Multi-GPU Design and Performance Evaluation of Homomorphic Encryption on GPU Clusters

Ahmad Al Badawi, Bharadwaj Veeravalli, *Senior Member, IEEE*, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi, *Senior Member, IEEE*

Abstract—We present a multi-GPU design, implementation and performance evaluation of the Halevi-Polyakov-Shoup (HPS) variant of the Fan-Vercauteren (FV) levelled Fully Homomorphic Encryption (FHE) scheme. Our design follows a data parallelism approach and uses partitioning methods to distribute the workload in FV primitives evenly across available GPUs. The design is put to address space and runtime requirements of FHE computations. It is also suitable for distributed-memory architectures, and includes efficient GPU-to-GPU data exchange protocols. Moreover, it is user-friendly as user intervention is not required for task decomposition, scheduling or load balancing. We implement and evaluate the performance of our design on two homogeneous and heterogeneous NVIDIA GPU clusters: K80, and a customized P100. We also provide a comparison with a recent shared-memory-based multi-core CPU implementation using two homomorphic circuits as workloads: vector addition and multiplication. Moreover, we use our multi-GPU Levelled-FHE to implement the inference circuit of two Convolutional Neural Networks (CNNs) to perform homomorphically image classification on encrypted images from the MNIST and CIFAR $-$ 10 datasets. Our implementation provides 1 to 3 orders of magnitude speedup compared with the CPU implementation on vector operations. In terms of scalability, our design shows reasonable scalability curves when the GPUs are fully connected.

Index Terms—Homomorphic encryption, parallel algorithms, multi-GPU clusters, performance evaluation

1 Introduction

Fully Homomorphic Encryption (FHE) has drawn considerable attention from privacy-concerned application developers to design secure solutions that can guarantee data privacy at rest, motion and computation. FHE facilitates the capability to compute directly on encrypted data without decrypting. This is possible as FHE provides at least two gates: homomorphic addition (HAdd) and homomorphic multiplication (HMul) which take as operands encrypted input ($\mathbf{x}$) and produce encrypted output ($\mathbf{y} = f(\mathbf{x})$) [1]. In this setting, both $\mathbf{x}$ and $\mathbf{y}$ can only be decrypted by the secret key owner. On the other hand, the evaluator can learn nothing about the data other than what can be learned from a ciphertext such as its length and perhaps the encryption scheme parameters. To the best of our knowledge, under appropriate circular-security assumptions, FHE is considered semantically secure as hard as the underlying mathematical problems used to instantiate it [2].

FHE capabilities come at high computational cost in terms of space and runtime. This can be attributed to two main reasons: 1) plaintext data expansion due to encryption, and 2) the computation model used in FHE applications. The former is due to the reason that FHE masks the plaintext with very large algebraic structures such as vectors, matrices or polynomials. The dimensions of these structures are typically set to meet both functionality and security requirements. Therefore, arithmetic with such large structures requires an enormous amount of computation. What makes the situation even worse is that FHE can only be used if the desired computation is modelled as an arithmetic circuit. In such a computation model, loops with private counters have to be unrolled to the maximum value the counter can take. Moreover, conditional statements with private predicates require evaluating both branches and suppressing the false branch by masking.

当前FHE主要问题

引起上述两个问题的主要原因，解释主要原因的产生过程

给出一个FHE的应用例子

To better appreciate the computational complexity of FHE, an example of a recent FHE application may be useful. We summarize the complexity of homomorphic inference on encrypted images by means of Convolutional Neural Networks (CNNs) proposed by Faster CryptoNets [3]. The authors used levelled[1] FHE to classify several imaging problems such as MNIST [4], CIFAR $-$ 10 [5] and diabetic retinopathy medical images [6]. To classify a single encrypted image (with resolution 3 × 32 × 32) from the CIFAR $-$ 10 dataset, they had to run their CNN on n1$-$ *megamem* $-$ 96 Google Cloud Platform which includes 96 Intel Skylake 2.0 GHz vCPUs and 1433.6 GB RAM. Their 8-layer CNN required 22,372 seconds (or 6.2 hours) to perform the task using an unencrypted CNN model. Note

使用了级别（levelled）FHE来对几个图像问题进行分类，包括MNIST、CIFAR-10和糖尿病性视网膜病变医学图像。

[1] A levelled FHE is a scheme that can be parameterized to support circuits of a certain multiplicative depth.

- *Ahmad Al Badawi, Jie Lin, Xiao Nan, and Khin Mi Mi Aung are with the Institute for Infocomm Research (I^{2}R), A*STAR, Singapore 138632. E-mail: {ahmad, a0135956}@u.nus.edu, {lin-j, mi_mi_aung}@i2r.a-star.edu.sg.*
- *Bharadwaj Veeravalli is with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117576. E-mail: elebv@nus.edu.sg.*
- *Kazuaki Matsumura is with the Barcelona Supercomputing Center (BSC), Barcelona, Spain. E-mail: kazuaki.matsumura@bsc.es.*

Manuscript received 26 June 2019; revised 2 Aug. 2020; accepted 31 Aug. 2020. Date of publication 2 Sept. 2020; date of current version 11 Sept. 2020.
(Corresponding author: Ahmad Qaisar Ahmad Al Badawi.)
Recommended for acceptance by S. K Prasad.
Digital Object Identifier no. 10.1109/TPDS.2020.3021238

如果模型被加密，则需要更高的计算性能需求

that if the model is encrypted, higher computational requirements would be required.

The preceding use-case demonstrates clearly the computational requirements of basic FHE applications which cannot be met by commodity machines. An active research area in FHE is focused on hardware acceleration using GPUs [7], [8], [9] and FPGAs [10], [11]. Although these implementations have succeeded in improving the runtime, they have not considered the space requirement for FHE applications. A basic FHE application can require a large number of large ciphertexts. For instance, in homomorphic inference for deep learning imaging applications [3], [12], each pixel is encrypted in one ciphertext. Moreover, the output of each activation is also stored in a separate ciphertext.

Typically, ciphertext size is $2n \log q$ bits, where n is an integer in the order of several thousand and q is an integer in the order of several hundred. In order to enable FHE applications on such memory-limited accelerators, one solution is to distribute the workload on several devices. This can be done at the application level by the user or at a lower level included in the FHE implementation itself. The latter might be more preferable from the user perspective and provides higher usability.

提出了一种基于多gpu的FHE引擎的设计，它利用了多gpu集群的计算能力

1.1 Our Contributions

To this end, we propose a design for multi-GPU based FHE engine that leverages the computational power of multi-GPU clusters. Our design has the following major characteristics: 1) it is based on a data parallelism approach to distribute the computational load on available GPUs, which in turn compute cooperatively FHE primitives, 2) it includes efficient GPU-to-GPU data exchange protocols, 3) it is efficient and scales reasonably as the number of GPUs is increased on fully connected platforms, and 4) it is user-friendly as the user does not need to do manual parallelisation, task decomposition, scheduling, or load balancing.

In addition, we provide a set of experiments to evaluate the performance of our design in evaluating three homomorphic circuits (vector addition, vector multiplication and CNN inference with encrypted images) on 2 homogeneous and heterogeneous GPU clusters: NVIDIAK80, and a customized P100. A comparison with a recent task-parallelism-based implementation on multi-core CPUs is also provided. Without loss of generality, we target a particular levelled FHE scheme proposed originally by Fan and Vercautren (FV) [13] which was improved further by Halevi, Polyakov and Shoup [14] by introducing a Residue Number System (RNS)-based variant known as the Halevi-Polyakov-Shoup (HPS) scheme. It should be remarked that most second generation FHE schemes,[2] in particular those instantiated from the Ring Learning With Errors (Ring-LWE) problem, follow the same blueprint in their construction. Therefore, our design can be applied to other Ring-LWE schemes with minor adaptations.

Our design provides speedup factors ranging from 1 to 3 orders of magnitude compared to the multi-core CPU implementation on the vector operations workloads.

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

- We provide a distributed-memory multi-GPU design and implementation for Ring-LWE based FHE schemes on CUDA-enabled multi-GPU clusters, with focus on the HPS RNS variant of the FV levelled FHE scheme.
- We show how to partition the basic FHE data structures to distribute the computational load evenly on the available GPUs.
- We also provide efficient GPU-to-GPU communication protocols to facilitate data exchange among the GPUs.
- We provide a thorough performance evaluation of our implementation using different FHE applications as benchmarks on 2 homogeneous and heterogeneous GPU clusters and compare it with a recent multi-core CPU FHE implementation.

1.2 Organization of the Paper

The rest of the paper is organized as follows: in Section 2, we review a number of studies that have tried to improve the performance of FHE. Section 3 introduces briefly the general CUDA programming paradigm and mathematical background. In Section 4 we provide full details and layout of our implementation in addition to the optimization techniques used. Our experiments, benchmarking, and comparison results are presented in Section 5. Lastly, Section 6 draws some conclusions and provides guidelines for potential future work.

2 Literature Review

介绍了FHE性能优化的过程

The literature is rich with studies that have tried to improve the performance of FHE. A great deal of effort has been done at the construction level such as proposing new FHE schemes based on the hardness of different mathematical problems [13], [15], [19]. Another class of studies proposed new algorithms to accelerate the arithmetic operations incorporated in FHE such as employing the Chinese Remainder Theorem (CRT) [20], Double CRT [21], Number Theoretic Transform (NTT) [22], [23], [24], [25] and RNS [14], [26].

In addition to several algorithmic improvements to FHE theory, a great deal of effort has been dedicated to accelerating the performance of available schemes. This can be done by offloading time-consuming operations to hardware-accelerators such as FPGAs [10], [11], [27], [28], [29], [30], [31]. These works targeted several HE schemes and low-level mathematical operations of FHE such as large number multiplications, NTT and CRT. Of particular interest is HEPCloud which provides hardware design and implementation of the FV FHE scheme on Xilinx Virtex-6 ML605. HEPCloud shows around 13× speedup against CPU-only FV implementation [32] for toy parameter settings ($n = 2^{12}$). However, their design fails to scale well for larger FHE settings due to off-chip memory access. For instance,

[2] We note that FHE schemes are generally categorized into 3 generations. The first generation refers to Gentry's first realization of FHE [1]. The second includes a number of schemes such as BGV [15], FV [13], and CKKS [16]. The third generation includes schemes that are based on the techniques proposed in the GSW scheme [17] such as TFHE [18].

利用FPGAs加速FHE硬件

HEPCloud computes homomorphic multiplication for ($n = 2^{15}$) in 26.67 seconds (3.36 seconds spent on the computation and the rest on the off-chip memory access). This particular study shows the challenges that face FPGA implementations of FHE. We note that, to the best of our knowledge, the literature lacks any study that shows the effectiveness of FPGA implementations for any FHE application. Such a challenge has been set as a target of the DARPA DPRIVE project for FHE hardware acceleration with FPGAs [33].

More related to this work is FHE acceleration via GPUs. cuHE [7] is considered the first GPU acceleration of Ring-LWE based homomorphic encryption schemes. It is written in CUDA and includes an implementation of the Doröz-Hu-Sunar (DHS) [34] Somewhat HE (SHE) scheme. It supports both single- and multi-GPU machines following a task-parallelism approach. cuHE treats the underlying GPUs as independent processors and requires the user to distribute the computational task manually on available GPUs. Moreover, cuHE does not fully exploit the parallelism incorporated in FHE tasks and uses a large task granularity since the GPUs do not cooperate in the computational tasks. For instance, arithmetic on CRT or NTT matrices is handled by a single GPU, although such arithmetic is embarrassingly parallel and can be distributed among multiple GPUs for concurrent execution.

Cingulata是一个编译器工具链和运行时环境，
用于在使用FHE的加密数据上运行受限制的C++程序。

Another work that has tried to accelerate FHE via multicore CPUs was proposed by Cingulata [35] (previously known as Armadillo) which is a compiler toolchain and runtime environment for running restricted C++ programs on encrypted data using FHE. The input to Cingulata is a C++ code that must comply with some predefined restrictions. For instance, division or loops with encrypted counters are not supported yet. Cingulata converts the input C++ code into a Boolean circuit represented as a Directed Acyclic Graph (DAG). Moreover, it is shipped with a custom implementation of the FV levelled FHE scheme [13] that can be used to run the generated circuit on multi-core CPUs. Similar to cuHE, Cingulata views the underlying cores as independent processors as the cores do not cooperate on computing FHE primitives. It follows a task parallelism approach to execute independent FHE primitives concurrently on different cores. The task-processor assignment is performed automatically by a built-in task scheduler making Cingulata a more usable and user-friendly framework. One severe limitation of Cingulata is that it can only be used to generate Boolean circuits in F_2. With the current state of FHE, it is sometimes more efficient to design the target computation as an arithmetic circuit in F_p, where p is a prime number.

将底层处理器视作协作处理器

In this work, we follow a different parallelism approach and treat the underlying processors as cooperative processors with a focus on multi-GPU clusters. Our design combines the computational power of all GPUs into one. It aims at solving the following problems: 1) a GPU card is normally shipped with a small memory that may not be sufficient to accommodate for the large ciphertexts incorporated in FHE applications, and 2) most of FHE primitives are embarrassingly parallel and can benefit from a larger number of processors. In addition, our design is user-friendly as the user is not involved in task decomposition, scheduling or load balancing.

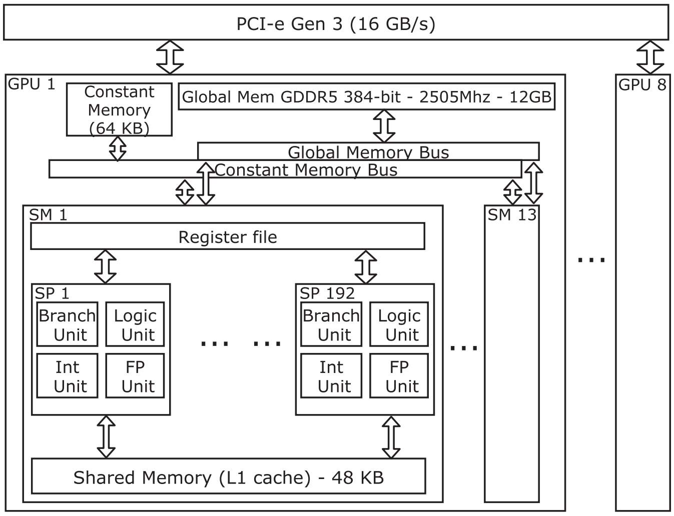


Fig. 1. A block diagram of NVIDIA Tesla K80. The figure is inspired by Figure 3.5 in [36].

cuHE是一个使用GPU加速FHE的实现，
但它的设计没有充分利用GPU的并行计算能力，
因为它将GPU视为独立的处理器，并未实现
计算任务的协作执行。

3 Background

This section introduces the basic notions the paper builds on. We start by introducing some technical background on CUDA GPU programming model. Then we provide an overview of the FV levelled FHE scheme.

介绍CUDA编程模型

3.1 CUDA Programming Model

GPUs are many-core computing platforms that follow the Single Instruction Multiple Data (SIMD) architecture. An NVIDIA GPU card consists of an array of Streaming Multiprocessors (SM) that share the GPU global memory. Each SM can be viewed as a vector processor that consists of many Scalar Processors (SPs) - a.k.a. cores - that share on-chip register file and an L1 cache that is known as the Shared Memory. An SP does the actual computation using the arithmetic and logical units it includes. Fig. 1 illustrates an NVIDIA Tesla K80 single-node GPU cluster that includes 8 cards connected via PCI-e Gen 3 bus.

In this cluster, GPUs can share data through the host node (CPU), a.k.a. staging, or communicate directly via PCI-e interconnection network. Obviously, remote access time is much longer than local memory access time similar to Non-Uniform Memory Access (NUMA) architectures.

An important notion in GPU programming is thread organization. At the highest level, the programmer organizes a number of threads in two structures before launching a kernel, which is part of the code that is executed by the GPU. The first structure is a grid that includes a number of thread blocks organized along 3 dimensions. The other structure is a block which is a 3D structure of threads as shown in Fig. 2. The maximum dimensions of these structures are hardware-dependent. Intra-block and inter-block communication can be performed through shared memory and global memory, respectively. 介绍了GPU的线程结构

Upon launching a kernel for execution on a particular device, thread blocks are mapped to the available SMs. A block cannot be divided and distributed onto several SMs but an SM can simultaneously execute multiple blocks if the resources are sufficient. The shared memory on an SM is divided evenly among the resident blocks. Similarly, the

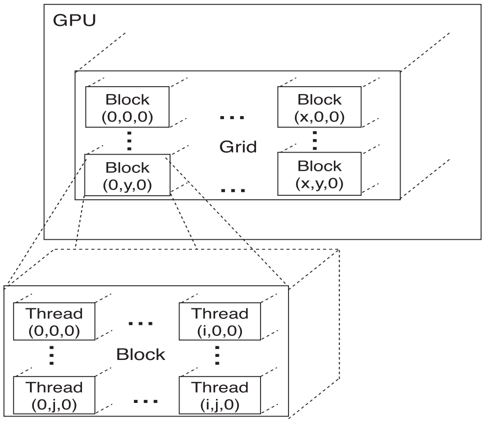


Fig. 2. CUDA thread organization in 3D grids and blocks.

registers are divided among all the resident threads in the SM. SM schedules for execution a group of 32 threads, known as warp, given that their data are available. If a warp stalls for any reason, the SM schedules another ready warp for execution to hide the latency. It is hence crucial for performance to ensure that there is enough number of active warps on SM. This can be done by minimizing the usage of shared memory and register files and distributing the computational task into several kernels or distributing the computation among multiple GPUs. single gpu到multi gpu的原因

Moving from single- to multi-GPU systems is usually motivated by three main reasons: 1) GPU cards include small memory that may not be sufficient to fit in the computational problem, 2) one may improve the performance by distributing the workload onto multiple GPUs, and 3) amortize the power consumption across the available GPUs by providing more performance per unit of power consumed. A major challenge facing multi-GPU applications is the inter-GPU communication overhead. Although current GPU clusters include high-speed networks for peer-to-peer communication, it is still much slower than local memory transfer rates. For instance, the global memory bandwidth in NVIDIA Tesla K80 is 240 GB/sec [37]. On the other hand, PCI-e Generation 3 (16× lanes) offers 16 GB/sec in one direction. Even NVIDIANVlink technology offers only 80 GB/sec in one direction. Therefore, designing an efficient inter-GPU communication is crucial to obtain the best performance of multi-GPU systems.

For a complete reference on CUDA programming, the reader is referred to [38] and the CUDA toolkit documentation [39].

3.2 FV

In the following paragraphs, we review the FV [13] scheme as an instance of Ring-LWE-based levelled-FHE schemes.

3.2.1 Background

The FV scheme implemented here employs the polynomial ring $R = \mathbb{Z}[X]/(X^n+1)$, where n is a power of 2. R can be viewed as a set of polynomials of degree less than n. Addition and multiplication in R are done modulo (X^n+1). In some FV primitives, the polynomials are sampled from predefined distributions. We use the symbol $a \overset{\mathcal{U}}{\leftarrow} \mathcal{S}$ to refer to uniform sampling of a from the set $\mathcal{S}$, whereas the symbol $a \overset{\mathcal{G}}{\leftarrow} \mathcal{S}$ is used for sampling from a Gaussian distribution.

The plaintext space in FV is R_t, with $t \geq 2$ being an integer plaintext modulus. The polynomials in R_t are reduced modulo t and (X^n+1). A plaintext is normally a single element in R_t encoding the original plaintext message. Likewise, the ciphertext space R_q has $q \gg t$ as the coefficient modulus. For practical implementations, q is usually composite s.t. $q = \prod_{i=1}^{k} p_i$, where p_i is a prime that fits in the underlying machine word size. A ciphertext ct is a pair of two elements in R_q, denoted by $(ct[0], ct[1])$.

3.2.2 The FV Scheme

介绍FV方案

The textbook FV scheme, described in [13], is a tuple of 5 procedures: key generation, encryption, decryption, homomorphic addition and homomorphic multiplication. It defines a set of parameters as follows: 密钥生成、加密、解密、同态加法和同态乘法

- λ: known as the security parameter, which characterizes the computational requirements to break the scheme using 2^λ elementary operations.
- w: a decomposition base used to express a polynomial in R_q in terms of $l+1$ polynomials in base $w \in \mathbb{Z}$, where $l = \lfloor \log_w q \rfloor$.
- $\mathcal{X}_{err}$: a zero-mean discrete Gaussian distribution used to sample error polynomials, parameterized by the standard deviation σ and error bound β_{err}.
- $t \geq 2$: a plaintext modulus integer.
- $q \gg t$: a ciphertext modulus integer.

The main five primitives of the scheme are as follows:

- *KeyGen*(λ, w): The Secret Key (sk) is a ternary polynomial $sk \overset{\mathcal{U}}{\leftarrow} R_2$ with values from $\{-1, 0, 1\}$. The Public Key (pk) is a pair of polynomials $(pk_0, pk_1) = (-[a \cdot sk + e]_q, a)$, where $a \overset{\mathcal{U}}{\leftarrow} R_q$ and $e \overset{\mathcal{G}}{\leftarrow} \mathcal{X}_{err}$. The Evaluation Key (evk) is a set of $(l+1)$ pairs of polynomials generated as follows: for $0 \leq i \leq l$, sample $a_i \overset{\mathcal{U}}{\leftarrow} R_q$ and $e_i \overset{\mathcal{G}}{\leftarrow} \mathcal{X}_{err}$. $evk[i] = ([w^i s^2 - (a_i \cdot sk + e_i)]_q, a_i)$. This procedure outputs the tuple: (sk, pk, evk).
- *Enc*(μ, pk): takes a plaintext $\mu \in R_t$, and samples $u \overset{\mathcal{U}}{\leftarrow} R_2$ and $e_1, e_2 \overset{\mathcal{G}}{\leftarrow} \mathcal{X}_{err}$. It computes the ciphertext $ct = ([\Delta\mu + pk[0]u + e_1]_q, [pk[1]u + e_2]_q)$, where $\Delta = \lfloor q/t \rfloor$.
- *Dec*(ct, sk): computes $\mu = \left[\left\lfloor \frac{t}{q}[ct[0] + ct[1]sk]_q \right\rceil\right]_t$.
- *HAdd*(ct_0, ct_1): homomorphic addition takes two ciphertexts and produces: $ct_{add} = ([ct_0[0] + ct_1[0]]_q, [ct_0[1] + ct_1[1]]_q)$.
- *HMul*(ct_0, ct_1, evk): homomorphic multiplication takes two ciphertexts and computes:

  1) Tensoring: compute c_τ, with $\tau \in \{0,1,2\}$, such that

$$c_0 = \left[\left\lfloor \frac{t}{q} ct_0[0] \cdot ct_1[0] \right\rceil\right]_q,$$

$$c_1 = \left[\left\lfloor \frac{t}{q} (ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0]) \right\rceil\right]_q,$$

$$c_2 = \left[\left\lfloor \frac{t}{q} ct_0[1] \cdot ct_1[1] \right\rceil\right]_q.$$

  2) Relinearization:
  2.1) decompose c_2 in base w as $c_2 = \sum_{i=0}^{l} c_2^{(i)} w^i$.
  2.2) return $ct_{mult}[j]$, with $j \in \{0,1\}$, such that

TABLE 1
Main Operations in RNS Variants of the FV Scheme, the Affected Matrix, and the Way it is Processed

Operation	Affected matrix	Orientation
Polynomial addition	RNS or DGT	point-wise
Polynomial subtraction	RNS or DGT	point-wise
Polynomial multiplication	DGT	point-wise
Fast Base Extension	RNS	column-wise
Scale and Round	RNS	column-wise
DGT	RNS	row-wise
DGT^{-1}	DGT	row-wise

$$ct_{mult}[j] = \left[c_j + \sum_{i=0}^{l} evk[i][j]c_2^{(i)}\right]_q.$$

Note that the relinearization procedure shown above is based on the first version of relinearization in the FV scheme [13]. This version does not require an extended coefficient modulus and does not affect the security analysis when selecting FHE parameters.

3.2.3 *Polynomial Representations*

It can be clearly seen that the basic data structure in FV, and almost all Ring-LWE schemes, is polynomial. The dimensions of this data structure - given that RNS representation is employed - are n and k, where n is the number of coefficients and k is the number of prime factors in q. In current implementations of Ring-LWE-based cryptography, polynomials are generally represented in two forms: 1) coefficient representation in RNS and 2) evaluation represeentation in NTT [8], [9], [32], [40], [41], [42]. Polynomials are usually generated in the first representation which allows efficient polynomial addition and subtraction in $\mathcal{O}(kn)$. The evaluation representation can be used for efficient modular addition, subtraction and multiplications, each in $\mathcal{O}(kn)$. The NTT and its inverse NTT^{-1} are used to switch between the two representations with time complexity $\mathcal{O}(kn \log n)$ for each operation. We remark that in our implementation, we use a variant of NTT known as the Discrete Galois Transform (DGT) [43] to compute the nega-cyclic convolution for efficient modular polynomial multiplication in power-of-two cyclotomic rings. The key property of DGT is that it allows computing an n-point convolution using an $\frac{n}{2}$-FFT datapath, at the expense of using Gaussian integers arithmetic. In a previous study, we showed that the DGT is suitable for GPU implementations due to halving the number of twiddle roots and storing them in smaller lookup tables [8].

Although one can perform addition, subtraction and multiplication in $\mathcal{O}(n)$ in NTT/DGT representation, unfortunately, one cannot stay indefinitely in NTT/DGT as some operations require switching to RNS. Examples of these operations are scale and round and base decomposition [14], [26], both required in decryption and homomorphic multiplication for RNS variants of the FV scheme. The switching is required as they are non-DGT/NTT friendly and require access to the coefficients of the polynomial. In fact, performing these operations in DGT/NTT is still an open problem.

3.3 Elementary Operations in RNS FV

We chose the HPS [14] RNS variant of the FV scheme for our implementation. The use of RNS is to alleviate the need for multi-precision arithmetic. Moreover, it provides a substantial amount of parallelism that can be exploited by parallel implementations. There are at least two RNS variants of the FV scheme: 1) the Bajard-Eynard-Hasan-Zucca (BEHZ) scheme [26] and 2) the HPS scheme. The (BEHZ) scheme is reported to offer 2× to 4× speedup in homomorphic multiplication compared with the textbook FV [26]. We target HPS as it has been shown to offer slightly better performance over the BEHZ variant [9].

The core primitives of the HPS RNS variant of the FV scheme process the polynomial matrices row-, column- or point-wise. There are also a set of row-wise operations to convert one matrix to the other. Table 1 shows the main operations in RNS FV, matrix they operate on and how it is processed.

Polynomial arithmetic such as addition, subtraction and multiplication can be done using point-wise operations on either the RNS or DGT matrix for addition and subtraction or the DGT matrix for multiplication. The FastBaseExtension (FBE) and ScaleandRound (SaR) are used for scaling (division) in RNS. The DGT and its inverse DGT^{-1} are used to convert the polynomial matrix from RNS to DGT and vice versa, respectively. For further details on these operations, we refer the reader to [9].

4 Implementation Layout

In this section, we provide full details of our implementation.

4.1 Data Structures

A polynomial, regardless of its representation, is stored in a matrix of size $k \times n$ of $(\max_{1\leq i\leq k} \log_2 p_i)$-bit integers. As we shall see later, the FV primitives process these matrices row-, column- or element-wise. For instance, key generation, encryption and homomorphic addition require only row- and point-wise operations in DGT representation. On the other hand, decryption and homomorphic multiplication require a combination of row-, column-, and point-wise operations. Since our design exploits data parallelism, it requires efficient methods to partition the matrices among available GPUs. Since we have a mix of access patterns to the polynomial matrices, this imposes a need for different matrix partitioning methods. In one layout, the matrices are partitioned vertically (assigning a set of columns) across the GPUs, which we refer to as column-partitioning. In the other layout, the matrices are partitioned horizontally (assigning a set of rows) across the GPUs, which we refer to hereafter as row-partitioning.

To this end, we provide methods to partition the matrices in a row- or column-partitioning depending on the desired operation. We also need a method to convert the partitioning layout from row-partitioning to column-partitioning and vice versa.

Fig. 3 shows our partitioning procedures. For column-partitioning, we allocate a block of size $k \times (n/m)$ on each GPU.[3] Matrices are then split and copied to the respective

[3] We assume m is a power of 2 and less than n.

GPU memory. After partitioning, each GPU includes (n/m) columns. For row-partitioning, we allocate a block of size $(k/m) \times n$ on each GPU. The partitioning results in storing (k/m) rows in each GPU. Although the figure shows the original matrix packed, this may not be the case as the matrix can be created already partitioned across the GPUs.

To switch the partitioning layout, we provide Algorithms 1 and 2 which convert from column- to row-partitioning and row- to column-partitioning, respectively. Both algorithms assume that the input segments are allocated as 1-dimensional linear arrays. This requirement is actually compatible with the CUDA language. They also assume that the segments are partitioned evenly on the GPUs. However, they can be easily generalized to support non-even partitions by allocating the excessive amount to a designated GPU. Note that the algorithms switch the partitioning layout directly between the GPUs without referring back to the CPU.

Algorithm 1. Column to Row Partitioning

Input: k, n, m number of rows, columns and available GPUs. *cvec*: a vector of pointers to the column segments in each GPU
Output: *rvec*, a vector of pointers to the row segments in each GPU

```
for $i$ : 0 to $m - 1$ do
  $rvec[i]$ = Allocate $(k/m)$ rows in GPU $i$
for $src$ : 0 to $m - 1$ do
  for $dst$ : 0 to $m - 1$ do
    for $row$ : 0 to $(k/m) - 1$ do
      $from = cvec[src] + (k/m) * (n/m) * dst + row * n/m$
      $to = rvec[dst] + (n/m) * src + row * n$
      $segment\_size = (n/m)$
      copy(to, from, segment_size)
Return $rvec$
```

Algorithm 2. Row to Column Partitioning

Input: k, n, m number of rows, columns and available GPUs. *rvec*: a vector of pointers to the rows in each GPU
Output: *cvec*, a vector of pointers to the column segments in each GPU

```
for $i$ : 0 to $m - 1$ do
  $cvec[i]$ = Allocate $(n/m)$ columns in GPU $i$
for $src$ : 0 to $m - 1$ do
  for $row$ : 0 to $(k/m)$ do
    for $dst$ : 0 to $m - 1$ do
      $from = rvec[src] + (n/m) * dst + row * n$
      $to = cvec[dst] + (k/m) * (n/m) * src + row * (n/m)$
      $segment\_size = (n/m)$
      copy(to, from, segment_size)
Return $cvec$
```

Each of Algorithms 1 and 2 requires $\mathcal{O}(mk)$ communication across the GPUs. The amount of data transferred (`segment_size`) in each GPU-to-GPU transaction is n/m elements. It is worth noting that in FHE, the polynomial matrices are usually wide, i.e., $n \gg k$. Also, the number of available GPUs m can be assumed to be much smaller than n. This suggests that the total number of transfers in both algorithms may not be very high. We note that CUDA includes a function to migrate 2D data block - (`cudaMemcpy2D`). Although this can lead to simpler algorithms (2 loops instead of 3), this function may not be supported in other programming models. Therefore, we opt to describe the general case. An optimization we used here is to use CUDA multiple streams. On each device, we reserve $m - 1$ streams and dispatch the copy on each stream for maximum parallelization. This way the copies from one device are broadcast concurrently to all other devices. We should note that we keep these streams in a pool (initialized at start up) to eliminate redundant stream allocation and deallocation overhead.

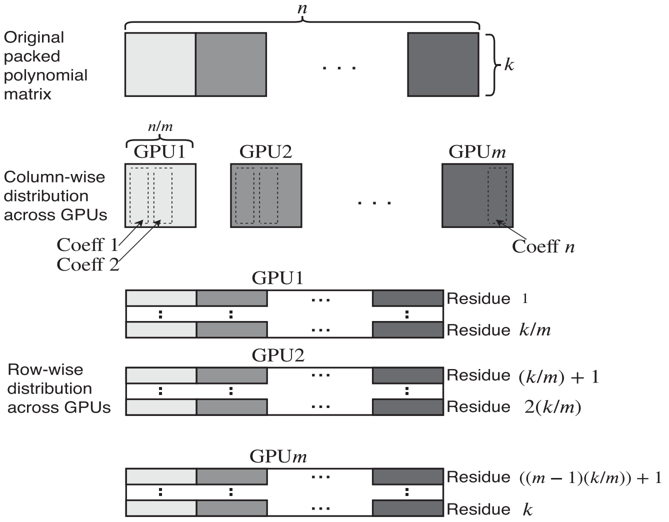
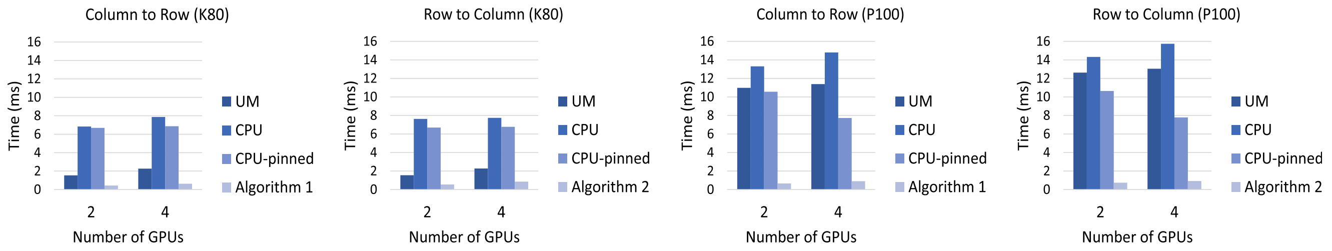

n
Original packed polynomial matrix
k
n/m
GPU1
GPU2
GPUm
Column-wise distribution across GPUs
Coeff 1
Coeff 2
Coeff n
GPU1
Residue 1
Residue k/m
GPU2
Row-wise distribution across GPUs
Residue (k/m) + 1
Residue 2(k/m)
GPUm
Residue ((m − 1)(k/m)) + 1
Residue k

Fig. 3. Row and column partitioning of polynomial matrix across multiple GPUs.

An alternative and probably simpler solution is to combine the segments in CPU memory and do the partitioning on CPU. However, this solution requires the CPU involvement in communication which can be slower than inter-GPU communication.

Another easy solution is to use CUDA unified memory which provides a single address space that can be accessed by any processor in the system - CPU or any GPU. In unified memory, an allocated memory pointer can be used in the program by any device. CUDA migrates the data automatically when it is requested by any processor in the system. The problem with this approach is that the programmer cannot guarantee where the data will be stored and may not be able to distribute the computation efficiently across the GPUs.

To evaluate the performance of the aforementioned partitioning methods, we implemented each one using CUDA 9.0 on NVIDIATESLAK80 and NVIDIATESLAP100 GPU clusters whose specifications can be found in Table 3. Fig. 4 shows the latency in (milliseconds) of partitioning a matrix of size 40×2^{16} 4-byte elements while varying the number of GPUs. As seen, explicit memory transfer among GPUs has the best performance among the four approaches as current GPU clusters are shipped with high-speed communication links. Note that although the communication bus in both clusters has the same bandwidth, the latency on P100 is slightly higher as the interconnection topology is slightly different. As shown in Fig. 6, the 4 GPUs in K80 are fully



(a) (b) (c) (d)

Fig. 4. Latency (in ms) of partitioning a matrix of dimensions $(k = 40, n = 2^{16})$ via four different approaches on NVIDIATESLAK80 and NVIDIATESLAP100. UM: unified memory. CPU: partitioning on CPU. CPU-pinned: partitioning on CPU but with pinned memory.

connected. On the other hand, P100 includes one isolated GPU which stages exchanged data through the CPU.

4.2 Implementing the FV Primitives

4.2.1 Key Generation

This primitive requires three elementary operations: 1) random sampling, 2) polynomial addition and 3) polynomial multiplication, all can be done using point-wise operations on the RNS/DGT matrices.

First, we ensure that the polynomial structure is in column-partitioning. Each GPU generates n/m columns of each random polynomial in RNS representation. Three random distributions are required: 1) uniform in $\mathbb{Z}_2$, 2) uniform in $\mathbb{Z}_q$ and 3) discrete Gaussian from $\mathcal{X}_{err}$. We use CUDAcuRAND to sample from these distributions. cuRAND already includes functions to generate uniform integers. To implement $\mathcal{X}_{err}$, we use the Box-Muller algorithm [44] which transforms a uniformly sampled number into a pair of independent normally distributed random numbers. Note that this method provides a rounded continuous Gaussian distribution instead of true discrete Gaussian distribution. It is chosen for its simplicity and high-suitability for GPUs as the coefficients can be generated independently. 生成公钥和评估密钥所需的步骤

Generating the public and evaluation keys requires polynomial multiplication and addition. First, we use Algorithm 1 to switch into row partitioning. The DGT procedure is then performed row-wise simultaneously on each GPU. Afterwards, polynomial multiplication and addition are performed via point-wise operations simultaneously across all GPUs. The generated keys are stored in DGT representation in row-partitioning layout across the available GPUs.

4.2.2 Encryption

The encryption procedure requires the generation of 3 random polynomials, $u \xleftarrow{\mathcal{U}} R_2$ and $e_1, e_2 \xleftarrow{\mathcal{G}} \mathcal{X}_{err}$. At first, they are generated across the GPUs in a column-partitioning layout. Next, the layout is converted to row-partitioning to perform polynomial arithmetic using point-wise operations across available GPUs. The generated ciphertext includes polynomials that are stored in row-partitioning across available GPUs.

4.2.3 Homomorphic Addition

Homomorphic addition accepts two ciphertexts that are stored in row/column partitioning. Without layout conversion, the two ciphertexts are point-wise added. The polynomials in the generated ciphertext are also in the same inputs' partitioning layout.

4.2.4 Decryption

The decryption procedure is calculated in two steps: 1) evaluating the ciphertext at the secret key, and 2) performing the SaR function on the result, where the scale quantity is (t/q). The first step includes polynomial addition and multiplication that can be done via point-wise operations.

The second step is more involved and requires a conversion of the polynomials to the RNS domain which can be done via DGT^{-1} on each row. The next step is computed using the simple SaR procedure in [14], which requires access to the columns of the RNS matrix. Therefore, we change the partitioning into a column-partitioning layout. For completeness, we briefly describe the simple SaR procedure referring the reader to the original proposal for more details.

Let $x \in \mathbb{Z}_q$ be in RNS representation $(x_1, \ldots, x_k)$. The procedure computes $y = \lfloor \frac{t}{q} \cdot x \rceil \in \mathbb{Z}_t$. This can be done using Equation (1).

$$\left\lfloor \frac{t}{q}[x]_q \right\rceil = \left[\left\lfloor \left(\sum_{i=1}^{k} x_i \cdot \left[\left(\frac{q}{q_i}\right)^{-1}\right]_{q_i} \cdot \frac{t}{q_i}\right)\right\rceil\right]_t . \tag{1}$$

It is worth noting that the quantity $[\left(\frac{q}{q_i}\right)^{-1}]_{q_i} \cdot \frac{t}{q_i}$ can be precomputed and stored in a lookup table.

The output of decryption is a polynomial that is stored in a column partitioning layout.

4.2.5 Homomorphic Multiplication

Homomorphic multiplication is known to be the most performance-critical primitive in FHE schemes. Fig. 5 shows a simplified flow of an RNS-compatible FV homomorphic multiplication. For better visualization, we show the RNS matrices without partitioning.

The input polynomials are assumed to be in RNS with column-partitioning layout. The polynomials are extended from base $q = \{p_1 \times p_2 \times \ldots \times p_k\}$ to an auxiliary base $B = \{p_{k+1} \times p_{k+2} \times \ldots \times p_{k+l}\}$ using the FBE procedure shown in Equation (2). The partitioning layout of the polynomials is converted into row-partitioning for DGT conversion.

$$\text{FastBaseExtension}(x, B, B') =$$
$$\left[\left(\sum_{i=1}^{k} \left[x_i \left[\left(\frac{q}{p_i}\right)^{-1}\right]_{p_i}\right]_{p_i} \frac{q}{p_i}\right) - \upsilon \cdot [q]_{p'_j}\right]_{p'_j \in B'} . \tag{2}$$

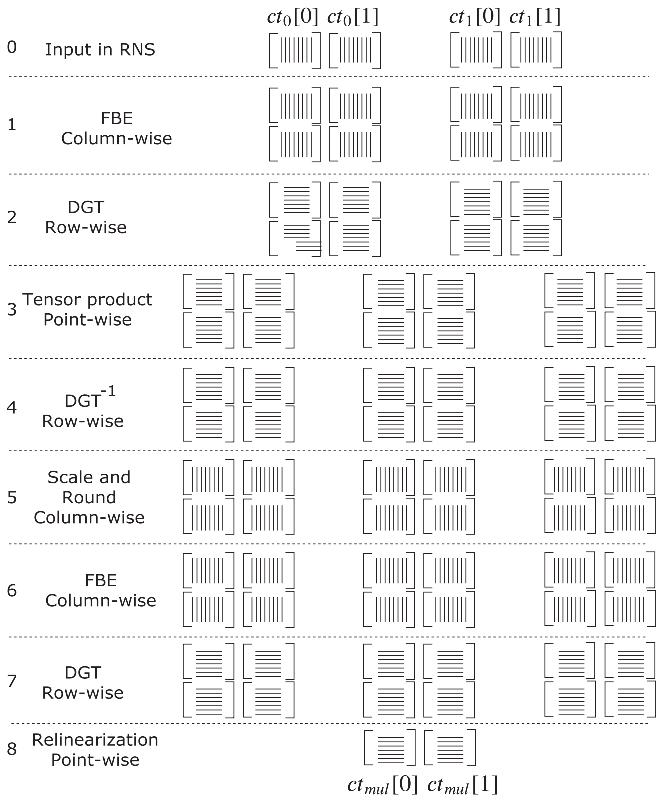

$ct_0[0]$ $ct_0[1]$
$ct_1[0]$ $ct_1[1]$
0 Input in RNS
1 FBE Column-wise
2 DGT Row-wise
3 Tensor product Point-wise
4 DGT^{-1} Row-wise
5 Scale and Round Column-wise
6 FBE Column-wise
7 DGT Row-wise
8 Relinearization Point-wise
$ct_{mul}[0]$ $ct_{mul}[1]$

Fig. 5. RNS-compatible homomorphic multiplication [14].

where,

$$v = \left\lfloor \sum_{i=1}^{k} \frac{1}{p_i} \left[x_i \cdot \left[\left(\frac{q}{p_i} \right)^{-1} \right]_{p_i} \right]_{p_i} \right\rceil .$$

Next, we calculate the tensor product (Step 1 in HMul) without SaR. This is mainly a set of polynomial additions and multiplications that can be done point-wise on the DGT matrix. In Step 4, the DGT^{-1} is called to convert the matrices into the RNS domain to do the SaR procedure which requires access to the coefficients of the polynomials. This is followed by switching the partitioning layout to column-partitioning.

The SaR procedure (Steps 5 and 6) is similar to the simple SaR procedure used in decryption. We refer the reader to the original text for a detailed description.

Since SaR is computed on the RNS matrix column-wise, the partitioning layout is converted to row-partitioning for DGT conversion which is performed in Step 7. Lastly, the relinearization procedure is performed using a dot-product-like operation on the intermediate resultant polynomials and the evaluation key. This is done via point-wise operations on the DGT matrices. The output is a ciphertext in DGT representation with row-wise partitioning. Given that the input ciphertexts are in RNS, each homomorphic multiplication requires 3 partitioning layout transforms using Algorithms 1 and 2.

It is worth noting that during homomorphic multiplication, the ciphertext size expands even further. The FBE (Step 1 in Fig. 5 almost doubles the input ciphertexts size. Moreover, the tensor product (Step 3) requires an extra 4 more temporary matrices. All in all, homomorphic multiplication requires 3× the input ciphertext size.

4.2.6 A Note on the Precomputed Constants

We note that all the precomputed quantities used such as the set of CRT primes p_i's, and the constants used in FBE, decryption, and homomorphic multiplication are precomputed on CPU and transferred to GPU memory at system initialization.

5 Performance Evaluation

In this section, we evaluate the performance of our implementation using two sets of experiments. In the first, we use two simple FHE workloads: 1) homomorphic vector addition and 2) homomorphic vector multiplication and compare the performance with Cingulata [35], a multi-core CPU FHE compile-chain. Our purpose is to show the performance improvement our implementation can offer compared to shared-memory multi-core CPU implementations. In the second set, we build 2 CNNs to perform homomorphic inference task and classify encrypted images from the MNIST [45] and CIFAR – 10 [5] datasets. The main purpose of this set of experiments is to show that our implementation can handle realistic ultra-large FHE computation.

5.1 Methodology

用于评估设计和实现性能的三个同态应用：1）同态向量加法（HVAdd），2）同态向量乘法（HVMul），以及两个CNN的同态推理电路（HCNN）

Three homomorphic applications are used to evaluate the performance of our design and implementation: 1) homomorphic vector addition (HVAdd), 2) homomorphic vector multiplication (HVMul), and the homomorphic inference circuit of two CNNs (HCNN). In HVAdd and HVMul, we basically do point-wise addition and multiplication on two vectors of ciphertexts. The size of these vectors is fixed at 500 for HVAdd and 250 for HVMul. We built both workloads using our implementation and the Cingulata framework. The parameters of the FV scheme are fixed to simulate a heavy FHE computation where we set $(\log_2 n, \log_2 q, t) = (15, 600, 2)$；

HVAdd表示一个没有通信开销的工作负载，因为既不调用算法1也不调用算法2。

HVAdd represents a workload with zero communication overhead as both Algorithms 1 and 2 are not invoked. On the other hand, each ciphertext in HVMul requires 3 partitioning layout transforms. The main purpose of this experiment is to show how our implementation scales in the best-case scenario (HVAdd) where there is no inter-GPU communication and the worst-case scenario (HVMul) where inter-GPU communication is highest.

It should be remarked that Cingulata follows a different parallelism approach. In contrast to our data parallelism approach, Cingulata uses a task parallelism approach. It treats the input circuit as a DAG and assigns independent tasks to idle processors. In other words, the underlying cores are treated as autonomous Processing Units (PUs) that process the assigned tasks without mutual cooperation. Moreover, Cingulata is designed for shared memory architectures, unlike our approach which supports distributed memory architectures.

In the HCNN workloads, we aim at showing that our implementation can handle realistic workloads with ultra large FHE computations. We implement the two CNNs (HCNN – 5 – layer and HCNN – 11 – layer) in [46] using our multi-GPU implementation of the FV scheme to perform the MNIST and CIFAR – 10 image classification tasks. The MNIST dataset consists of 60,000 images (50,000 in training dataset and 10,000 in testing dataset) of hand-

TABLE 2
Workload Characteristics

Workload	# Inputs	# Outputs	# $(C \times C)$	# $(C \times P)$	# $(C + C)$	# $(C + P)$
HVAdd	1,000	500	0	0	500	0
HVMul	500	250	250	0	0	0
HCNN-5-Layer	784	10	1,520	46,000	46,000	1,950
HCNN-11-Layer	3,072	10	57,344	6,952,332	6,952,332	91,728

C and P denote ciphertext and plaintext, respectively.

written digits, each is a 28×28 array of values between 0 and 255. CIFAR − 10 [5] consists of 60,000 color images (50,000 in training dataset and 10,000 in testing dataset) of 10 different classes. Each image consists of $32 \times 32 \times 3$ pixels of values between 0 and 255. Note that in these experiments, the model is not encrypted and assumed to be pre-learned on non-encrypted data. However, input data that need to be labelled are encrypted by the data owner and sent to the evaluator for homomorphic prediction. We followed [46] to train the CNNs used in our implementation. In terms of accuracy, HCNN − 5 − Layer and HCNN − 11 − Layer provides 99 and 77.55 percent on MNIST and CIFAR − 10 datasets, respectively.

The FV parameters used for the HCNN workloads are $(\log_2 n, \log_2 q, t) =$ (14, 330, 5522259017729) for HCNN − 5− $Layer$ and $(\log_2 n, \log_2 q, t) =$ (13, 300, 2424833, 2654209, 2752513, 3604481, 3735553, 4423681, 4620289, 4816897, 4882433, 5308417) for HCNN − 11 − Layer. These sets of parameters provide security level $\lambda \geq 80$ bits, which is considered sufficiently secure for current deployment [47]. The Learning-with-Errors (LWE) estimator is used to measure the security level of the parameters [2].

To better appreciate the computational load of these workloads, Table 2 shows the total number of homomorphic operations in each workload.

5.2 Testbed Environment

Experiments of this section were performed on 2 NVIDIA multi-GPU clusters: 1) K80, and 2) P100 whose hardware configurations can be found in Table 3. A top-level view of the interconnection topology in these clusters is shown in Fig. 6. Note that K80 is a homogeneous cluster comprising 2 groups of 4 fully connected identical K80 GPU cards. On the other hand P100 is heterogeneous as it consists of 1 isolated V100 and 3 fully connected P100 cards.

TABLE 3
Hardware Configuration of the Testbed Servers

Specification	K80	P100	
Model	K80	V100	P100
Compute Capability	3.7	7.0	6.0
# of PUs	8	1	3
# Cores (total)	8 × 2496	5120	3 × 3584
Core Frequency	0.82 GHz	1.380 GHz	1.328 GHz
Mem Type	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Mem Bandwidth	240 GB/sec	732 GB/sec	900 GB/sec
Mem Capacity	8 × 12 GB	16 GB	3 × 16 GB
CUDA Version	CUDA 9.0	CUDA 9.0	
CPU	Intel Xeon E5-2620	Intel(R) Xeon(R) Platinum	
CPU Frequency	2.40 GHz	2.10 GHz	
CPU Mem Capacity	64 GB	191 GB	
CPU cores	12	52	
OS	ArchLinux 4.19.32-1	ArchLinux 5.0.5-arch1-1	
Compiler	gcc 8.2.1	gcc 8.2.1	
PCI-e Bandwidth	16 GB/sec	16 GB/sec	

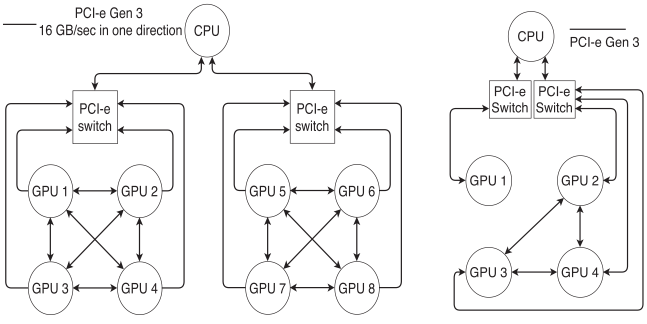
5.3 Benchmarks

In this section, we evaluate the performance of our implementation using the two sets of workloads: 1) encrypted vector operations and 2) homomorphic inference of CNN for image classification.

5.3.1 Encrypted Vector Operation

Table 4 shows the latency of running HVAdd and HVMul. It can be clearly seen that our GPU implementation outperforms Cingulata in both workloads regardless of the number of PUs. Our GPU implementation shows speedup factors ranging from 173.69× to 936.20× for HVAdd and 13.29× to 366.74× for HVMul compared with the multi-core CPU implementation provided by Cingulata. P100 shows the best performance for HVAdd on 1, 2 and 4 GPUs.

For HVMul, P100 shows in general better performance up to 2 GPUs. On 4 GPUs, the performance degrades due to the



(a) K80 (b) P100

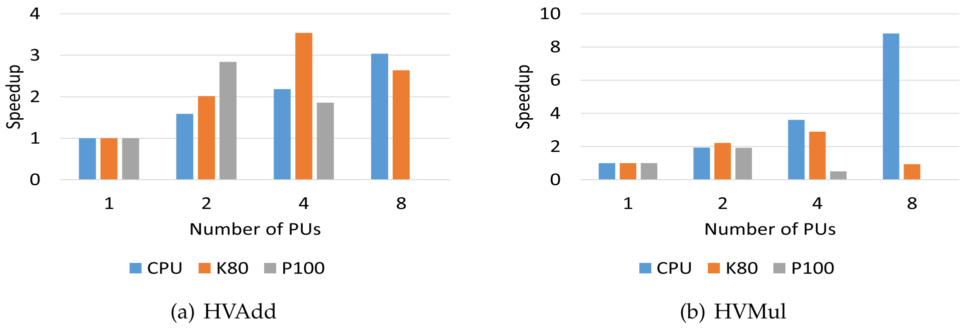
Fig. 6. Interconnection topology in a) K80, and b) P100.

TABLE 4
Latency (in seconds) of HVAdd and HVMul Via Cingulata (CPU), and Our Implementation on Two GPU Clusters While Varying the Number of Processing Units (PUs)

		Number of PUs					
	HW	1	2	4	8	16	20
HVAdd	CPU	12.672	7.970	5.798	4.169	3.28	3.45
	K80	0.063	0.031	0.018	**0.024**	-	-
	P100	**0.024**	**0.009**	**0.013**	-	-	-
HVMul	CPU	561.04	289.18	155.37	63.64	65.57	62.81
	K80	5.788	2.602	**1.997**	**6.162**	-	-
	P100	**2.097**	**1.091**	4.135	-	-	-

Experiments with Cingulata were run on K80 *CPU (Intel Xeon E5-2620).*

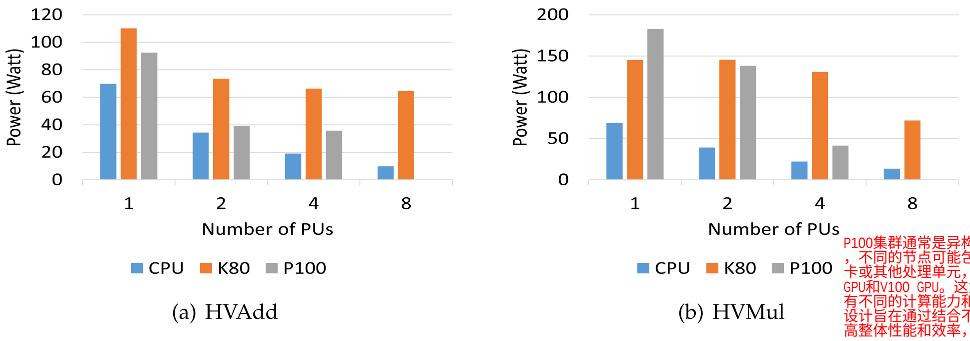
K80:同构集群：每个节点都包含相同类型的GPU卡。
每个节点都具有相似的计算能力和硬件配置，从而简化了
集群管理和任务调度。同构集群通常更容易管理和维护，
因为节点之间的硬件和软件环境是一致的。



(a) HVAdd

(b) HVMul

Fig. 7. Speedup factors for encrypted vector addition and multiplication on different platforms.



(a) HVAdd

(b) HVMul

P100集群通常是异构的。在异构集群中，不同的节点可能包含不同类型的GPU卡或其他处理单元，例如NVIDIA P100 GPU和V100 GPU。这意味着集群中的节点具有不同的计算能力和硬件配置。异构集群的设计旨在通过结合不同类型的处理单元来提高整体性能和效率，以满足各种工作负载的需求。

Fig. 8. Average power consumption per PU for HVAdd and HVMul workloads on CPU, K80, and P100.

involvement of CPU in data exchange as the GPUs are not fully connected in P100 (See Fig. 6). This is not unexpected as this workload contains heavy GPU-to-GPU communication due to the conversion of polynomials partitioning layout. The same applies to K80 when the number of GPUs is 8.

We are also interested in evaluating the scalability of the GPU and CPU implementations. More concretely, we study the increase in parallel speedup as the number of PUs is increased. Fig. 7 shows the scalability of each implementation on different platforms. There are a number of observations that can be drawn from the scalability plot. First, our GPU implementations fail to achieve optimal scalability[4] when the number of PUs is greater than 4. In our distributed memory design, inter-GPU communication overhead is considered the main reason for degrading the scalability of the system. On the other hand, Cingulata - which adopts a shared memory design - includes negligible communication overhead among the PUs. Moreover, as the workloads do not include dependent tasks, all tasks can run concurrently without additional stalls. Therefore, linear scalability can still be observed when the number of CPU cores is greater than 4 in Cingulata. However, as the number of CPU cores is increased further, Cingulata fails to show significant improvement due to the contention on the memory subsystem.

Second, it can be observed that K80 provides generally better scalability compared to P100 when the number of PUs is 4. This is due to the staging through the CPU required in P100 as the GPUs are not fully connected. Note that when the number of GPUs is 8, K80 does not scale due to the amplified communication overhead that is performed via CPU staging.

Third, in some cases we notice superlinear scalability on both K80 and P100 when the number of GPUs is 2. The reason is that distributing the workload on multiple devices can provide more computational resources for thread blocks on each GPU. This results in launching larger number of thread blocks and better utilization of GPU resources.

The results of this experiment demonstrate that our distributed-memory design shows reasonable scalability for a low number of fully connected GPUs. Nevertheless, one can still benefit from the extended memory view our design provides for large computational problems.

To give more insights on power consumption, Fig. 8 shows the average power consumption per PU for HVAdd and HVMul on CPU, K80 and P100. As seen, the CPU platform has much lower power consumption compared to GPUs. The percentage difference in the average power consumption of GPU to CPU for HVAdd (resp. HVMul) ranges from 14 to 556 percent (resp. 88 to 492 percent).

5.3.2 HCNN Workloads

As HCNN − 5 − Layer is computationally less expensive than HCNN − 11 − Layer (See Table 2), we could run it on both GPU clusters. On the other hand, HCNN − 11 − Layer requires at least 88 GB memory, therefore we only could run it on K80 with 8 GPUs. Table 5 shows the latency (in seconds) of running both CNNs on K80, and P100. Note for HCNN − 11 − Layer, the plaintext decomposition is used to accommodate the large plaintext coefficient size (218-bit). We ran the network 10 times, each with different plaintext coefficient t. We show both the per-prime latency and the total running time. Note that these instances are completely independent and can be run concurrently on separate hardware.

It can be noticed that HCNN − 5 − Layer performance improves slightly as the number of GPUs is increased up to

[4] An increase in speedup that is equal to the increase in the number of PUs

TABLE 5
Latency in Seconds of Running HCNN − 5 − Layer and HCNN − 11 − Layer for the MNIST and CIFAR − 10 Image Classification Tasks

# of GPUs	HCNN-5-Layer		HCNN-11-Layer	
	K80	P100	K80	
			Per-prime time	Total time
1	25.25	**14.31**	-	-
2	15.31	**11.77**	-	-
4	**15.84**	22.50	-	-
8	**34.53**	-	1120.71	11207.33

HCNN − 11 − Layer *requires at least 88 GB RAM, hence results obtained on 8 GPUs only.*

4 GPUs except for P100 which requires data staging on CPU. With 8 GPUs, performance degrades on K80 due to CPU involvement in communication. This realistic workload shows that our design scales reasonably for 2 GPUs, and is still acceptable for a larger number of GPUs if they are fully connected.

It is worth noting that our design allowed us to run an ultra large computational task that requires a few millions of homomorphic operations and 88 GB memory without task decomposition, scheduling or load balancing on GPU clusters. Note that the total time of HCNN − 11 − Layer is about 2× faster than Faster CryptoNets [3], which reports 22,372 seconds on Google Cloud Platform n1 − megamem − 96 equipped with 96 Intel Skylake 2.0 GHz vCPUs and 1433.6 GB RAM.

6 Conclusion

In this work, we presented design, implementation and performance evaluation of the HPS RNS variant of the FV levelled FHE scheme on multi-GPU clusters. We showed how to exploit further data parallelism within the FV primitives using data partitioning methods to distribute the workload across available GPUs. It has been shown that our design is ideally suitable for distributed memory architectures that include fast interconnection networks. Moreover, our design is user-friendly as it does not require the user intervention in task decomposition, scheduling or load balancing. A set of experiments have been provided to evaluate the performance of our implementation on homogeneous and heterogeneous GPU clusters. We also compared its performance with a shared-memory multi-core CPU implementation of FV - Cingulata. Our implementation showed significant improvement in performance and achieved speedup factors ranging from 1 to almost 3 orders of magnitude compared to Cingulata. In terms of scalability, we found that neither our implementation nor Cingulata achieves optimal scalability with a high number of processing units due to communication and memory contention.

We also used our implementation to evaluate homomorphically the inference phase of two CNNs (HCNN− 5 − *Layer* and HCNN − 11 − Layer) to classify images from the MNIST and CIFAR − 10 datasets. HCNN − 5 − Layer scaled slightly as the number of GPUs is increased up to 4 GPUs. Although our design was shown not to scale well for 8 GPUs (due to data staging between disconnected GPUs), running HCNN − 11 − Layer on 8 GPUs showed better performance (2×) compared to a recent CPU implementation (Faster CryptoNets) on Google Cloud Platform n1− *megamem* − 96.

As future work, we will try to port our design to support multi-node multi-GPU systems. For these systems, we believe that a hybrid solution that merges task and data parallelism approaches may be the best solution. This can be done by exploiting parallelism at the FHE application level itself. FHE workloads are circuits that can be represented as DAGs of homomorphic gates. A scheduler can be used to partition the DAG into sub-graphs and assign them to the available nodes. The scheduler may include different partitioning granularity to control task-parallelism. Each node in the multi-node multi-GPU system can utilize the data-parallelism designs presented in this work.

Acknowledgments

This work was supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Programmtic Programme (Award A19E3b0099). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the A*STAR.

References

[1] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 169–178.

[2] M. R. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015.

[3] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, “Faster cryptonets: Leveraging sparsity for real-world encrypted inference,” 2018, *arXiv: 1811.09953.*

[4] Y. LeCun, “The MNIST database of handwritten digits,” 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[5] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” Univ. Toronto, Toronto, Canada, Tech. Rep. TR-2009, 2009.

[6] V. Gulshan *et al.*, “Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs,” *J. Amer. Med. Assoc.*, vol. 316, no. 22, pp. 2402–2410, 2016.

[7] W. Dai and B. Sunar, “cuHE: A homomorphic encryption accelerator library,” in *Proc. Int. Conf. Cryptogr. Inf. Secur. Balkans*, 2015, pp. 169–186.

[8] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, “High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA,” *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2018, no. 2, pp. 70–95, 2018.

[9] A. Q. A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, “Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme,” *IEEE Trans. Emerg. Topics Comput.*, to be published, doi: 10.1109/TETC.2019.2902799.

[10] A. Cilardo and D. Argenziano, “Securing the cloud with reconfigurable computing: An FPGA accelerator for homomorphic encryption,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2016, pp. 1622–1627.

[11] S. Sinha Roy, K. Jarvinen, I. Verbauwhede, F. Vercauteren, and J. Vliegen, “HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation,” *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

[12] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy,” in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.

[13] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” *IACR Cryptol. ePrint Archive*, vol. 2012, 2012, Art. no. 144.

[14] S. Halevi, Y. Polyakov, and V. Shoup, “An improved RNS variant of the BFV homomorphic encryption scheme,” in *Cryptographers’ Track RSA Conf.*, 2018, pp. 83–105.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, 2014, Art. no. 13.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2017, pp. 409–437.

[17] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. Annu. Cryptol. Conf.*, 2013, pp. 75–92.

[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[19] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 1–23.

[20] P. Lyubashevsky and Regev,"A toolkit for Ring-LWE cryptography," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2013, vol. 7881, pp. 35–54.

[21] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," *IBM Res.*, vol. 6, pp. 12–15, 2013.

[22] C. Du, G. Bai, and X. Wu, "High-speed polynomial multiplier architecture for Ring-LWE based public key cryptosystems," in *Proc. 26th Edition Great Lakes Symp. VLSI*, 2016, pp. 9–14.

[23] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFLlib: NTT-based fast lattice library," in *Proc. Cryptographers' Track RSA Conf.*, 2016, pp. 341–356.

[24] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Proc. Int. Conf. Cryptol. Netw. Secur.*, 2016, pp. 124–139.

[25] A. Al Badawi, B. Veeravalli, and K. M. M. Aung, "Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution," in *Proc. Future Inf. Commun. Conf.*, 2018, pp. 666–682.

[26] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2016, pp. 423–442.

[27] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 143–163.

[28] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *IACR Cryptol. ePrint Archive*, vol. 2013, 2013, Art. no. 616.

[29] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1509–1521, Jun. 2015.

[30] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.

[31] J. Cathébras, A. Carbon, P. Milder, R. Sirdey, and N. Ventroux, "Data flow oriented hardware design of RNS-based polynomial multiplication for SHE acceleration," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2018, pp. 69–88, 2018.

[32] CryptoExperts, "FV-NFLlib: Library implementing the fan-vercauteren homomorphic encryption scheme," 2016. [Online]. Available: https://github.com/CryptoExperts/FV-NFLlib

[33] T. Rondeau, "Data protection in virtual environments (DPRIVE)," 2020.

[34] Y. Doröz, A. Shahverdi, T. Eisenbarth, and B. Sunar, "Toward practical homomorphic evaluation of block ciphers using PRINCE," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2014, pp. 208–220.

[35] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proc. 3rd Int. Workshop Secur. Cloud Comput.*, 2015, pp. 13–19.

[36] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*. London, U.K.: Newnes, 2012.

[37] N. Corp., "TESLA K80 GPU accelerator board specification," 2015, Accessed: Jan. 19, 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317–001-v05.pdf

[38] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. London, U.K.: Pearson, 2013.

[39] C. Nvidia, "Toolkit documentation," *NVIDIA CUDA Getting Started Guide for Linux*, 2014.

[40] S. Halevi, "HElib: An implementation of homomorphic encryption," 2014. [Online]. Available: https://github.com/shaih/HElib

[41] Microsoft, "SEAL: Simple encrypted arithmetic library," 2014. [Online]. Available: https://www.microsoft.com/en-us/research/project/simple encrypted-arithmetic-library/

[42] R. G. Rohloff Kurt and P. Yuiry, "PALISADE: Lattice encryption software library," 2016. [Online]. Available: https://git.njit.edu/palisade/PALISADE

[43] R. E. Crandall, "Integer convolution via split-radix fast galois transform," *Center for Advanced Computation Reed College 1999*, Accessed: Jan. 2019. [Online]. Available: https://www.reed.edu/physics/faculty/crandall/papers/confgt.pdf

[44] G. E. Box *et al.*, "A note on the generation of random normal deviates," *The Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 1958.

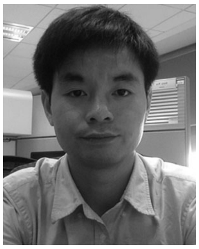
[45] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," AT&T Labs. 2010. [Online]. Available: http://yann.lecun. com/exdb/mnist

[46] A. A. Badawi *et al.*, "The AlexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs," *IEEE Trans. Emerg. Topics Comput.*, to be published, doi: 10.1109/TETC.2020.3014636.

[47] E. Barker and Q. Dang, "NIST special publication 800–57 Part 1, Revision 4," *Recommendation for Key Management*, 2016, Accessed: Feb. 2019. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4

Ahmad Al Badawi received the BSc degree in computer engineering, from Al-Balqa' Applied University, Jordan, in 2007, the MSc degree in computer engineering from JUST, Jordan, in 2010, and the PhD degree in computer engineering from The National University of Singapore, Singapore, in 2018. He is currently a research scientist with the Institute of Infocomm Research, A*STAR, Singapore. His research interests include but not limited to modern cryptography, homomorphic encryption, privacy-preserving technologies, parallel processing, combinatorial optimization, evolutionary algorithms, and multiprocessor task scheduling.

Bharadwaj Veeravalli (Senior Member, IEEE) received the BSc degree in physics, from Madurai-Kamaraj University, India, in 1987, the master's degree in electrical communication engineering from the Indian Institute of Science, Bangalore, India, in 1991, and the PhD degree from the Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India, in 1994. He is currently with the Department of Electrical and Computer Engineering, The National University of Singapore, Singapore, as a tenured associate professor. His main stream research interests include, cloud/grid/cluster computing, scheduling in parallel and distributed systems, bioinformatics and computational biology, and multimedia computing.



Jie Lin received the BS and PhD degree from the School of Computer Science and Technology, Beijing Jiaotong University, China. He is a research scientist with the Institute of Infocomm Research, A*STAR, Singapore. His research interests include deep learning and computer vision. His work on image feature coding has been recognized as core contribution to the MPEG-7 Compact Descriptors for Visual Search (CDVS) standard. He also works on software-hardware co-optimization for deep learning and build ASIC accelerators from scratch for applications on edge.

Xiao Nan received the master's degree of computer application technology from the University of Science and Technology, Beijing, China, in 2008. He is currently a senior research engineer with A*STAR, Institute for Infocomm Research, Singapore. His research interests include data and information security, data centres, and high performance computing.

Kazuaki Matsumura received the BE degree from the University of Tsukuba, Japan, in 2017 and the MSc degree from the Tokyo Institute of Technology, Japan, in 2019. He is currently affiliated with a doctoral program at Barcelona Supercomputing Center. His research interests include program optimization and compilers for high-performance computing.

Khin Mi Mi Aung (Senior Member, IEEE) received the PhD degree in computer engineering from the Korea Aerospace University, Seoul, Korea. She is currently a senior research scientist with the Institute of Infocomm Research, A*STAR, Singapore. Her current research interests include privacy-preserving and data security technologies.