Faster Number Theoretic Transform on Graphics Processors for Ring Learning with Errors Based Cryptography

Ahmad Al Badawi
Department of Electrical and Computer Engineering
National University of Singapore
Singapore, 117583
Email: ahmad@u.nus.edu

Bharadwaj Veeravalli
Department of Electrical and Computer Engineering
National University of Singapore
Singapore, 117583
Email: elebv@nus.edu.sg

Khin Mi Mi Aung
Data Storage Institute
A * Star
Singapore, 138634
Email: Mi_Mi_AUNG@dsi.a-star.edu.sg

***Abstract*—The Number Theoretic Transform (NTT) has been revived recently by the advent of the Ring-Learning with Errors (Ring-LWE) Homomorphic Encryption (HE) schemes. In these schemes, the NTT is used to calculate the product of high degree polynomials with multi-precision coefficients in quasilinear time. This is known as the most time-consuming operation in Ring–based HE schemes. Therefore; accelerating NTT is key to realize efficient implementations. As such, in its current version, a fast NTT implementation is included in cuHE, which is a publicly available HE library in Compute Unified Device Architecture (CUDA). We analyzed cuHE NTT kernels and found out that they suffer from two performance pitfalls: shared memory conflicts and thread divergence. We show that by using a set of CUDA tailored-made optimizations, we can improve on the speed of cuHE NTT computation by 20%-50% for different problem sizes.**

***Index Terms*—Lattice Cryptography; Ring-LWE; Homomorphic Encryption; GPGPU Implementation; CUDA**

I. Introduction

Homomorphic Encryption (HE) schemes are taking the cryptography community by storm, due to their incredible capabilities. A HE scheme is essentially a cryptosystem that preserves the algebraic structure of data space in its plain and encrypted forms. In other words, HE allows one to induce purposive operations on data by only manipulating their ciphertexts without decryption[1] at any stage of computation [13, 9]. Furthermore, with Fully HE (FHE) schemes, one can theoretically compute arbitrary computable functions on encrypted data. What makes HE more attractive is that homomorphic evaluations do not need the decryption key, in its plain form, at any stage of computation. Moreover, they produce encrypted results that only the decryption key owner can decrypt. These features qualify HE to be a reliable solution for privacy-preserving computation problems.

At the higher level, HE schemes provide the user with five key primitives: key generation, encryption, decryption, homomorphic addition, and homomorphic multiplication. The last two primitives are similar to the AND and XOR Boolean gates. They allow the user to evaluate functions, also known as circuits, whose inputs are encrypted bits producing encrypted output. Since {AND, XOR} are Turing complete, they allow creating any computable function. This means that the user utilizes HE primitives to emulate a homomorphic processor whose inputs and outputs are encrypted bits. Although, this seems trivial in principle, in practical and secure situation, ciphertexts are extremely large and require large amount of processing and storage [11].

The literature includes a number of HE implementations [10, 3, 4, 6]. The list includes ideal lattices, integer, Learning with Errors (LWE), and Ring LWE (RLWE) based schemes. The notable schemes are based on (RLWE) as they provide special tools for parallel homomorphic evaluation [8] such as plaintext packing. Although they are theoretically efficient, i.e. they have polynomial running time, they are normally considered impractical.

A great amount of research has been put to improve HE performance. One viable approach is to accept the current status quo and accelerate HE computation. This can be done by offloading time-consuming operations to hardware-accelerators such as Graphics Processing Units (GPUs) [18, 4, 17], Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) [5, 12, 2]. These accelerators are known for their massive parallel execution paradigm which can be useful in harnessing the inherent parallelism existing in HE computation.

In this work, we focus on GPU implementation of HE via CUDA. More precisely, we review and analyze the performance of cuHE [4], a publicly available fast CUDA library for polynomial arithmetic. The core component of cuHE is a Number Theoretic Transform (NTT)-based polynomial multiplier. cuHE multiplier implements the Schnhage-Strassen's algorithm [14] using Emmart and Weems GPU data-path [7]. More precisely, cuHE supports polynomial arithmetic in the ring $R_q : \mathbb{Z}_q[x]/\Phi_m(x)$, where $q \in \mathbb{Z}$ and $\Phi_m(x)$ is the m-th cyclotomic polynomial. Polynomials of degree less than

[1]In fact, decryption may be invoked homomorphically using an encrypted decryption key in a procedure known as bootstrapping.

978-1-5386-4522-2/18/$31.00

$n = 2^{15}$ and coefficient size $|q| \approx 2575$ bits are supported by the library. cuHE employs a number of modular algorithms such as the Chinese Remainder Theorem (CRT) to handle high-precision coefficients. After analyzing the NTT kernels in cuHE, we found that they suffer from two main problems: 1) shared memory bank conflicts and 2) thread divergence. In this work, we show the source of these problems and how to eliminate them resulting in 20%-50% improvement for different problem sizes.

Precisely, the main contributions and scope of the paper can be summarized as follows:

- We review and analyze the performance of cuHE NTT kernels pointing out the sources of two key performance limitations.
- We propose solutions to eliminate those limitations.
- We provide a set of experiments to evaluate our solutions and quantify the improvement our optimizations provide.

The rest of the paper is organized as follows: in Section II we introduce briefly the general CUDA programming paradigm. Next, in Section III we introduce some mathematical background. In Section IV we review cuHE NTT implementations and point out performance limitations. We also provide full details of our proposed solution to eliminate those limitations. Our experiments and comparison results are presented in Section V. Finally, Section VI concludes the work.

II. CUDA-Capable GPU Architecture and Programming Model

A GPU is essentially a many-core processor ideally suitable for parallel applications. These cores are grouped into an array of highly threaded Streaming Multiprocessors (SMs). Each SM includes a set of cores that share computational resources such as registers, cache memory, and control logic. A GPU is also shipped with a DRAM global memory separate from the system memory. GPU execution model is as follows: 1) CPU transfers data to GPU memory, 2) CPU launches a kernel (GPU code) specified with the number of threads to execute that kernel, 3) finally; once the kernel is executed, CPU transfers the results from GPU memory to its local memory.

The launched threads are organized in several structures. At the highest level, threads are organized in 3 dimensional structures called blocks. These blocks are themselves grouped in 3 dimensional grids. Specific kernel launch parameters determine the shape and number of threads in those structures. At the lowest level, a low number of threads are grouped together in what so called a warp. Currently, warps include 32 threads. A warp can be viewed as an atomic execution unit, meaning that all threads in a warp start and end execution together. This assembling imposes a restricted execution model whose ideal case is branch-free code. When the kernel includes branch instructions, threads of a warp may take different directions. CUDA eliminates small branches by means of predicated execution. However, with long or nested branches, warp threads are serialized reducing thread-level parallelism. Hence, CUDA programmers need to avoid or reduce the amount of branches in kernels.

GPUs include different types of memories that vary in speed, and recommended access pattern. Global memory is the slowest as it is off-chip. Recommended access pattern for global memory is coalescent access, i.e. consecutive threads read/write consecutive words. Non-coalescent memory access amplifies the number of transactions to accommodate a memory request for a warp. Hence, GPUs include on-chip fast memory known as shared memory. A common CUDA programming style is to load data from global memory to shared memory, operate on data in shared memory, and then write it back to global memory. Shared memory can also be used to facilitate communication among threads in a thread block. As shown in Figure 1, shared memory is divided into equally-sized banks, 32 in current GPUs. These memory banks can be accessed simultaneously. An n-way bank conflict occurs when $n \geq 2$ threads in a warp access different words in the same bank. To resolve an n-way conflict, the hardware issues n load/store instructions. The ideal case is when each thread within a warp accesses a different word in any memory bank, since one memory transaction would be sufficient to accommodate the request. CUDA programmers need to avoid or reduce the amount of conflicts for optimum performance.

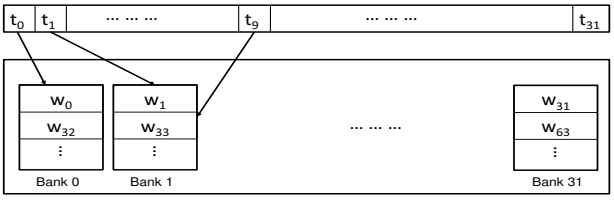

Fig. 1: A not necessarily realistic diagram of CUDA shared memory with 4-byte access mode showing 2-way bank conflict.

This will suffice to understand the optimizations proposed in this work. For a complete reference on CUDA programming, the reader is referred to [19].

III. Mathematical Background

In this section, we review the NTT definition, and how it can be used to multiply polynomials. We start by introducing the notations that are used hereafter. Next, we introduce the NTT multiplication algorithm.

A. Notation

Polynomials in coefficient representation are denoted by small letters. We capitalize their symbols to denote their NTT representation. $\mathbb{Z}_p$ is the set of integers modulo prime p represented by the set $\{0, 1, \ldots, p-1\}$. We use R_p to refer to the polynomial ring $\mathbb{Z}_p[x]/(\Phi_m(x))$, where m is a positive integer.

B. NTT-based Multiplication

NTT is a generalization of the Discrete Fourier Transform (DFT) to a finite field $GF(p)$. The NTT of polynomial a with degree $deg(a) < n$ is essentially a set of evaluations of a

on positive powers of an n-th primitive root of unity ω_n in $GF(p)$. Formally, the n-point NTT of polynomial a in $GF(p)$ is defined as:

$$A_k = \text{NTT}_{\omega_n}(a_k) = \sum_{i=0}^{n-1} a_i w^{ki} \pmod p \quad (1)$$

where $k = 0, 1, \ldots, n-1$.

Its inverse (NTT^{-1}) is defined as:

$$a_i = \text{NTT}^{-1}_{\omega_n}(A_i) = n^{-1} \sum_{k=0}^{n-1} A_k w^{-ik} \pmod p \quad (2)$$

Computing NTT or NTT^{-1} by simply applying Equations (1, 2) is of $\mathcal{O}(n^2)$ complexity. Nevertheless, a Fast Fourier Transform (FFT) data-path can be adapted to calculate either the NTT or NTT^{-1} in $\mathcal{O}(n \log n)$. This requires that additions and multiplications are performed in $GF(p)$ instead of $\mathbb{C}$.

The NTT and NTT^{-1} can be used to mutiply polynomials efficiently. Given the polynomials a and b of degree $< n$, their product c of degree $< 2n-1$, can be calculated as:

$$c = \text{NTT}^{-1}_{\omega_{2n}}(\text{NTT}_{\omega_{2n}}(a) \cdot \text{NTT}_{\omega_{2n}}(b)) \quad (3)$$

where $\cdot$ is a point-wise multiplication.

There are many guidelines on how to find NTT parameters (p, w) which are beyond the scope of this work. For further details, the reader is referred to [16].

IV. NTT Computation in cuHE

cuHE employs Emmart and Weems GPU data-path [7] to compute the NTT and NTT^{-1}. Emmart and Weems's multiplier uses the standard four-step Cooley-Tukey algorithm [1] shown in Algorithm 1. They provide an efficient memory layout to compute the NTT for sizes of the form $N = 8xy$, where $1 \leq x \leq 64$ and $y = 64^k$. They also avoid the transpose steps by treating the input array to the NTT^{-1} as $n \times m$ 2D matrix.

Algorithm 1 Cooley-Tukey four-step NTT

Given an array of coefficients a of size $N = m * n$. Treat a as a 2D $m \times n$ matrix assuming row-major order.

1: perform n m-point NTT_{ω_m} over the columns of a
2: multiply each element a_{ij} in the array with twiddle factor $\omega^{\pm ij}$ ▷ $+$ for NTT and $-$ for NTT^{-1}
3: perform m n-point NTT_{ω_n} over the rows of a
4: transpose a to $n \times m$

Emmart and Weems compute the NTT in a fixed $GF(p)$ where $p = 2^{64} - 2^{32} + 1$ is a special 64-bit Solinas prime [15]. The reasons for that are as follows:

1) p is 64-bit integer. CUDA has an ample support for 64-bit data types.
2) p provides a fast algorithm for reducing a 128-bit integer by using only 32-bit addition, subtraction and shift operations.
3) 8 is a 64-th primitive root of unity in $GF(p)$. Thus, 64-point NTTs can be computed by shifts rather than 64-bit multiplication.

Performance of NTT on GPU can depend heavily on the design of memory layout. Emmart and Weems used Bailey's algorithm [1] to design the memory layout of NTT for different sizes. In short, N-point NTT computation requires 3 kernels. For each kernel, the computation is distributed on $\frac{N}{512}$ thread blocks. Each block contains 64 threads so that a single thread handles 8 points. Threads load their points from global memory to shared memory. Shared memory is used to facilitate communication between threads within a block. Each thread loads its points from the shared memory to local registers, performs its part of the computation and writes back the result to the shared or global memory before exiting the kernel.

```
__global__ void ntt_1_16k_ext(uint64 *dst, uint32 *src) {
 __shared__ uint64 buffer[512];
 __shared__ uint64 roots[128];
 register uint64 samples[8];
 register uint32 fmem, tmem, fbuf, tbuf; // from/to mem/buffer

 //coalesced GMEM access & minimum SMEM bank conflicts
 fmem = ((tidx&0x38)<<5)|(bidx<<3)|(tidx&0x7);
 tbuf = ((tidx&0x38)<<3)|(tidx&0x7);
 fbuf = tidx;
 tmem = (bidx<<9)|((tidx&0x7)>>2<<8)|(tidx>>3<<2)|(tidx&0x3);
 roots[tidx] = tex1Dfetch(tex_roots_16k, (((bidx*2)*tidx)<<3)+1);
 roots[tidx] <<= 32;
 roots[tidx] += tex1Dfetch(tex_roots_16k, ((bidx*2)*tidx)<<3);
 roots[tidx+64] = tex1Dfetch(tex_roots_16k, (((bidx*2+1)*tidx)<<3)+1);
 roots[tidx+64] <<= 32;
 roots[tidx+64] += tex1Dfetch(tex_roots_16k, ((bidx*2+1)*tidx)<<3);

 //load 4 or 8 samples from GMEM, compute 8-sample ntt
 #pragma unroll
 for (int i=0; i<4; i++)
  samples[i] = (src+(bidy<<14))[(i<<11)|fmem];
 _ntt8_ext(samples);

 //times twiddle factors of 64 samples, store to buffer
 #pragma unroll
 for(int i=0; i<8; i++)
  buffer[(i<<3)|tbuf] = _ls_modP(samples[i], (tidx>>3)*i*3);
 __syncthreads();

 //load 8 samples from SMEM in transposed way, compute 8-sample ntt
 #pragma unroll
 for(int i=0; i<8; i++)
  samples[i] = buffer[(i<<6)|fbuf];
 _ntt8(samples);

 #pragma unroll
 for(int i=0; i<8; i++)
  (dst+(bidy<<14))[tmem|(i<<5)] = _mul_modP(samples[i],
   roots[((tidx&0x7)>>2<<6)|(tidx>>3)|(i<<3)]);
}
```

Listing 1: cuHE 1st kernel to compute the NTT of 16384 points. Each thread handles 8 points/samples. The shared memory `buffer` *is used to facilitate communication among threads. The twiddle factors are loaded from 1D texture tex_roots_16k and stored to shared memory. Each thread writes back the results of partial NTTs to either shared memory or global memory at the end of the kernel.*

Listing 1 shows the 1st kernel used in cuHE to compute the NTT of 16384 points. While loading and storing the points from and to global memory are coalescent, the kernel suffers from two main problems: 1) shred memory bank conflicts and 2) thread divergence. In fact, almost all cuHE kernels suffer from those problems. In the following subsections, we point out the source of these problems and propose solutions to overcome them.

A. Shared Memory Bank Conflicts

As we mentioned previously, shared memory is divided into equally-sized banks. On current GPUs, accesses to shared memory are serviced for a warp at a time. Data stored/loaded

to/from shared memory in an interleaved way so that consecutive words are assigned to consecutive banks. Formally, word d_i is assigned to bank $i \pmod{32}$. This increases the throughput of shared memory by 32x compared to a single bank memory. Bank conflicts occur when two or more threads within a warp access two or more distinct words in the same bank. In this case, the hardware splits the request into as many requests such that no bank conflicts still exist.

Listing 1 exhibits two sources of bank conflicts. In line 28, there is a 4-way bank conflict in shared memory store. The conflict occurs between the four warp quarters. For example, in block 0, warp 0, quarter 0, threads $\{t_0, t_1, \ldots, t_7\}$ store their samples to banks $\{b_0, b_1, \ldots b_{15}\}$. Note that the samples are 2 words (64-bit). At the same time within the warp, quarters 1, 2 and 3, need to store their samples to the same banks. i.e. in each warp, 4 threads request access to distinct words in same memory bank leading to 4-way bank conflict.

To eliminate this conflict, we need to perform the following:

1) Split the 64-bit shared memory into two 32-bit arrays.
2) Add extra padding to shared memory.

First, we split the 64-bit array `buffer` into two 32-bit arrays: `buffer_l` and `buffer_h`, storing the lower words in `buffer_l` and the higher words in `buffer_h`. Next, we add extra padding to both arrays. The padding in this scenario is to insure that warp quarters 1, 2 and 3 store their words to different banks. This can be done by adding extra $7 * 8$ words and shifting the indices of quarter i by 8*i. By doing so, we eliminate all bank conflicts at the expense of adding extra 112 words.

The other source of bank conflicts is in line 39. This line includes a 2-way bank conflict in loading to shared memory `roots`. The conflict is within a warp quarter itself. For example, in block 0, warp 0, quarter 0, threads $\{t_0, \ldots, t_3\}$ request the same word d_0 in b_0. At the same time within the quarter, threads $\{t_4, \ldots, t_7\}$ request access to a different word in the same bank.

This bank conflict can be eliminated using the same approach used above. We split `roots` into 2 32-bit arrays. Next we add only 4 extra words to each array. Note that the reading and writing indices of `roots` are shifted by 4.

The rest of the kernels can be optimized similarly by padding shared memory arrays. The padding size varies from kernel to another. Table I shows the minimum size of required padding to eliminate shared memory conflict in each kernel for the shared memory arrays: `buffer` and `roots`.

In order to quantify the significance of this optimization, we measured the total number of shared memory transactions in kernel `ntt_1_16_k_ext` with and without our optimizations. After including our optimization, the total number of shared memory transactions has been reduced from 3840 to 3328. Table II shows the total shared memory requests for the NTT kernels with and without memory padding. The total shared memory requests is found as the sum of total shared memory load and store requests with the help of NVIDIA profiler *nvprof*. As we can see, the reduction in total memory transactions varies from 0% to 20%. Some kernels do not benefit from our optimization although shared memory bank conflicts are eliminated. The reason is that memory padding may result in vacuumed transactions as they contain non-useful padding words.

TABLE I: Size (in bytes) of shared memory paddings specified for each kernel. Any kernel may contain zero to two shared memory arrays: `buffer` *and* `roots`*.*

Kernel	Padding size	
	`buffer`	`roots`
`ntt_1_16k_ext`	448	32
`ntt_2_16k`	448	–
`ntt_3_16k`	120	–
`intt_1_16k`	448	32
`intt_3_16k_modcrt`	120	–
`ntt_1_32k_ext`	448	–
`ntt_2_32k`	448	–
`ntt_3_32k`	120	–
`intt_1_32k`	448	–
`intt_3_32k_modcrt`	120	–
`ntt_1_64k_ext`	448	–
`ntt_2_64k`	448	–
`ntt_3_64k`	–	–
`intt_1_64k`	448	–
`intt_3_64k_modcrt`	–	–

B. Thread Divergence

As we mentioned previously, branch statements may cause threads within a warp to take different paths. If the branches are short, the compiler uses predicated execution to eliminate divergence. On the other hand, if the branches are long such as nested conditionals or switch statements, thread execution is serialized which degrades the overall performance.

cuHE kernels suffer from this problem. In Listing 1 the call to `_ls_modP` in line 28 causes a thread divergence. This function takes two operands: 1) a data point, and 2) shift amount. It shifts the data point left $\pmod{p}$ by the specified amount. As mentioned in the beginning of this section, p has 8 as 64-th primitive root of unity. Instead of multiplying by powers of 8, shifting can be used. This is supposed to be more efficient than performing 64-bit by 64-bit multiplication $\pmod{p}$. However, the problem occurs when threads within a warp calls `_ls_modP` with different shift amounts. The implementation of `_ls_modP` includes a long switch case for every possible shift amount. A closer examination of line 28 shows that every 8 threads require a different shift amount. For example, for sample $i = 1$ in block 0, warp 0, threads within quarter 0 require 0 shifts, while those within quarter 1 require 3, those within quarter 2 require 6 and those within quarter 3 require 9. The variance in shift amount is even larger with subsequent samples and warps.

To eliminate thread divergence, we propose to pre-compute and store all powers of the 64-th primitive root of unity $\pmod{p}$ in GPU constant memory. Memory requirement for this is only 64 64-bit words. GPU constant memory is on-chip fast memory that imposes no restriction on access pattern. So, instead of shifting, each thread reads its corresponding twiddle factors and perform 64-bit integer multiplication $\pmod{p}$.

Experiments show that our proposed solution is more suitable for GPUs than shifting.

We remark here that although cuHE `_ls_modP` might be efficient and more suitable for CPUs, as it requires less work, this needs not to apply to GPU platforms due to the difference in execution model. Note that the same optimization can be used in other kernels. More specifically: kernels `ntt_1_16k_ext`, `ntt_2_16k`, `intt_1_16k`, `ntt_1_32k_ext`, `ntt_2_32k`, `intt_1_32k`, `ntt_1_64k_ext`, `ntt_2_64k`, and `intt_1_64k` can be modified to use the pre-computed roots.

In order to quantify the significance of this optimization, we measured the warp execution efficiency before and after including our optimization. Warp execution efficiency is defined as: the ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. With thread divergence, warp execution efficiency was 84.1%. After eliminating thread divergence, warp execution efficiency was increased to 100%. Table II shows the average warp execution efficiency for the NTT kernels with and without pre-computed roots. Note that kernels `intt_3_16k_modcrt` and `intt_3_32k_modcrt` do not include twiddling, therefore, our optimization is no included in them. Listing 2 shows the kernel after including our optimizations.

```
__global__ void ntt_1_16k_ext(uint64 *dst, uint32 *src) {
 __shared__ uint32 buffer_l[512+7*8];
 __shared__ uint32 buffer_h[512+7*8];
 __shared__ uint32 roots_l[64*2+1*4];
 __shared__ uint32 roots_h[64*2+1*4];
 register uint64 samples[8];
 register uint32 fmem, tmem, fbuf, tbuf; // from/to mem/buffer addr mapping

 //coalesced GMEM & zero SMEM bank conflicts
 fmem = ((tidx&0x38)<<5)|(bidx<<3)|(tidx&0x7);
 tbuf = (((tidx&0x38)<<3)|(tidx&0x7)) + 8 * ((tidx)>>3);
 fbuf = tidx;
 tmem = (bidx<<9)|((tidx&0x7)>>2<<8)|(tidx>>3<<2)|(tidx&0x3);
 roots_h[tidx] = tex1Dfetch(tex_roots_16k, (((bidx*2)*tidx)<<3)+1);
 roots_l[tidx] = tex1Dfetch(tex_roots_16k, ((bidx*2)*tidx)<<3);
 roots_h[tidx+64+4*1] = tex1Dfetch(tex_roots_16k, (((bidx*2+1)*tidx)<<3)+1);
 roots_l[tidx+64+4*1] = tex1Dfetch(tex_roots_16k, ((bidx*2+1)*tidx)<<3);

 #pragma unroll
 for (int i=0; i<4; i++)
 samples[i] = (src+(bidy<<14))[(i<<11)|fmem];
 _ntt8_ext(samples);

 #pragma unroll
 for(int i=0; i<8; i++)
 {
  uint64 tmp = _mul_modP(samples[i], const_w64Roots[((tidx>>3)*i)]);
  buffer_l[tbuf + 8*i] = tmp;
  buffer_h[tbuf + 8*i] = tmp >>32;
 }
 __syncthreads();
 //load 8 samples from shared mem in transposed way, compute 8-sample ntt
 #pragma unroll
 for(int i=0; i<8; i++)
 {
  uint64 tmp = buffer_h[((i<<6)|fbuf)+i*8];
  tmp = (tmp<<32) + buffer_l[((i<<6)|fbuf)+i*8];
  samples[i] = tmp;
 }
 _ntt8(samples);

 #pragma unroll
 for(int i=0; i<8; i++)
 {
  uint64 tmp = roots_h[((((tidx&0x7)>>2<<6)|(tidx>>3))|(i<<3))+4*((tidx&0x7)>>2)];
  tmp = (tmp<<32)+roots_l[((((tidx&0x7)>>2<<6)|(tidx>>3))|(i<<3))+4*((tidx&0x7)
      >>2)];
  (dst+(bidy<<14))[tmem|(i<<5)] = _mul_modP(samples[i], tmp);
 }
}
```

Listing 2: An optimized version of cuHE 1st kernel to compute the NTT of 16384 points. The optimizations eliminate shared memory bank conflicts in `buffer` and `roots` by adding extra memory padding and splitting arrays into two 32-bit arrays. The constant memory `const_w64Roots` is used to eliminate thread divergence.

V. Experimental Results and Evaluation

In this section, we evaluate our solutions and compare the optimized kernels with cuHE original kernels. We present our testing methodology followed by the hardware platform and configurations used. Next we present and discuss our experimental results.

A. Methodology and Testbed Environment

To evaluate our optimizations, we measure the running time of cuHE NTT and NTT^{-1} with and without optimization. The running time is measured using CUDA events. We test for three polynomial degrees: 8191, 16383, 32767 to compute the NTT and NTT^{-1} using cuHE kernel categories (cat) 16k, 32k and 64k, respectively. In each category, NTT is computed by invoking three kernels sequentially: `ntt_1_`⟨`cat`⟩`_ext`, `ntt_2_`⟨`cat`⟩ and `ntt_3_`⟨`cat`⟩. NTT^{-1} is also computed by invoking three kernels sequentially: `intt_1_`⟨`cat`⟩, `ntt_2_`⟨`cat`⟩ and `intt_3_`⟨`cat`⟩`_modcrt`. Each experiment was performed 512 times and the average running time is reported.

CUDA Toolkit v9.0 was used to develop our optimizations on a 64-bit server equipped with two 6-core CPUs (with hyper-threading enabled) and a Tesla K80 GPU with 3.7 compute capability. The operating system is ArchLinux (4.8.13-1-ARCH). The compilers used are GCC (6.3.1 20170109) and nvcc (8.0.61).

B. Timing Results

Table III lists timing and speedup gains before and after including our optimizations. It can be seen that NTT^{-1} takes longer time than NTT for all problem sizes. In cuHE, NTT^{-1} is computed as follows: given the NTT representation A of polynomial a, they reorder A as $\hat{A} = A_0, A_{2n-1}, A_{2n-1}, \ldots, A_1$. Then the NTT^{-1} of A is computed via $\text{NTT}^{-1}(A) = \frac{1}{N}\text{NTT}(\hat{A})$. At the end of NTT^{-1} computation, they perform one further reduction modulo a 32-bit prime to convert the polynomial to an internal representation, known as the Chinese Remainder Theorem (CRT) representation.

Another noticeable feature is that our optimizations enhance the performance of both NTT and NTT^{-1} by different factors ranging from 20% to 50%. Higher improvement can be noticed for category 16k. This is due to the fact that 16k kernels suffer from 2 sources of shared memory bank conflicts in arrays `buffer` and `roots`. However, categories 32k and 64k include only a single source of bank conflict in array `buffer` as shown in Table I.

VI. Conclusions

In this work, we conclusively demonstrated the performance influences of two distinct issues: shared memory bank conflict and thread divergence, for CUDA library. We analyzed the core kernels that compute the NTT and its inverse and showed that almost all kernels suffer from the above mentioned two problems. As both the issues correspond to the way in which data are organized, which facilitates easy access of the required

TABLE II: Memory requests count and warp execution efficiency for cuHE NTT kernels with and without optimizations. The warp execution efficiency is found via NVIDIA profiler nvpro specifying the metric: warp_execution_efficiency. The total shared memory requests is the sum of CUDA metrics: 1) shared_load_transactions and 2) shared_store_transactions. Both metrics are retrieved from nvprof.

Kernel	Warp execution efficiency		Total shared memory requests		
	Base %	Optimized %	Base	Optimized	Reduction %
ntt_1_16k_ext	83.98	**100.00**	3840	**3328**	**13.33**
ntt_2_16k	84.84	**99.99**	2560	**2048**	**20.00**
ntt_3_16k	99.93	**99.77**	2560	**2048**	**20.00**
intt_1_16k	84.95	**100.00**	3840	**3328**	**13.33**
intt_3_16k_modcrt	**100.00**	**100.00**	2560	**2055**	**19.73**
ntt_1_32k_ext	83.77	**100.00**	**6400**	**6400**	**0.00**
ntt_2_32k	84.61	**99.99**	5120	**4096**	**20.00**
ntt_3_32k	99.93	**99.75**	5442	**4437**	**18.47**
intt_1_32k	84.88	**100.00**	**6400**	**6400**	**0.00**
intt_3_32k_modcrt	**100.00**	**100.00**	5456	**4393**	**19.48**
ntt_1_64k_ext	83.82	**100.00**	12800	**11264**	**12.00**
ntt_2_64k	84.65	**100.00**	10240	**8192**	**20.00**
ntt_3_64k	87.65	**99.87**	**24576**	**24576**	**0.00**
intt_1_64k	84.87	**100.00**	12800	**11264**	**12.00**
intt_3_64k_modcrt	91.47	**99.96**	**24576**	**24576**	**0.00**

TABLE III: Running time (in microseconds) averaged over 512 runs of cuHE NTT and NTT^{-1} operations for different categories before and after including our optimizations. Base refers to cuHE kernels without optimizations whereas optimized refer to cuHE kernels with our optimizations.

Category	Operation	Base	Optimized	Speedup
16k	NTT	12.94	**9.66**	1.34x
	NTT^{-1}	14.51	**9.76**	1.49x
32k	NTT	19.03	**15.31**	1.22x
	NTT^{-1}	21.60	**16.22**	1.33x
64k	NTT	42.77	**34.47**	1.24x
	NTT^{-1}	49.78	**36.03**	1.38x

words for computation, we proposed solutions to alleviate the problems.

For shared memory bank conflicts, we split and padded the arrays with extra unused words to insure that shared memory accesses are directed to different banks reducing the total number of shared memory transactions. On the other hand, for thread divergence, we pre-computed 64 twiddle factors and stored them in GPU constant memory, which increased the warp execution efficiency to 100%. Although, our solutions require extra memory, experiments show that a fair performance improvement can be achieved.

We also provided a set of experiments to evaluate our solutions. Speedup gains ranging from 20% to 50% has been achieved for different problem sizes.

VII. Acknowledgment

This work was supported by Data Storage Institute, A * STAR and the National University of Singapore.

References

[1] David H Bailey. Ffts in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989.

[2] Alessandro Cilardo and Domenico Argenziano. Securing the cloud with reconfigurable computing: An fpga accelerator for homomorphic encryption. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1622–1627. IEEE, 2016.

[3] CryptoExperts. Fvnfllib. *Retrieved from FV-NFLlib: https://github.com/CryptoExperts/FV-NFLlib*, 2014.

[4] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

[5] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, 2015.

[6] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. *Proceedings of the IEEE*, 105(3):552–567, 2017.

[7] Niall Emmart and Charles C Weems. High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes. *Parallel Processing Letters*, 21(03):359–375, 2011.

[8] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[9] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[10] Shai Halevi and Victor Shoup. Helib. *Retrieved from HELib: https://github. com. shaih/HElib*, 2014.

[11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[12] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[13] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[14] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[15] Jerome A Solinas et al. *Generalized mersenne numbers*. Faculty of Mathematics, University of Waterloo, 1999.

[16] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[17] Wei Wang, Zhilu Chen, and Xinming Huang. Accelerating leveled fully homomorphic encryption using gpu. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 2800–2803. IEEE, 2014.

[18] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using gpu. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[19] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.