, J.W., Lauter, K.E., Naehrig, M.: Private predictive analysis on encrypted medical data. J. Biomed. Inform. **50**, 234–243 (2014). 1
6. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012). 2
7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. IACR Cryptology ePrint Archive, 2011:277 (2011). 18
8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, 8–10 January 2012, pp. 309–325. ACM (2012). 2, 18
9. Cao, X., Moore, C., O'Neill, M., Hanley, N., O'Sullivan, E.: High-speed fully homomorphic encryption over the integers. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. LNCS, vol. 8438, pp. 169–180. Springer, Heidelberg (2014). Extended version: [10]. 2, 17, 19
10. Cao, X., Moore, C., O'Neill, M., O'Sullivan, E., Hanley, N.: Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction. IACR Cryptology ePrint Archive, 2013:616 (2013). Conference version of [9]. 19

11. Chen, D.D., Mentens, N., Vercauteren, F., Sinha Roy, S., Cheung, R.C.C., Pao, D., Verbauwhede, I.: High-speed polynomial multiplication architecture for Ring-LWE and SHE cryptosystems. IACR Cryptology ePrint Archive, 2014:646 (2014). 2
12. Cheon, J.H., Kim, M., Kim, M.: Search-and-compute on encrypted data. Cryptology ePrint Archive, Report 2014/812 (2014). http://eprint.iacr.org/2014/812. 1
13. Cheon, J.H., Kim, M., Lauter, K.: Homomorphic computation of edit distance. Cryptology ePrint Archive, Report 2015/132 (2015). http://eprint.iacr.org/2015/132. 1
14. Chu, E., George, A.: Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms. CRC Press, Boca Raton (2000). 4, 5
15. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965). 5
16. Coron, J.-S., Mandal, A., Naccache, D., Tibouchi, M.: Fully homomorphic encryption over the integers with shorter public keys. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 487–504. Springer, Heidelberg (2011). 2, 17
17. Cousins, D., Rohloff, K., Peikert, C., Schantz, R.E.: An update on SIPHER (scalable implementation of primitives for homomorphic encryption) - FPGA implementation using simulink. In: IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, 10–12 September 2012, pp. 1–5. IEEE (2012). 17
18. Crandall, R., Pomerance, C.: Prime Numbers: A Computational Perspective. Springer, New York (2001). 4
19. Dai, W., Doröz, Y., Sunar, B.: Accelerating NTRU based homomorphic encryption using GPUs. IACR Cryptology ePrint Archive, 2014:389 (2014). To appear in IEEE Transaction on Computers. 2, 7, 18
20. Doröz, Y., Yin, H., Sunar, B.: Homomorphic AES evaluation using NTRU. IACR Cryptology ePrint Archive, 2014:39 (2014). 18
21. Doröz, Y., Öztürk, E., Sunar, B.: Evaluating the hardware performance of a million-bit multiplier. In: 2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, 4–6 September 2013, pp. 955–962. IEEE Computer Society (2013). 2, 17
22. Gentleman, W.M., Sande, G.: Fast fourier transforms: for fun and profit. In: American Federation of Information Processing Societies: Proceedings of the AFIPS 1966 Fall Joint Computer Conference, 7–10 November 1966, San Francisco, California, USA. AFIPS Conference Proceedings, vol. 29, pp. 563–578. AFIPS/ACM/Spartan Books, Washington D.C. (1966). 5
23. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009, pp. 169–178. ACM (2009). 2
24. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011). 2, 17, 18
25. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012). 2
26. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 75–92. Springer, Heidelberg (2013). 2

27. Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 554–571. Springer, Heidelberg (2014). https://shaih.github.io/HElib/. 18
28. Lauter, K.E., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Cachin, C., Ristenpart, T. (eds.) Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, 21 October 2011, pp. 113–124. ACM (2011). 1
29. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes FV and YASHE. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 318–335. Springer, Heidelberg (2014). 2, 3, 4, 18
30. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Karloff, H.J., Pitassi, T. (eds.) Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, 19–22 May 2012, pp. 1219–1234. ACM (2012). 2, 3, 18
31. Nussbaumer, H.J.: Fast Fourier Transform and Convolution Algorithms. Springer Series in Information Sciences, vol. 2. Springer, Berlin (1982). 4
32. Pease, M.C.: An adaptation of the fast Fourier transform for parallel processing. J. ACM **15**(2), 252–264 (1968). 12
33. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 139–158. Springer, Heidelberg (2012). 5, 12
34. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J.R., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D.: A reconfigurable fabric for accelerating large-scale datacenter services. In: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, 14–18 June 2014, pp. 13–24. IEEE Computer Society (2014). 2, 5, 16
35. Sinha Roy, S., Järvinen, K., Vercauteren, F., Dimitrov, V.S., Verbauwhede, I.: Modular hardware architecture for somewhat homomorphic function evaluation. IACR Cryptology ePrint Archive, 2015:337 (2015). To appear in Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp, xx–yy. Springer, Heidelberg (2015). 2, 18
36. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-LWE cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 371–391. Springer, Heidelberg (2014). 5, 7, 12, 14
37. Solinas, J.A.: Generalized Mersenne numbers. Technical Report MCORR 99–39, Faculty of Mathematics, University of Waterloo (1999). 13
38. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: Paterson, Kenneth G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 27–47. Springer, Heidelberg (2011). 3
39. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010). 2
40. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Exploring the feasibility of fully homomorphic encryption. IEEE Trans. Comput. **64**(3), 698–706 (2015). 2, 18
41. Wang, W., Chen, Z., Huang, X.: Accelerating leveled fully homomorphic encryption using GPU. In: IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, 1–5 June 2014, pp. 2800–2803. IEEE (2014). 2, 18

42. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. In: IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, 10–12 September 2012, pp. 1–5. IEEE (2012). 18
43. Wang, W., Huang, X.: FPGA implementation of a large-number multiplier for fully homomorphic encryption. In: 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, 19–23 May 2013, pp. 2589–2592. IEEE (2013). 17
44. Wang, W., Huang, X., Emmart, N., Weems, C.C.: VLSI design of a large-number multiplier for fully homomorphic encryption. IEEE Trans. VLSI Syst. **22**(9), 1879–1887 (2014). 2, 17
45. Winkler, F.: Polynomial Algorithms in Computer Algebra. Texts and Monographs in Symbolic Computation, 1st edn. Springer, Wien (1996). 4