The Dissertation Committee for Roshan Dathathri
certifies that this is the approved version of the following dissertation:

Compiler and Runtime Systems for Homomorphic Encryption and Graph Processing on Distributed and Heterogeneous Architectures

Committee:

Keshav Pingali, Supervisor

Madanlal Musuvathi

Vijaya Ramachandran

Christopher Rossbach

Marc Snir

Compiler and Runtime Systems for Homomorphic Encryption and Graph Processing on Distributed and Heterogeneous Architectures

by

Roshan Dathathri

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
May 2020

Dedicated to my wife Priya.

Acknowledgments

First and foremost, I would like to thank my advisor, Keshav Pingali, for his invaluable guidance. Keshav is a great mentor who helped me think in terms of abstractions and taught me to effectively communicate ideas. I am inspired by Keshav's approach to research – his curiosity in exploring problems, his rigor in solving them, and his attention to detail in understanding problems or solutions. His positive attitude has boosted my morale. Keshav gave me the opportunity to work with and mentor an exceptional team in University of Texas (UT) Austin. Furthermore, he was instrumental in setting up my collaborations with groups in University of Illinois Urbana-Champaign, Microsoft Research (MSR), Intel, and BAE Systems.

I would like to thank all my dissertation committee members for their support and advice. Madanlal Musuvathi is a great mentor. He helped me ask the right questions and focus on the key problems. Marc Snir and Vijaya Ramachandran offered perspectives on low-level communication runtimes and high-level graph algorithms respectively. This influenced my work on graph processing which bridges the gap between those interfaces. Their questions and feedback helped improve my work as well as my communication. Christopher Rossbach was always accessible and helpful.

Research is a team effort and I have been fortunate to collaborate with

researchers both in industry and in academia working on areas from cryptography to computer architecture. Andrew Lenharth and Gurbinder Gill were part of the initial team working on distributed graph processing. The infrastructure that we built together was the basis for our research in graph processing. In addition to the three of us, the distributed and heterogeneous graph processing project was made possible by an outstanding team which included: Loc Hoang, Vishwesh Jatala, Xuhao Chen, Hochan Lee, Bozhi You, Hoang-Vu Dang, Alex Brooks, and Nikoli Dryden, Krishna Nandivada, and Ramesh Peri. I would also like to thank Sreepathi Pai for his mentorship and guidance, especially during the initial part of my work on heterogeneous graph analytics. I spent two delightful, invigorating, instructive, and fruitful summers at MSR. Kim Laine, Wei Dai, and Hao Chen helped me understand the intricacies of Fully-Homomorphic Encryption (FHE). Olli Saarikivi helped me get acquainted with FHE and develop FHE applications in a very short period of time. I would like to thank Saeed Maleki and Todd Mytkowicz for their support and encouragement. I would also like to thank Kristin Lauter for giving me an opportunity to intern in her team and get the perspective of cryptographers.

I want to thank all the members of the Intelligent Software Systems group at UT Austin for maintaining a culture that is stimulating and refreshing. It has been a pleasure to work in such an environment. I want to especially thank Julie Heiland for organizing all the team lunches and managing our administrative work. I would also like to thank all the non-technical staff at UT

Austin for ensuring that the time I spent at UT was hassle-free.

It was a privilege to study at UT Austin. I would like to thank UT for providing advanced computational resources, subsidized housing, convenient health services, and excellent recreational facilities. I volunteered for Association for India's Development (AID) during my time at UT and helped lead the Run For India (RFI) initiative in Austin. Both activities were a crucial part of my life at UT and I would like thank AID and RFI volunteers for the cheerful, motivational, and purposeful experiences.

Finally, I would like to thank my friends and family for all their support and encouragement. My parents, Rajeshwari and Dathathri, provided me with excellent education since childhood. I am grateful for my brother and his wife, Ravikiran and Padmashree, for their love, support, and advice. Watching my niece, Arya, grow up was joyful and uplifting. I would especially like to thank my wife, Priyadarshini, for her patience, support, and encouragement. Living with her made me more confident, more relaxed, and more productive. She was the foundation that made this thesis possible.

Compiler and Runtime Systems for Homomorphic Encryption and Graph Processing on Distributed and Heterogeneous Architectures

Publication No. _____

Roshan Dathathri, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Keshav Pingali

Distributed and heterogeneous architectures are tedious to program because devices such as CPUs, GPUs, and FPGAs provide different programming abstractions and may have disjoint memories, even if they are on the same machine. In this thesis, I present compiler and runtime systems that make it easier to develop efficient programs for privacy-preserving computation and graph processing applications on such architectures.

Fully Homomorphic Encryption (FHE) refers to a set of encryption schemes that allow computations on encrypted data without requiring a secret key. Recent cryptographic advances have pushed FHE into the realm of practical applications. However, programming these applications remains a huge challenge, as it requires cryptographic domain expertise to ensure correctness,

security, and performance. This thesis introduces a domain-specific compiler for fully-homomorphic deep neural network (DNN) inferencing as well as a general-purpose language and compiler for fully-homomorphic computation:

1. I present CHET, a domain-specific optimizing compiler, that is designed to make the task of programming DNN inference applications using FHE easier. CHET automates many laborious and error prone programming tasks including encryption parameter selection to guarantee security and accuracy of the computation, determining efficient data layouts, and performing scheme-specific optimizations. Our evaluation of CHET on a collection of popular DNNs shows that CHET-generated programs outperform expert-tuned ones by an order of magnitude.

2. I present a new FHE language called Encrypted Vector Arithmetic (EVA), which includes an optimizing compiler that generates correct and secure FHE programs, while hiding all the complexities of the target FHE scheme. Bolstered by our optimizing compiler, programmers can develop efficient general-purpose FHE applications directly in EVA. EVA is designed to also work as an intermediate representation that can be a target for compiling higher-level domain-specific languages. To demonstrate this, we have re-targeted CHET onto EVA. Due to the novel optimizations in EVA, its programs are on average $\sim 5.3\times$ faster than those generated by the unmodified version of CHET.

These languages and compilers enable a wider adoption of FHE.

Applications in several areas like machine learning, bioinformatics, and security need to process and analyze very large graphs. Distributed clusters are essential in processing such graphs in reasonable time. I present a novel approach to building distributed graph analytics systems that exploits heterogeneity in processor types, partitioning policies, and programming models. The key to this approach is Gluon, a domain-specific communication-optimizing substrate. Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run on heterogeneous clusters in the bulk-synchronous parallel (BSP) model and optimizes communication in a novel way by exploiting structural and temporal invariants of graph partitioning policies. We also extend Gluon to support lock-free, non-blocking, bulk-asynchronous execution by introducing the bulk-asynchronous parallel (BASP) model. Our experiments were done on CPU clusters with up to 256 multi-core, multi-socket hosts and on multi-GPU clusters with up to 64 GPUs. The communication optimizations in Gluon improve end-to-end application execution time by $\sim 2.6\times$ on the average. Gluon's BASP-style execution is on average $\sim 1.5\times$ faster than its BSP-style execution for graph applications on real-world large-diameter graphs at scale. The D-Galois and D-IrGL systems built using Gluon scale well and are faster than Gemini, the state-of-the-art distributed CPU-only graph analytics system, by factors of $\sim 3.9\times$ and $\sim 4.9\times$ on average using distributed CPUs and distributed GPUs respectively. The Gluon-based D-IrGL system for distributed GPUs is also on

average $\sim 12\times$ faster than Lux, the only other distributed GPU-only graph analytics system. The Gluon-based D-IrGL system was one of the first distributed GPU graph analytics systems and is the only asynchronous one.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Heterogeneous architectures are being designed to improve the performance of applications. However, these architectures are tedious to program because of their heterogeneity, as devices such as CPU, GPU, FPGA, and TPU provide different programming abstractions and may have disjoint memories. Moreover, these heterogeneous platforms are being interconnected on the cloud or distributed clusters. While such distributed platforms can give better performance, they require even more programming effort to extract performance.

The datasets used by applications are growing in size and are being deployed on public clouds. However, many datasets may need to be private and deploying on it on public clouds requires encrypting the data. This makes it difficult for programmers to write applications to operate on sensitive data that execute on untrusted clouds. On the other hand, large datasets are usually sparse due to the locality of interaction. The sparsity needs to be exploited by the programmer to make the application efficient.

Applications are written by experts in the application domain rather than experts in parallel programming. Consequently, it is hard for program-

mers to: (1) write efficient code for each device, (2) distribute execution across devices, (3) orchestrate communication between devices, and (4) port code to new emerging devices. They may also need to manage the privacy of the datasets used by applications or exploit the sparsity in the datasets. To address these problems, compilers and runtime systems can take a common domain-specific interface for all devices and generate efficient architecture-specific code for each device, distribute computation among devices, and move data between them efficiently. Two major challenges for building such compiler and runtime systems are to provide abstractions for privacy and sparsity of the datasets.

In this thesis, I present novel techniques and optimizations in compilers and runtime systems that exploit domain-specific knowledge for privacy-preserving computing and sparse computing applications. The compiler and runtime systems introduced in this thesis not only make it easy to develop applications that operate on sensitive data or sparse data but also enable those applications to run efficiently, yielding high performance with high productivity. In privacy-preserving computing, the focus of this thesis is on applications that use Fully Homomorphic Encryption (Section 1.1) and in sparse computing, the focus of this thesis is on graph processing applications (Section 1.2).

1.1 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) [64] refers to a set of encryption schemes [26, 42, 59] that allow computations on encrypted data without requiring a secret key. Recent cryptographic advances [41, 42] have pushed FHE

into the realm of practical applications. Despite the availability of multiple open-source FHE libraries [81, 92, 138, 149], programming these applications remains a huge challenge, as it requires cryptographic domain expertise to ensure correctness, security, and performance.

This thesis introduces CHET, a domain-specific optimizing compiler, that is designed to make the task of programming FHE applications easier. Motivated by the need to perform neural network inference on encrypted medical and financial data, CHET supports a domain-specific language for specifying tensor circuits. It automates many of the laborious and error prone tasks of encoding such circuits homomorphically, including encryption parameter selection to guarantee security and accuracy of the computation, determining efficient tensor layouts, and performing scheme-specific optimizations.

Our evaluation of CHET on a collection of popular neural networks shows that CHET generates homomorphic circuits that outperform expert-tuned circuits and makes it easy to switch across different encryption schemes. We demonstrate its scalability by evaluating it on a version of SqueezeNet, which to the best of our knowledge, is the deepest neural network to be evaluated homomorphically.

To further improve the developer friendliness of FHE, this thesis proposes a new general-purpose language for FHE computation called Encrypted Vector Arithmetic (EVA). EVA is also designed to be an intermediate representation that is a backend for other domain-specific compilers. At its core, EVA supports arithmetic on fixed-width vectors and scalars. The vector in-

structions naturally match the encrypted SIMD – or batching – capabilities of FHE schemes today. EVA includes an optimizing compiler that hides all the complexities of the target FHE scheme, such as encryption parameters and noise. It ensures that the generated FHE program is correct, performant, and secure. In particular, it eliminates all common runtime exceptions that arise when using FHE libraries.

To demonstrate EVA's usability, we have built a Python frontend for it. Using this frontend, we have implemented several applications, including image processing applications, in EVA with very few lines of code and much lesser complexity than in FHE libraries directly. In addition, we have re-targeted CHET onto EVA. Due to the novel optimizations in EVA, its programs are on average $\sim 5.3\times$ faster than those generated by the unmodified version of CHET.

1.2 Graph Processing

Several applications exist that analyze and operate on unstructured data. The data in such applications can be represented as graphs, and graph algorithms can be used to analyze such datasets. Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges, or the indexable Web graph, which has roughly 100 billion nodes and trillions of edges. Such systems can be used in programming graph analytics applications like ranking web pages in search engines [136], finding clusters in large scale biological networks [161],

evaluating recommender systems [15, 82], finding shortest routes in maps, and analyzing power grid component failures [97].

This thesis introduces a novel approach to building distributed graph analytics systems that exploits heterogeneity in processor types (CPU and GPU), partitioning policies, and programming models. The key to this approach is Gluon, a communication-optimizing substrate. Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run on heterogeneous clusters in the bulk-synchronous parallel (BSP) model [159] and optimizes communication in a novel way by exploiting structural and temporal invariants of graph partitioning policies. We also extend Gluon to support lock-free, non-blocking, bulk-asynchronous execution by introducing the bulk-asynchronous parallel (BASP) model.

Our experiments were done on CPU clusters with up to 256 multi-core, multi-socket hosts and on multi-GPU clusters with up to 64 GPUs. The communication optimizations in Gluon improve end-to-end application execution time by $\sim 2.6\times$ on the average. Gluon's BASP-style execution is on average $\sim 1.5\times$ faster than its BSP-style execution for graph applications on real-world large-diameter graphs at scale. The D-Galois and D-IrGL systems built using Gluon scale well and are faster than Gemini [173], the state-of-the-art distributed CPU-only graph analytics system, by factors of $\sim 3.9\times$ and $\sim 4.9\times$ on average using distributed CPUs and distributed GPUs respectively. The Gluon-based D-IrGL system for distributed GPUs is also on average $\sim 12\times$

faster than Lux [95], the only other distributed GPU-only graph analytics system. The Gluon-based D-IrGL system was one of the first distributed GPU graph analytics systems and is the only asynchronous one.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 describes my work on CHET [56], which compiles tensor programs to run on homomorphically encrypted data. Chapter 3 describes my work on EVA [55], which is a general-purpose language and compiler for computation on homomorphically encrypted data. Chapters 4 and 5 present my work on Gluon [52] and Gluon-Async [53] respectively, which are communication substrates for distributed and heterogeneous graph analytics. Finally, Chapter 6 discusses conclusions and future work.

Chapter 2

CHET: An Optimizing Compiler and Runtime for Fully-Homomorphic Neural-Network Inferencing

Fully Homomorphic Encryption (FHE) refers to a set of encryption schemes that allow computations on encrypted data without requiring a secret key. Recent cryptographic advances have pushed FHE into the realm of practical applications. However, programming these applications remains a huge challenge, as it requires cryptographic domain expertise to ensure correctness, security, and performance.

CHET[1] is a domain-specific optimizing compiler designed to make the task of programming FHE applications easier. Motivated by the need to perform neural network inference on encrypted medical and financial data, CHET supports a domain-specific language for specifying tensor circuits. It automates many of the laborious and error prone tasks of encoding such circuits

[1]My main contributions to this work include the design and implementation of the compiler and runtime as well as the compiler optimizations. The full citation of the published version of this work is: "Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pages 142–156, New York, NY, USA, 2019. ACM".

homomorphically, including encryption parameter selection to guarantee security and accuracy of the computation, determining efficient tensor layouts, and performing scheme-specific optimizations.

Our evaluation on a collection of popular neural networks shows that CHET generates homomorphic circuits that outperform expert-tuned circuits and makes it easy to switch across different encryption schemes. We demonstrate its scalability by evaluating it on a version of SqueezeNet, which to the best of our knowledge, is the deepest neural network to be evaluated homomorphically.

2.1 Introduction

Fully Homomorphic Encryption (FHE) provides an exciting capability of performing computation on encrypted data without requiring the decryption key. This holds the promise of enabling rich privacy-preserving applications where the clients offload their data storage and computation to a public cloud without having to trust either the cloud software vendor, the hardware vendor, or a third party with their key.

The first FHE scheme was proposed by Gentry et al. [64] in 2009. While a theoretical breakthrough, a direct implementation of this scheme was considered impractical. Cryptographic innovations in the past decade have since made significant progress in both performance and supporting richer operations. Original FHE schemes only supported Boolean operations [64]. Subsequent schemes [26, 59] supported integer operations, thereby avoiding the need to encode arithmetic operations as Boolean circuits. Recently, Cheon et al. [41, 42] proposed an FHE scheme that efficiently supports fixed-point arithmetic extending the reach of FHE to domains such as machine learning. Together with these innovations, optimized open-source implementations of FHE schemes, such as SEAL [148] and HEAAN [92], have made FHE more accessible.

Nevertheless, building effective FHE applications today requires direct involvement of a cryptographic expert intricately familiar with the encryption schemes. Current FHE schemes work by introducing noise during encryption that is subsequently removed during decryption. The amount of noise intro-

duced during encryption and each intermediate operation is controlled by a set of encryption parameters that are set manually. Setting these parameters low can make the encryption insecure. On the other hand, setting them large can increase the size of encrypted text and increase the cost of homomorphic operations. Moreover, when the accumulated noise exceeds a bound determined by these parameters, the encrypted result becomes corrupted and unrecoverable.

As individual homomorphic operations are orders of magnitude more expensive than equivalent plaintext operations, amortizing the cost of individual FHE operations requires utilizing the vectorization capabilities (also called as "batching" in the FHE literature) of the encryption schemes. Different ways of mapping application parallelism onto ciphertext vectors can result in different circuits each of which require a subsequent retuning of encryption parameters for correctness, security, and performance. As a result, developing FHE applications today is laboriously hard.

In many ways, programming FHE applications today is akin to low-level programming against a custom hardware with counter-intuitive tradeoffs. As such, our hypothesis is that a compiler that raises the programming abstraction while hiding and automating many of the manual tasks can make FHE programming easier and scalable. More importantly, by systematically exploring the various performance tradeoffs, a compiler can generate far more efficient code than those produced manually by experts.

This thesis evaluates this hypothesis with CHET, a compiler for homomorphic tensor programs. The input domain is primarily motivated by the

need to perform neural-network inference on privacy-sensitive data, such as medical images and financial data. For privacy reasons, these applications are performed on-premise today. Offloading the storage and computation to a cloud provider would not only simplify the operational cost of maintaining on-premise clusters but also dramatically reduce the data-management cost of protecting sensitive data from unauthorized accesses both from within and outside the organization. Thus, FHE enables an attractive way to move these applications to the cloud without enlarging the trust domain beyond the organization owning the data.

Given a tensor circuit, CHET compiles the circuit into an executable that can be linked with FHE libraries such as SEAL [41, 148] or HEAAN [42, 92]. The compiler uses a cost model of homomorphic operations to systematically search over different ways of mapping input tensors into FHE vectors. For each choice, the compiler analyzes the resulting circuit to determine the encryption parameters that maximize performance while ensuring security and correctness. During this process, CHET additionally performs scheme-specific optimizations to increase the end-to-end performance.

Apart from the compiler, CHET includes a runtime, akin to the linear algebra libraries used in unencrypted evaluation of neural networks. We have developed a set of layouts and a unified metadata representation for them. For these new layouts, we have developed a set of computational kernels that implement the common operations found in convolutional neural networks (CNNs). All of these kernels were designed to use the vectorization capabilities

of modern FHE schemes.

Schemes that support fixed-point arithmetic [41, 42] do so by scaling fixed-point numbers to integers. Determining the scaling factors to use for the inputs and the output is difficult as it involves a tradeoff between performance and output precision. CHET simplifies this choice with a profile-guided optimization step. Given a set of test inputs, CHET automatically determines the fixed-point scaling parameters that maximize performance while guaranteeing the desired precision requirements of the output for these test inputs.

We evaluate CHET with a set of real-world CNN models and show how different optimization choices available in our compiler can significantly improve inference latencies. As an example, for a neural network obtained from an industry partner for medical imaging, the base implementation took more than 18 hours per image, while a FHE expert was able to bring this to under 45 minutes with a hand-tuned circuit. By systematically searching over a wider set of possible optimizations, CHET generated a circuit that took less than 5 minutes per image. Moreover, CHET was able to easily port the same input circuit to a more recent and efficient FHE scheme that is harder to hand tune. Our port took less than a minute per image. CHET is also able to scale to large neural networks, such as SqueezeNet. To the best of our knowledge, this is the deepest neural network to be homomorphically evaluated.

In summary, our main contributions are as follows:

- A compiler and runtime called CHET for homomorphic evaluation of

tensor circuits that can be used for neural network inference.

- A set of data layouts and a metadata format for packing tensors into ciphertexts.
- An analysis and transformation framework for adding new optimizations and new target FHE schemes easily.
- Analysis to choose the encryption parameters and data layouts for SEAL [41, 148] and HEAAN [42, 92].
- A profile-guided optimization to tune the fixed-point scaling parameters for the inputs and the output.
- Multiple layout-specific kernel implementations of homomorphic tensor operations.
- An evaluation of CHET on a set of real-world CNN models, including the homomorphic evaluation of the deepest CNN to date.

The rest of this chapter is organized as follows. Section 2.2 introduces homomorphic encryption and tensor programs. Section 2.3 provides an overview of using CHET with an example. Section 2.4 describes the intermediate representations and Section 2.5 describes our runtime. Section 2.6 describes our compiler. Section 2.7 presents our evaluation of CHET. Section 2.8 describes related work.

2.2 Background

This section provides background about FHE that is necessary to understand the contributions underlying CHET. Interested readers can look at [8] for more details.

2.2.1 Homomorphic Encryption

Say a plaintext message m is encrypted into a ciphertext $\langle m \rangle$ by an encryption scheme. This encryption scheme is said to be *homomorphic* with respect to an operation $\oplus$ if there exists an operation $\langle \oplus \rangle$ such that

$$\langle a \rangle \langle \oplus \rangle \langle b \rangle = \langle a \oplus b \rangle$$

for all messages a and b. For example, the popular encryption scheme RSA is homomorphic with respect to multiplication but not with respect to addition.

An encryption scheme is *fully homomorphic* (FHE) if it is homomorphic with respect to a set of operators that are sufficient to encode arbitrary computations. Current FHE schemes are *levelled* (also called as *somewhat homomorphic*) in that for fixed encryption parameters they only support computation of a particular depth.[2] In this thesis, we will only deal with levelled homomorphic schemes as the size of the input circuit is known before hand.

[2] A levelled scheme may be turned into a fully homomorphic one by introducing a *bootstrapping* operation [64].

2.2.2 Integer FHE Schemes with Rescaling

Early FHE schemes only supported Boolean operations. However, recent FHE schemes [26, 59] directly support integer *addition* and *multiplication* operations. This allows us to evaluate arbitrary arithmetic circuits efficiently without having to "explode" them into Boolean circuits. In addition, these FHE schemes also support optimized *constant* operations when one of the operand is a plaintext message.

Many applications, such as machine learning, require floating point arithmetic. To support such applications using integer FHE schemes, we can use fixed-point arithmetic with an explicit scaling factor. For instance, we can represent $\langle 3.14 \rangle$ as $\langle 314 \rangle$ with scale 100. However, this scaling factor quickly grows with multiplication. The resulting *plaintext-coefficient growth* [109] limits the size of the circuits one can practically evaluate. The recently proposed integer FHE scheme CKKS [42] enables *rescaling* ciphertexts — allowing one to convert say $\langle 2000 \rangle$ at fixed-point scale 100 to $\langle 20 \rangle$ at scale 1. This mitigates the problem of growing scaling factors. This thesis only focusses on FHE schemes with rescaling support, namely the CKKS scheme [42] and its variant RNS-CKKS [41]. Obviously, CHET can trivially target other FHE schemes such as FV [59] or BGV [26], but this is not the main focus of the thesis.

Note that many applications, including machine learning, use non-polynomial operations such as exp, log, and tanh. We will assume that such functions are appropriately approximated by polynomials before the circuit is provided to CHET. Prior work [67] has already shown that this is feasible for

Table 2.1: The asymptotic costs of homomorphic operations for the CKKS and RNS-CKKS scheme variants in HEAAN v1.0 and SEAL v3.1 respectively. $M(Q)$ is the complexity of multiplying large integers and is $O(\log^{1.58} Q)$ for HEAAN.

Homomorphic Operation	CKKS	RNS-CKKS with $Q = \Pi_{i=1}^{r} Q_i$
addition, subtraction	$O(N \cdot \log Q)$	$O(N \cdot r)$
scalar multiplication	$O(N \cdot M(Q))$	$O(N \cdot r)$
plaintext multiplication	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot r)$
ciphertext multiplication	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot \log N \cdot r^2)$
ciphertext rotation	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot \log N \cdot r^2)$

machine learning.

2.2.3 Encryption Parameters

One of the essential tasks CHET performs is to automate the selection of encryption parameters that determine the correctness, security, and performance of the FHE computation. In both the CKKS and RNS-CKKS schemes, a ciphertext is a polynomial of degree N with each coefficient represented modulo Q. N is required to be a power of two in both schemes. While Q is a power of two in the CKKS scheme, Q is a product of r primes $Q = \Pi_{i=1}^{r} Q_i$ in the RNS-CKKS scheme. Table 2.1 shows the asymptotic cost of homomorphic operations for the CKKS and RNS-CKKS schemes implemented in HEAAN v1.0 [92] and SEAL v3.1 [148] respectively. Larger values of N and Q (or r) increase the cost of homomorphic operations. Note that although r can be much smaller than $\log(Q)$, real world performance should not be directly inferred from the asymptotic complexities, as implementations

of the two schemes can have very different constants. Finally, the size of the encrypted messages grows with N.

While the performance constraints above require N and Q to be as small as possible, they have to be reasonably large for the following reasons. First, the parameter Q has to be large enough to correctly evaluate a given circuit. As the coefficients in the ciphertext polynomial are represented modulo Q, any coefficient that exceeds Q will result in an overflow, causing the message to be corrupted and unrecoverable. To avoid this overflow, the computation should periodically rescale the values as described in Section 2.2.2. But this rescaling "consumes" the modulus Q resulting in a polynomial with a smaller modulus. Thus, the computation has to perform a delicate balance between introducing sufficient rescaling to limit the growth of coefficient values and ensuring large-enough modulus to represent the output correctly. As multiplications are the primary reason for the growth of coefficients, Q directly limits the *multiplicative depth* of the circuit one can safely evaluate.

Additionally, for a given Q, larger values of N make the encryption harder to attack. The *security level* of an encryption is usually measured in bits, where n-bit security implies that a brute-force attack is expected to require at least 2^n operations. The security level for a given Q and N is a table provided by the encryption scheme [36] which CHET explicitly encodes. By default, CHET chooses the smallest values of N and Q that guarantee 128-bit security.

2.2.4 FHE Vectorization

A unique capability of FHE schemes is the ability to support large *single instruction multiple data* (SIMD) vectors. These schemes, for appropriate setting of parameters, enable one to encode multiple integers into a larger integer and use the Chinese Remainder Theorem (see [109] for details) to simultaneously perform operations on individual integers by performing a single operation on the larger integer. When compared to SIMD capabilities of current hardware processors, these SIMD widths are large — vector sizes of tens of thousands or more are not uncommon. In particular, the SIMD width in CKKS and RNS-CKKS is $N/2$.

FHE schemes naturally support vector addition and multiplication. In addition, they support *rotation* operations that mimic the shuffle instructions of the SIMD units of modern processors. For instance, rotating a vector $[a_1, a_2, \ldots, a_n]$ by a constant i results in the vector $[a_{i+1}, \ldots, a_n, a_1, \ldots, a_i]$. However, FHE schemes do not support random access to extract a particular slot of a vector. Such operations need to be implemented by multiplying the vector with a plaintext mask followed by a rotation. This unfortunately adds to the multiplicative depth and should be avoided when possible.

Rotating a vector by an amount i requires a public rotation key that is specific to the constant i [42]. Given the large vector widths, it is impractical to generate a rotation key for every possible i. Instead, FHE libraries usually generate a public rotation key for every power-of-2 and then use multiple rotations to achieve the desired amount of rotation. CHET optimizes this by

generating public rotation keys explicitly for a given input circuit.

2.2.5 Approximation in CKKS

As discussed above, FHE schemes introduce noise during homomorphic operations. Unlike other schemes, CKKS and RNS-CKKS are *approximate* and introduce noise in the lower-order bits of the message. To ensure that this noise does not affect the precision of the output, these schemes require that the inputs and the output be scaled by large-enough values. CHET includes a profile-guided optimization to determine these scaling factors automatically, given a set of test inputs.

2.2.6 Tensor Programs

A tensor is a multidimensional array with regular dimensions. A tensor t has a data type $\text{dtype}(t)$, which defines the representation of each element, and a shape $\text{shape}(t)$, which is a list of the tensor's dimensions. For example, a single 32 by 32 image with 3 channels of color values between 0 and 255 could be represented by a tensor I with $\text{dtype}(I) = \texttt{int8}$ and $\text{shape}(I) = [3, 32, 32]$.

In machine learning, neural networks are commonly expressed as programs operating on tensors. Some common tensor operations in neural networks include:

Convolution calculates a cross correlation between an input and a filter tensor.

Matrix multiplication represents neurons that are connected to all inputs, which are found in dense layers typically at the end of a convolutional neural network.

Pooling combines adjacent elements using a reduction operation such as maximum or average.

Element-wise operations such as batch normalization and ReLUs.

Reshaping reinterprets the shape of a tensor, for example, to flatten preceding a matrix multiplication.

We consider the tensor program as a circuit of tensor operations and this circuit is a Directed Acyclic Graph (DAG).

2.3 Overview of CHET

Developing an FHE application involves many challenges. CHET is an optimizing compiler designed to alleviate many of the issues. This section provides the overview of the compiler and how it aids in the development of FHE applications.

2.3.1 Motivating Example

We first motivate CHET with a simple example of homomorphically performing matrix-matrix multiplication. Figure 2.1 shows two 2×2 matrices A and B. We need to perform a total of 8 multiplications. The obvious way

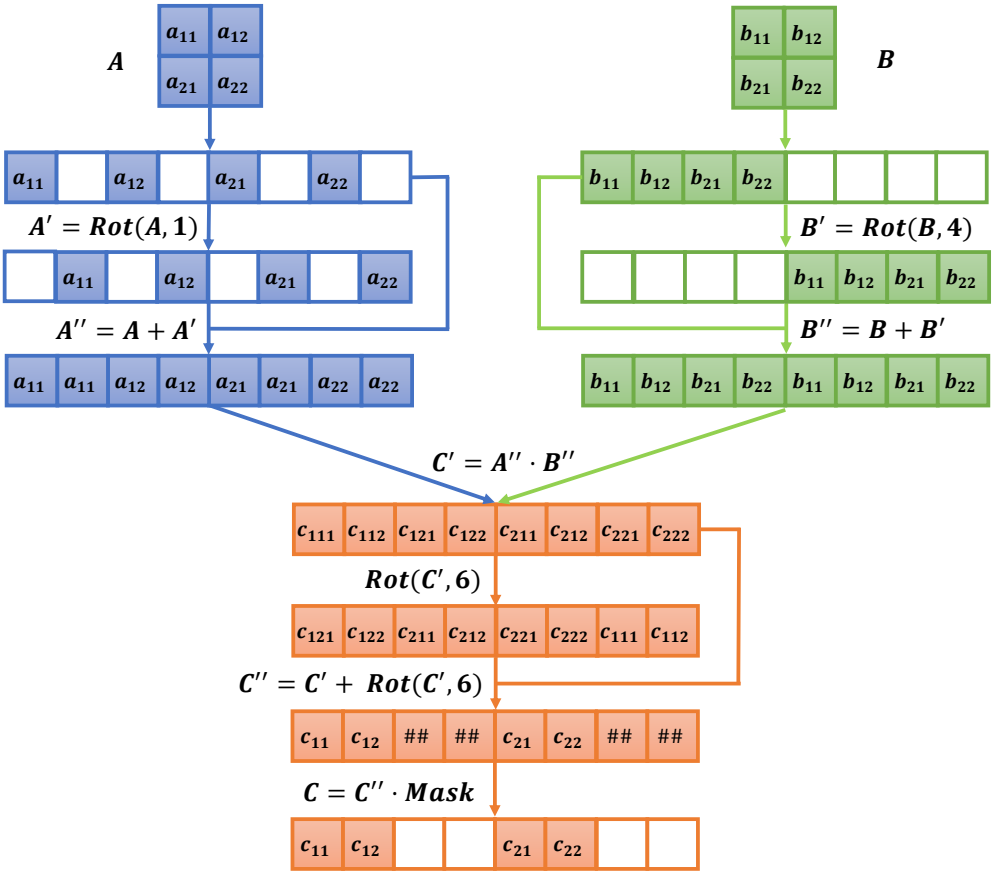


Figure 2.1: Homomorphic matrix-matrix multiplication.

is to encrypt the scalar values of A and B individually and perform matrix-matrix multiplication the normal way. This is not a viable solution for large matrices due to the cost of FHE operations.

Instead, we would like to make use of the large SIMD capabilities provided by FHE schemes. This requires us to map the values onto vectors using specific *layouts*. In Figure 2.1, B is in the standard row-major format, but A's layout contains some padding. As shown in the figure, this special layout allows us to replicate the values twice, using rotations and additions, such that all the 8 products we need $c_{ijk} = a_{ij} \cdot b_{jk}$ can be obtained with 1 FHE multiplication in the C' vector.

We need additional rotation and addition operations to compute the elements of the output matrix $c_{ik} = \sum_j c_{ijk}$ in C''. This vector contains additional junk entries marked ##, which are masked out to produce the result C. Note that the layout of C is different from that of the input matrices A and B.

The key challenge here is that operations such as masking and rotations which are relatively cheap in plaintext SIMD operations are expensive in FHE. For instance, rotating C' by 6 would either require a special rotation key for 6 or we have to use two rotations using the power-of-2 keys for 4 and 2 generated by default by the underlying encryption schemes. Equally importantly, masking operations such as the one required to obtain C from C'' involve a multiplication that adds to the multiplicative depth.

Thus, when matrix C is used in a subsequent matrix multiplication, rather than converting it into a standard layout of A or B, we can emit a different set of instructions that are specific to the layout of C. Doing this manually while managing the different layouts of the variables in the program can soon become overwhelming and error prone.

While the simple example above already brought out the complexity of FHE programming, performing neural network inferencing brings out many more. First, we have to deal with multi-dimensional tensors where the choices of layouts are numerous. Moreover, useful machine learning models have large tensors that need not fit within the SIMD widths of the underlying schemes when using standard parameters. Thus, one has to deal with the trade-off between increasing the SIMD widths (by increasing the N parameter) or splitting tensors into multiple ciphertexts.

As we have seen above, layout decisions can change the operations one needs to perform which can in turn affect the multiplicative depth required to execute the circuit. However, setting the encryption parameters to allow the required multiplicative depth in turn changes the SIMD widths available which of course might require changes in the layout. Making these interdependent choices manually is a hard problem. CHET is designed to automatically explore these choices.

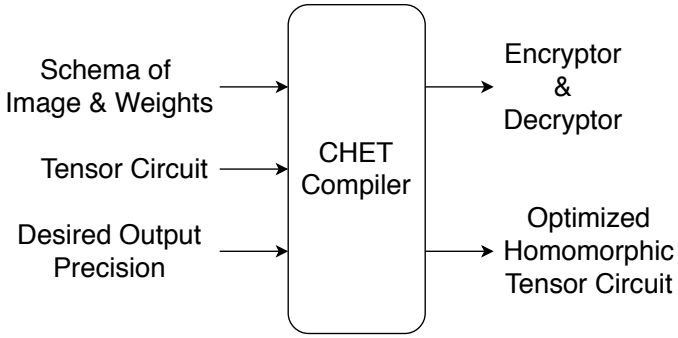


Figure 2.2: Overview of the CHET system at compile-time.

2.3.2 Using the Compiler

We will assume that the neural network inferencing computation is specified as a sequence of tensor operations that we call a tensor circuit. This input is very similar to how these models are specified in frameworks such as TensorFlow [6].

Figure 2.2 shows how an application programmer can use CHET to compile a tensor circuit. Consider a tensor circuit with a single operation:

$$output = conv2d(image, weights); \tag{2.1}$$

In addition to the tensor circuit, CHET requires the schema of the inputs to the circuit. The input schema specifies the tensor dimensions as well as the fixed-point scales to use for the tensor values. In Equation 2.1 for example, the user specifies that the input "image" (i) is encrypted, (ii) is a 4-dimensional tensor of size $1 \times 1 \times 28 \times 28$, and (iii) has a fixed-point scaling factor of 2^{40}. CHET also requires the desired fixed-point scales for the output (output precision).

Neural network models typically use floating-point values and determining the right fixed-point scaling factors to use is not straight-forward, especially given the approximation present in CKKS. Larger scaling factors might lead to higher cost of encrypted computation while smaller scaling factors might lead to loss in prediction accuracy. To aid in tuning the scaling factors, CHET provides an optional profile-guided optimization. Instead of specifying the scaling factors to use, the user provides a set of representative (training) images. CHET uses these to automatically determine appropriate fixed-point scales for the images, the weights, and the output.

For a target FHE scheme using the given constraints, CHET generates an equivalent, optimized homomorphic tensor circuit as well as an encryptor and decryptor. Both of these executables encode the choices made by the compiler to make the homomorphic computation efficient. For Equation 2.1, the homomorphic tensor circuit generated is:

$$encOutput = hconv2d(encImage, weights, HW); \tag{2.2}$$

There are several data layout options for the encrypted output of each operation and CHET chooses the best one; in this case, it chooses HW layout (described in Section 2.4.2). Similarly, the encryptor and decryptor use the encryption parameters decided by CHET. CHET also chooses the configuration of public keys that the encryptor should generate.

To evaluate the tensor circuit on an image, the client first generates a private key and encrypts the image using the encryptor generated by CHET,

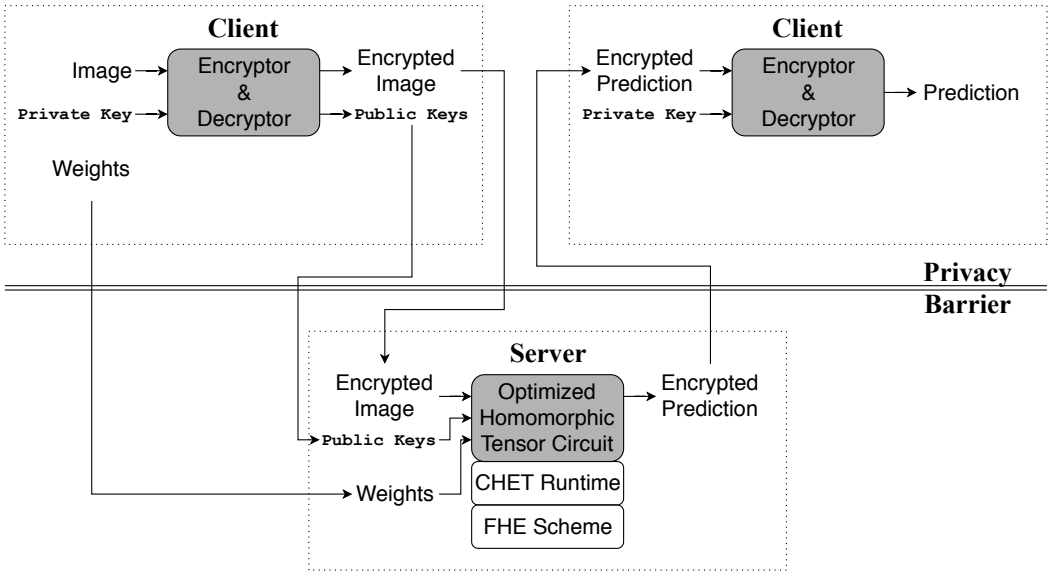


Figure 2.3: Overview of the CHET system at runtime (shaded boxes are programs generated by the CHET compiler).

as shown in Figure 2.3. The encrypted image is then sent to the server along with unencrypted weights and public keys required for evaluating homomorphic operations (i.e., multiplication and rotation). The server executes the optimized homomorphic tensor circuit generated by CHET. The homomorphic tensor operations in the circuit are executed using the CHET runtime, which uses the underlying target FHE scheme to execute homomorphic computations on encrypted data. The circuit produces an encrypted prediction, which it then sends to the client. The client decrypts the encrypted prediction with its private keys using the CHET generated decryptor. In this way, the client runs tensor programs like neural networks on the server without the server being privy to the data, the output (prediction), or any intermediate state.

In this thesis, we assume a semi-honest threat model (like CryptoNets [67]) where the server and the compiler are semi-honest, i.e., the server and the compiler execute the requested computation faithfully but would be curious about the client or user data. The client or user data is private and its confidentiality must be guaranteed. As FHE is non-deterministic and immune to side-channel attacks, encrypting the user data or image using CHET is sufficient to ensure this. Note that CHET only knows the dimensions of the image and weights while the server only knows the image dimensions and the weights used in the model. CHET can be thus used by the client to offload both storage (encrypted data) and computation (neural network inference) to public cloud providers.

While Figure 2.3 presents the flow in CHET for homomorphically evaluating a tensor circuit on a single image, CHET supports evaluating the circuit on multiple images simultaneously, which is known as *batching* in image inference. However, unlike in neural network training, for inference tasks, it is not always true that a large batch-size is available. For example, in a medical imaging cloud service, where each patient's data is encrypted under their personal key, the batch is limited to the images from a single patient. While batching increases the throughput of image inference [67], CHET's focus is decreasing image inference latency. In the rest of this thesis, we consider a batch size of 1, although CHET trivially supports larger batch sizes.

2.3.3 Design of the Compiler

A key design principle of CHET is the separation of concerns between the policies of choosing the secure, accurate, and most efficient homomorphic operation and the mechanisms of executing those policies. The policies include encryption parameters that determine that the computation is secure and accurate, and layout policies that map tensors onto vectors that are crucial for performance. Like Intel MKL libraries that have different implementations of linear algebra operations, the CHET runtime contains different implementations for tensor operations that cater to different input and output layouts. The CHET compiler is responsible for searching over the space of valid and efficient policies.

CHET introduces two abstractions that simplify its design and implementation. *Homomorphic Tensor Circuit* (HTC) is a tensor circuit annotated with the metadata that encodes all the policy decisions for every tensor variable. For instance in Figure 2.1, the HTC precisely specifies how the matrices A and B are mapped onto vectors and the desired layout for the output C. The HTC specification, including data layouts, is described in Section 2.4.2.

The CHET runtime implements the mechanisms necessary to support the policies in HTC. To design the runtime independent of the underlying FHE scheme, we introduce an abstraction called Homomorphic Instruction Set Architecture (HISA), in Section 2.4.1.

For each FHE scheme, the domain expert specifies the cost model of

each HISA primitive in it (the models could be derived through theoretical or experimental analysis). These models use only local information (arguments of the HISA primitive) and are independent of the rest of the circuit. CHET uses these specifications to globally analyze the tensor circuit and choose the encryption parameters, the public keys configuration, and the data layouts. We describe these in detail in Section 2.6.

2.4 Intermediate Representations

CHET uses two intermediate representations: Homomorphic Instruction Set Architecture (HISA) and Homomorphic Tensor Circuit (HTC). HISA is a low-level intermediate representation, that acts as an interface between the CHET runtime and the underlying FHE scheme, whereas HTC is a high-level intermediate representation that acts as an interface between the CHET compiler and the CHET runtime. HISA and HTC are closely related to the operations in the target FHE library and the input tensor circuit, respectively.

2.4.1 Homomorphic Instruction Set Architecture (HISA)

The goal of HISA is to abstract the details of FHE encryption schemes, such as the use of encryption keys and modulus management. This abstraction enables CHET to target new encryption schemes.

Table 2.2 presents instructions or primitives in the HISA. Each FHE library implementing the HISA provides two types: `pt` for plaintexts and `ct` for ciphertexts. During initialization, FHE library generates private keys required

Table 2.2: Primitives of the HISA.

Instruction	Semantics	Signature
encrypt(p)	Encrypt plaintext p into a ciphertext.	$\texttt{pt} \to \texttt{ct}$
decrypt(c)	Decrypt ciphertext c into a plaintext.	$\texttt{ct} \to \texttt{pt}$
copy(c)	Make a copy of ciphertext c.	$\texttt{ct} \to \texttt{ct}$
free(h)	Free any resources associated with handle h.	$\texttt{ct} \cup \texttt{pt} \to \texttt{void}$
encode(m, f)	Encode vector of reals m into a plaintext with a scaling factor f.	$\mathbb{R}^s, \mathbb{Z} \to \texttt{pt}$
decode(p)	Decode plaintext p into a vector of integers.	$\texttt{pt} \to \mathbb{R}^s$
rotLeft(c, x), rotLeftAssign(c, x)	Rotate ciphertext c left x slots.	$\texttt{ct}, \mathbb{Z} \to \texttt{ct}$
rotRight(c, x), rotRightAssign(c, x)	Rotate ciphertext c right x slots.	$\texttt{ct}, \mathbb{Z} \to \texttt{ct}$
add(c, c'), addAssign(c, c')	Add ciphertext, plaintext, or scalar to ciphertext c.	$\texttt{ct}, \texttt{ct} \to \texttt{ct}$
addPlain(c, p), addPlainAssign(c, p)		$\texttt{ct}, \texttt{pt} \to \texttt{ct}$
addScalar(c, x), addScalarAssign(c, x)		$\texttt{ct}, \mathbb{R} \to \texttt{ct}$
sub(c, c'), subAssign(c, c')	Subtract ciphertext, plaintext, or scalar from ciphertext c.	$\texttt{ct}, \texttt{ct} \to \texttt{ct}$
subPlain(c, p), subPlainAssign(c, p)		$\texttt{ct}, \texttt{pt} \to \texttt{ct}$
subScalar(c, x), subScalarAssign(c, x)		$\texttt{ct}, \mathbb{R} \to \texttt{ct}$
mul(c, c'), mulAssign(c, c')	Multiply ciphertext, plaintext, or scalar (at scale f) to ciphertext c.	$\texttt{ct}, \texttt{ct} \to \texttt{ct}$
mulPlain(c, p), mulPlainAssign(c, p)		$\texttt{ct}, \texttt{pt} \to \texttt{ct}$
mulScalar(c, x, f), mulScalarAssign(c, x, f)		$\texttt{ct}, \mathbb{R}, \mathbb{Z} \to \texttt{ct}$
rescale(c, x), rescaleAssign(c, x)	Rescale ciphertext c by scalar x. Undefined unless $\exists ub : x = \text{maxRescale}(c, ub)$.	$\texttt{ct}, \mathbb{Z} \to \texttt{ct}$
maxRescale(c, ub)	Returns the largest $d \leq ub$ that c can be rescaled by.	$\texttt{ct}, \mathbb{Z} \to \mathbb{Z}$

for encryption and decryption, and public keys required for evaluation. The appropriate use of these keys is the responsibility of the FHE library and is not exposed in HISA.

HISA supports point-wise fixed-point arithmetic operations and rotations. The number of slots in the FHE vector, s is a configurable parameter that is provided to the FHE library during the initialization. For schemes that do not support batching, this parameter can be set to 1. For such schemes, rotation operations are no-ops.

The rescale and maxRescale instructions abstract the re-scaling operations provided by the CKKS [42] and the RNS-CKKS [41] schemes. These schemes have restrictions on the scalar value by which a ciphertext can be rescaled. For the CKKS scheme, the scalar has to be a power of 2. For the RNS-CKKS scheme, it has to be the next modulus in the modulus chain. The maxRescale(c, ub) abstracts this detail and returns the maximum value less than or equal to the desired ub that the input ciphertext c can be rescaled by. This instruction additionally guarantees that the return value is less than the modulus of c. For schemes that do not support rescaling, maxRescale always returns 1.

2.4.2 Homomorphic Tensor Circuit (HTC)

The goal of HTC is to provide a high-level abstraction of tensor operations and map them to low-level HISA primitives described above. This abstraction enables CHET to choose the most efficient layouts for input and

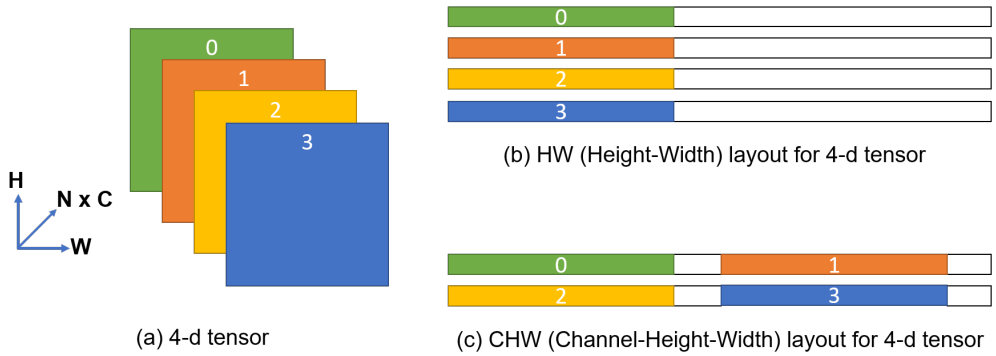


Figure 2.4: An example of a 4-dimensional tensor and its data layouts.

intermediate tensors in the tensor circuit, and call the appropriate optimized runtime functions that implement tensor operations.

HTC provides an encrypted tensor datatype called CipherTensor. It contains metadata about the *logical layout* of the unencrypted tensor, such as its dimensions, padding information, and strides. The metadata is stored as plain integers as it does not leak any information about the data.

Since HISA only supports (1-dimensional) vectors, CipherTensor is responsible for mapping the tensor into its *physical layout* as a vector of ciphertexts, with each ciphertext encrypting a vector. This problem is similar to tiling or blocking of vectors to improve locality of high-performance kernels, but with different constraints. For example, consider a four-dimensional tensor commonly used in image recognition with batch (outermost) N, channel C, height H, and (innermost) width W dimensions, as shown in Figure 2.4(a). Figure 2.4(b) shows one way of mapping such a tensor A by representing each $H \times W$ matrix as a ciphertext in row-major format (with no padding and

stride 1), and having $N \times C$ such ciphertexts in a vector. We call this the HW layout. One can also envision blocking the channel dimension too, where each ciphertext represents a $C \times H \times W$ tensor. We call this the CHW layout. We also support dividing the C dimension across multiple ciphertexts, in which case each ciphertexts represents a $c \times H \times W$ tensor for some $c \leq C$ and $c \bmod C = 0$. Figure 2.4(c) shows an example of CHW layout.

Some tensor operations enforce certain constraints on the cipher tensor datatype, primarily for performance reasons. For example, to perform convolution with same padding, the input 4-d tensor is expected to be padded. Such a padding on unencrypted data is trivial, but adding such padding to an encrypted cipher on-the-fly involves several rotation and point-wise multiplication operations. Similarly, a reshape of the tensor is trivial on unencrypted data but very inefficient on encrypted data. All these constraints arise because using only point-wise operations or rotations on the vector may be highly inefficient to perform certain tensor operations. Therefore, the metadata describing the physical layout of a Cipher-Tensor also includes information about the strides for each dimension. For example, for an image of height (row) and width (column) of 28, a stride of 1 for the width dimension and a stride of 30 for the height dimension allows a padding of 2 (zero or invalid) elements between the rows.

It is not clear which layout is the most efficient way to represent a tensor. As hinted in Section 2.3, the physical layout of the tensors determines the instructions CHET needs to implement tensor operations. For instance,

Figure 2.5 shows the convolution operation when the input image A and the filter F are in HW format. Convolution requires an element-wise multiplication of the image and the filter followed by an addition of neighboring elements. This can be implemented by rotating the image by appropriate amounts and multiplying each ciphertext with an element from the filter. When the values of the filter are unencrypted (which is the case when applying a known model to an encrypted image), this can be performed efficiently with a mulScalar.

Notice that the ciphertexts in Figure 2.5 have some empty space, which in many cases can be filled by using the CHW layout. However, in this case mulPlain needs to be used instead of mulScalar because different weights need to be multiplied to different 2-d HW images in the same ciphertext. As shown in Table 2.1, in RNS-CKKS, both mulPlain and mulScalar have $O(N \cdot r)$ complexity, whereas in CKKS, mulPlain and mulScalar have $O(N \cdot \log N \cdot M(Q))$ and $O(N \cdot M(Q))$ complexities, respectively. Therefore, for CKKS, homomorphic convolution in CHW layout is slower than that in HW layout, whereas for RNS-CKKS, CHW layout may be faster.

By abstracting the physical layout, HTC enables CHET to determine efficient layouts based on a global analysis of the entire circuit (described in Section 2.6). Moreover, by translating reshaping and padding operations in the circuit into operations that change the metadata, CHET avoids or delays performing these expensive operations only when necessary. HTC also enables us to incrementally support new tensor layouts. The current implementation of CHET only supports two layouts for each tensor: HW and CHW. We hope

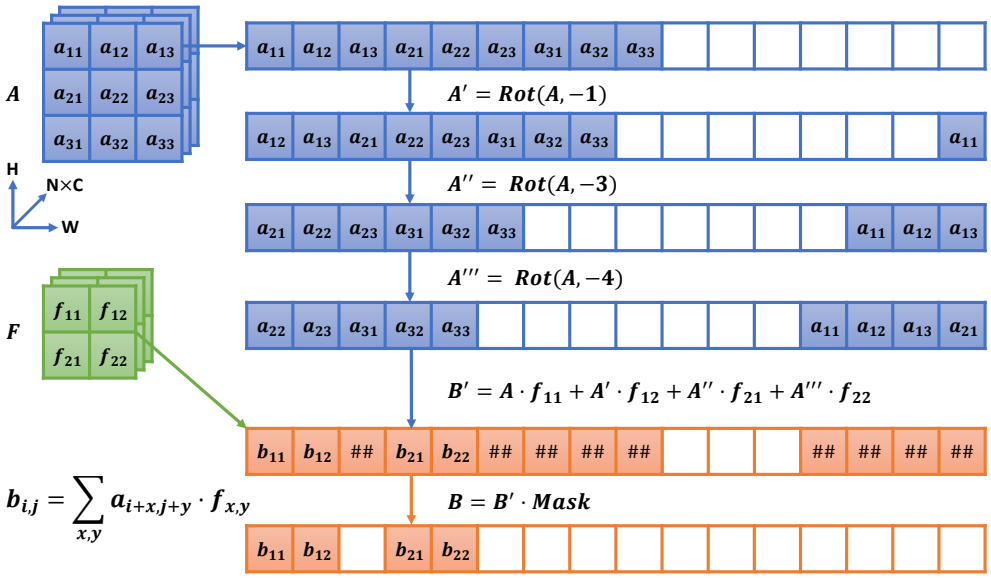


Figure 2.5: Homomorphic convolution of one channel in HW layout with a 2x2 filter (with valid padding).

to extend CHET to support additional layouts in the future.

2.5 A Runtime Library of Homomorphic Tensor Kernels

Intel MKL libraries provide efficient implementations of BLAS operations. In the same way, we design the CHET runtime to provide efficient implementations of homomorphic tensor operations for a given data layout specification. There are several algorithm trade-offs that have significant impact on performance. These optimizations are quite involved (similar to MKL implementations). We discuss some of these trade-offs briefly.

HW-tiled Homomorphic Convolution with valid padding: Consider a CipherTensor that represents a 4-d tensor in HW layout. Figure 2.5 illustrates how homomorphic convolution with valid padding works for a single channel in the HW layout. In convolution, the first element in a HW cipher needs to be added with some of its neighboring elements and before the addition, each of these elements need to be multiplied with different filter weights. To do so, we can left rotate each of these neighboring elements to the same position as the first element. Rotating the ciphertext vector in such a way moves the position appropriately for all HW elements. For each rotated vector, the same weight needs to be multiplied for all HW elements, so mulScalar of the ciphertext is sufficient.

HW-tiled Homomorphic Convolution with same padding: If a similar convolution has to be implemented, but with same padding, then each HW ciphertext is expected to be padded by the compiler so that the amount to rotate (left or right) varies but the number of rotations remain the same. However, after one convolution, there are invalid elements where the padded zeros existed earlier because those are added to the neighboring elements as well. The runtime keeps track of this using an additional metadata on the CipherTensor. The next time a convolution (or some other operation) is called on the CipherTensor, if the CipherTensor contains invalid elements where zeros are expected to be present, then the implementation can mask out all invalid elements with one mulPlain operation (the plaintext vector contains 1 where

valid elements exist and 0 otherwise). This not only increases the time for the convolution operation, but it also increases the modulus Q required because rescale may need to be called after such a masking operation. For security reasons, a larger Q can increase the N that needs to be used during encryption, thereby increasing the cost of all homomorphic operations.

CHW-tiled Homormorphic Convolution: Notice that the ciphertexts in Figure 2.5 have some empty space, which in many cases can be filled by using the CHW layout. However, in this case (with valid padding or with same padding), mulPlain needs to be used instead of mulScalar because different weights need to be multiplied to different HW images in the same ciphertext. In RNS-CKKS, both mulPlain and mulScalar have $O(N)$ complexity, whereas in CKKS, mulPlain and mulScalar have $O(N \log(N))$ and $O(N)$ complexities, respectively. Therefore, for CKKS, homomorphic convolution in CHW layout may be slower than that in HW layout, whereas for RNS-CKKS, CHW layout may be faster. This further demonstrates the need for a compiler to choose the appropriate implementation (like data layout) based on the target FHE scheme.

Furthermore, after multiplication, the multiple input channels in the ciphertext need to be summed up into a single output channel. Such a reduction can be done by rotating every channel to the position of the first one in the ciphertext and adding them up one by one. If C is the number of channels in the cipher, this involves $C - 1$ rotations. However, such a reduction can be

done more efficiently by exploiting the fact that the stride between the input channels is the same. This requires at the most $2log(C)$ rotations with additions in-between rotations, similar to vectorized reduction on unencrypted data. To produce an output ciphertext with multiple channels C, the input ciphertext needs to be replicated $C-1$ times. Instead of $C-1$ rotations serially, this can also be done in $2log(C)$ rotations, by adding the ciphertexts in-between.

Homomorphic Matrix Multiplication: Different FHE operations have different latencies. Although rotLeft and mulPlain have similar algorithmic complexity in HEAAN, the constants may vary and we observe that, mulPlain is more expensive than rotLeft. Due to this, it may be worth trading multiplications for rotations. This trade-off is most evident in homomorphic matrix multiplication operation. The number of mulPlain required reduce proportional to the number of replicas of the data we can add in the same cipher. Adding replicas increase the number of rotations but decrease the number of multiplications. This yields much more benefit because replicas can be added in *log* number of rotations instead of the linear number of multiplications.

2.6 Compiling a Tensor Circuit

This section describes the CHET compiler. We first describe a global data-flow analysis and transformation framework (Section 2.6.1), which is instantiated for three different analyses to a) determine the encryption parame-

ters (Section 2.6.2), b) choose an efficient layout for tensors (Section 2.6.3), and c) generate rotation keys necessary to evaluate the input circuit (Section 2.6.4). We then describe a profile-guided optimization to choose the fixed-point scales for the inputs and output (Section 2.6.5).

2.6.1 Analysis and Transformation Framework

The input tensor circuit is first translated to an equivalent Homomorphic Tensor Circuit (HTC) as follows: (i) any tensor marked as encrypted in the input schema is considered to be of type CipherTensor, and (ii) any tensor operation with at least one input CipherTensor is mapped to an equivalent homomorphic tensor operation that produces a CipherTensor as output. CHET then analyses and transforms the HTC.

Analyzing the HTC involves performing a sequence of data flow analyses on the HTC. Each such analysis requires an FHE expert to specify the data-flow equation for each HISA primitive. We call the set of these equations the HISA-Analyser. Each equation reads the data-flow information of its inputs and writes the data-flow information of its output, and is agnostic of the rest of the circuit.

The obvious way to run the data-flow analysis is to explicitly build the data-flow graph of HISA primitives from the HTC. In contrast to most traditional optimizing compilers, the HTC has two key properties: its data-flow graph is a Directed Acyclic Graph (DAG) and the dimensions of tensors in the graph are known at compile-time from the schema provided by the user

(similar to High Performance Fortran compilers). CHET exploits this to elide building such a data-flow graph in memory.

Instead, CHET performs the analysis by dynamically unrolling the graph *on-the-fly*. This is done by executing the HTC and the CHET runtime using a different interpretation. In our implementation, we customize the `ct` datatype in HISA primitives using templates to store data-flow information and overload operations to execute the data-flow equations rather than call into the underlying FHE library. Thus, executing the framework in this new interpretation automatically composes the data-flow equations to perform the analysis. Once a HISA-Analyser has finished, a subsequent *transformer* uses the data-flow information computed to transform the circuit appropriately.

To summarize, a compiler transformation pass in CHET for a particular FHE scheme can be defined by the following: (1) ct *datatype* to store the data-flow information along with its initialization, (2) data-flow equation for each HISA primitive (*HISA-Analyser*), and (3) a *transformer* that takes the data-flow analysis results and returns a specification for the HTC. The framework handles the rest. The analysis and transformation framework can thus be used to easily add new transformations and new target FHE schemes.

2.6.2 Encryption Parameters Selection

The goal of this analysis is to determine the encryption parameters to use when evaluating a given circuit. These parameters are also used in the Encryptor and Decryptor to encrypt the inputs and decrypt the results

respectfully. As described in Section 2.2.3, parameters N and Q determine the performance, security, and correctness of an FHE computation in the CKKS and RNS-CKKS schemes.

For a given Q, the minimum value of N that guarantees that security level against currently known attacks is a deterministic map, as listed in [36] (Section 5.4). We pre-populate this in a table and choose 128-bit security (this is a configurable hyper-parameter). The data-flow analysis in this compiler pass is thus to aid in determining only the coefficient modulus, Q.

As described in Section 2.2.3, maximizing performance requires us to find a tight lower-bound for Q that is sufficiently large to evaluate the input circuit correctly and to the desired precision. In both CKKS and RNS-CKKS, the coefficient modulus changes during execution of the rescale instruction: rescale takes the ciphertext message c with modulus m and an integer x, and produces a ciphertext c' with modulus $m' = m/x$; in effect, the modulus is "consumed" in the rescale operation. If the new modulus is below 1, then the resulting ciphertext is invalid. Hence, tracking the modulus consumed using data-flow analysis can find the minimum required Q.

The challenge in tracking the modulus consumed is that both CKKS and RNS-CKKS restrict the divisors that can be used in the rescale instruction (the maxRescale instruction can be used to query a suitable divisor for a given bound). To analyze their behavior at runtime, we use the analysis framework to execute the instructions using a different interpretation. This interpretation tracks how the modulus changes with rescale (Section 2.2.3) and which divisors

are valid, as returned from maxRescale. Using this, the encryption parameters selection pass for CKKS is defined by:

Datatype ct stores the modulus "consumed" and is initialized to 1.

HISA-Analyser The maxRescale instruction returns the same value that it would have returned if it was executed in CKKS. The rescale instruction multiplies the divisor with the input's ct, and stores it to the output's ct, thereby tracking the modulus consumed during computation. In all other instructions, if the output is a ct, then it is the same as the inputs' ct (which are required to all be the same).

Transformer The circuit output's ct captures the modulus "consumed" during the entire circuit. The user-provided desired output fixed-point scaling factor (precision) is multiplied by the circuit output's ct to get Q. Finally, N is chosen using the pre-populated table.

For the RNS-CKKS scheme, the analysis assumes there is a global list $Q_1, Q_2, \ldots, Q_n$ of pre-generated candidate moduli[3] for a sufficiently large n. The goal of the analysis is to pick the smallest r such that $Q = \Pi_{i=1}^{r} Q_i$ can be used as the modulus for the input circuit.

Datatype ct stores the index k in the list of moduli to be "consumed" next and is initialized to 1.

[3]By default, CHET uses a list of 60-bit primes distributed in SEAL.

HISA-Analyser For the maxRescale(c, ub) and rescale instructions, the analyser determines the largest $j \geq k$ such that $\Pi_{i=k}^{j} Q_i \leq ub$. If such a j exists, maxRescale returns $\Pi_{i=k}^{j} Q_i$ and rescale stores the index $j+1$ to the output's ct. If no such j exists, rescale stores the index k and maxRescale returns 1. In all other instructions, if the output is a ct, then it is the same as the inputs' ct (which are required to all be the same).

Transformer The circuit output's ct captures the number of moduli "consumed" during the entire circuit. The length of the modulus chain is then chosen as the smallest r such that $\Pi_{i=ct}^{r} Q_i$ is greater than the user-provided desired output fixed-point scaling factor (precision) multiplied by the circuit output's ct. Q is set to $\Pi_{i=1}^{r} Q_i$. Finally, N is chosen using the pre-populated table.

2.6.3 Data Layout Selection

The goal of the data layout selection pass is to determine the data layout of the output of each homomorphic tensor operation in the HTC so that the execution time of the HTC is minimized. We first need to search the space of data layout choices. For each such choice, we then need to analyze the cost of the HTC corresponding to that choice. To do this, we need an estimation of the cost of each HISA primitive. We describe each of these three components in detail next.

The cost of HISA primitives could be estimated using theoretical anal-

ysis (through asymptotic complexity) or experimental analysis (through microbenchmarking). Table 2.1 lists the asymptotic complexity of different HISA primitives in CKKS and RNS-CKKS. Different HISA primitives have different costs, even within the same FHE scheme. In this thesis, we use a combination of theoretical and experimental analysis, by using asymptotic complexity and tuning the constants involved using microbenchmarking of CKKS and RNS-CKKS instructions, to derive a cost model for HISA primitives (agnostic of the tensor circuit and inputs).

Given a cost model for HISA primitives, data-flow analysis is used to estimate the cost of executing a HTC with a specific data layout choice. We define the cost estimation pass for both CKKS and RNS-CKKS as follows:

Datatype ct stores the cost and is initialized to 0.

HISA-Analyser For each HISA primitive, if the output is a ciphertext, then its ct is the sum of the ct of the inputs and the cost of that primitive, according to the cost model specific to CKKS and RNS-CKKS.

Transformer The circuit output's ct captures the cost of the entire HTC. If the cost is smaller than the minimum observed so far, then the minimum cost is updated and the best data layout choice is set to the current one.

Note that as we execute HISA instructions using this new interpretation, we use parallel threads during analysis to estimate the cost on each thread and take the maximum across threads as the cost of the HTC.

The runtime exposes the data layout choices for the outputs of homomorphic tensor operations. In our current runtime, there are only 2 such choices per tensor operation, HW and CHW layouts (Section 2.4.2). The search-space for the HTC is exponential in the number of tensor operations, so it is huge. An auto-tuner might be useful to explore this search space. We instead use domain-specific heuristics to prune this search space:

- Homomorphic convolutions are typically faster if the input and output are in HW, while all the other homomorphic tensor operations are typically faster if the input and output are in CHW.
- Homomorphic matrix multiplications (fully connected layers) are typically faster when the output is in CHW, even if the input is in HW, while all other homomorphic tensor operations are typically faster if both input and output are in the same layout.

Using these heuristics, we consider only 4 data layout choices for the HTC:

1. HW: all homomorphic tensor operations use HW.
2. CHW: all homomorphic tensor operations use CHW.
3. HW-conv, CHW-rest: homomorphic convolution uses HW, while all other homomorphic tensor operations use CHW.
4. CHW-fc, HW-before: all homomorphic tensor operations till the first homomorphic fully connected layer use HW and everything thereafter uses CHW.

We thus restrict the search space to a constant size. Each data layout choice corresponds to a HTC. For each choice, we select the encryption parameters and analyze the cost (two passes). We then pick the minimum cost one.

2.6.4 Rotation Keys Selection

The goal of the rotation keys selection pass is to determine the public evaluation keys for rotation that needs to be generated by the encryptor and decryptor. In both CKKS and RNS-CKKS, to rotate a ciphertext by x slots, a public evaluation key for rotating by x is needed. Recall that the vector width used for FHE vectorization is $N/2$. Since the range of possible rotations of this vector, i.e., $N/2$, is huge and each rotation key consumes significant memory, it is not feasible to generate rotation keys for each possible rotation. Therefore, both CKKS and RNS-CKKS, by default, insert public evaluation keys for power-of-2 left and right rotations, and all rotations are performed using a combination of power-of-2 rotations. As a consequence, any rotation that is not a power-of-2 would perform worse because it has to rotate multiple times. Nevertheless, only $2log(N)-2$ rotation keys are stored by default. This is too conservative. In a given homomorphic tensor circuit, the distinct slots to rotate would not be in the order of N. We use data-flow analysis to track the distinct slots of rotations used in the HTC.

We define the rotation keys selection pass for both CKKS and RNS-CKKS as follows:

Datatype ct stores the set of rotation slots used and is initialized to $\emptyset$.

HISA-Analyser Any rotate HISA instruction inserts the slots to rotate to the input's ct, and stores it to the output's ct. In all other instructions, if the output is a ciphertext, then its ct is the union of the inputs' ct.

Transformer The circuit output's ct captures all the rotation slots used in the HTC. Rotation keys for these slots are marked for generation.

2.6.5 Fixed-Point Scaling Factor Selection

The inputs to the compiler passes discussed so far include the fixed-point scales of the inputs (images and weights) in HTC as well as the desired fixed-point scale (or precision) of the output. It is not straightforward for the user to determine the scaling factors to use for the floating-point values. Moreover, these factors interact with the approximation noise introduced by the FHE scheme, so the fixed-point scales must be large enough to ensure that the noise added does not reduce accuracy too much. To alleviate this, CHET includes an optional profile-guided optimization that finds appropriate scaling factors. Instead of fixed-point scales, the user provides unencrypted inputs along with a tolerance (or error-bound) for the output.

Floating-point numbers are encoded as fixed-point numbers by multiplying them by their specified fixed-point scale (inputs are scaled before encryption if applicable). Recall that multiplication of two such fixed-pointed numbers results in a number at a larger fixed-point scale, that may be *rescaled* back down if a suitable divisor exists (as provided by maxRescale). The magnitude of the divisors (for CKKS) or when divisors become available (for RNS-

CKKS) is determined by the fixed-point scales of the operands. Thus, the selection of scaling factors directly impacts rescale and consequently, encryption parameters selection.

Larger scaling factors yield larger encryption parameters and worse performance, whereas smaller scaling factors yield smaller encryption parameters and better performance but the output might vary beyond the tolerance, leading to prediction inaccuracy. The compiler minimizes the scaling factors while ensuring that the output values are within tolerance for the given inputs.

Given a set of scaling factors, we use CHET to generate the optimized HTC. For each input, we encrypt it using parameters chosen by CHET, run the HTC, and decrypt the output. We compare this with the output of the unencrypted tensor circuit. If the difference is beyond tolerance for any of the output values for any input, then we reject these scaling factors. Otherwise, the scaling factors are acceptable.

For neural network inference, CHET allows specifying 4 fixed-point scaling factors - one for the image, one for masks, and two for the weights depending on whether they are used as scalars or plaintext (vectors)[4]. If there are x choices for selecting the scaling factors for each, then there are x^4 choices in total. To limit the search space to $4 \cdot x$, we used a round-robin strategy: each scaling factor starts from 2^{40} and we decrease the exponents by one as long as the accuracy is acceptable. The search continues until a minimum is

[4]Separate scaling factors are used because approximation noise of encoding in CKKS is smaller when all the elements in a plaintext are equal.

Table 2.3: Deep Neural Networks used in our evaluation.

Network	No. of layers			# FP operations	Accuracy (%)
	Conv	FC	Act		
LeNet-5-small	2	2	4	159960	98.5
LeNet-5-medium	2	2	4	5791168	99.0
LeNet-5-large	2	2	4	21385674	99.3
Industrial	5	2	6	-	-
SqueezeNet-CIFAR	10	0	9	37759754	81.5

reached.

2.7 Experimental Evaluation

In this section, we first describe our experimental setup (Section 2.7.1). We then present our evaluation of homomorphical neural network inference using compiler-generated and hand-written programs (Section 2.7.2). Finally, we analyze the impact of compiler optimizations (Section 2.7.3).

2.7.1 Experimental Setup

We evaluate the CHET compiler with two target FHE libraries: HEAAN v1.0 [92] and SEAL v3.1 [148], that implement the FHE schemes, CKKS [42] and RNS-CKKS [41], respectively. Our evaluation targets a set of convolutional neural network (CNN) architectures for image classification tasks that are summarized in Table 2.3.

LeNet-5-like is a series of networks for the MNIST [113] dataset. We use three versions with different number of neurons: LeNet-5-small, LeNet-5-medium, and LeNet-5-large. The largest one matches the one used in

the TensorFlow's tutorials [154]. These networks have two convolutional layers, each followed by ReLU activation and max pooling, and two fully connected layers with a ReLU in between.

Industrial is a pre-trained HE-compatible neural network from an industry partner for privacy-sensitive binary classification of images. We are unable to reveal the details of the network other than the fact that it has 5 convolutional layers and 2 fully connected layers.

SqueezeNet-CIFAR is a neural network for the CIFAR-10 dataset [105] that follows the SqueezeNet [91] architecture. This version has 4 Fire-modules [47] for a total of 10 convolutional layers. To the best of our knowledge, SqueezeNet-CIFAR is the deepest NN that has been homomorphically evaluated.

All networks other than Industrial use ReLUs and max-pooling, which are not compatible with homomorphic evaluation (HE). For these networks, we modified the activation functions to a second-degree polynomial [33, 67]. The key difference with prior work is that our activation functions are $f(x) = ax^2 + bx$ with learnable parameters a and b. During the training phase, the CNN adjusts these parameters to implement an appropriate activation function. We also replaced max-pooling with average-pooling.

Table 2.3 lists the accuracies for the HE-compatible networks. For LeNet-5-large, the 99.3% accuracy matches that of the unmodified network. For SqueezeNet-CIFAR, the resulting accuracy of 81.5% is close to that of the

non-HE-compatible network, which achieves an accuracy of 84%. Note that encryption was not used during training. The learned weights are used for inference with an encrypted image.

To provide a fair comparison with a set of hand-written HEAAN baselines that used non-standard encryption parameters, experiments with CHET-HEAAN were run with matching parameters that offer somewhat less than 128-bit security. Experiments with CHET-SEAL use the default 128-bit security level.

All experiments were run on a dual socket Intel Xeon E5-2667v3@3.2GHz with 224 GB of memory. Hyperthreading was turned off for a total of 16 hardware threads. We present the average latency of image inference with a batch size of 1. All latencies are reported as averages over 20 different images.

2.7.2 Compiler-Generated vs. Hand-Written

For a fair comparison, we consider hand-written implementations using HEAAN for a subset of the networks. The RNS-CKKS scheme in SEAL is difficult to manually hand tune, and thus we do not compare with hand-written version for SEAL. Experts took weeks of programming effort to optimize each network independently. They reduced the latency of LeNet-5-small and LeNet-5-medium to 14 and 140 seconds, respectively. For Industrial, the default implementation by a developer took more than 18 hours per image, which the experts were able to tune to less than 45 minutes after months of work.

Figure 2.6 compares the hand-written implementations with CHET

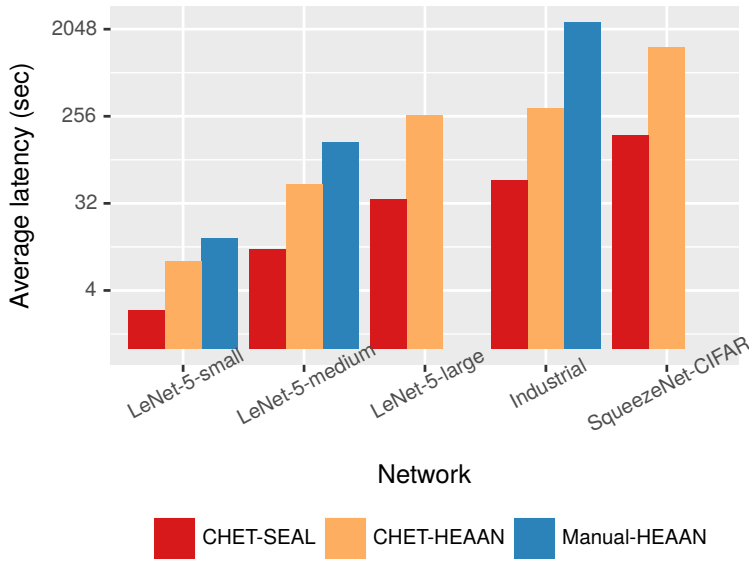


Figure 2.6: Average latency (log scale) of CHET-SEAL, CHET-HEAAN, and hand-written HEAAN versions.

generated optimized ones for HEAAN and SEAL. For context, CHET-SEAL is around two orders of magnitude slower than CHET's unencrypted reference inference engine. CHET clearly outperforms hand-written implementations, even when using HEAAN. The hand-written implementations are slower because it is tedious and error prone to explore different data layouts. CHET not only explores layouts but also chooses the best one automatically. Furthermore, CHET-SEAL is an order of magnitude faster than hand-written implementations, while the complications of the RNS-CKKS scheme are automatically taken care of.

Cryptonets [67] homomorphically evaluate a highly tuned neural network for the MNIST [113] data using the YASHE [22] scheme. While this neural network is smaller than our LeNet-5-small, its accuracy is similar to

Table 2.4: Encryption parameters N and Q selected by CHET-HEAAN and the user-provided fixed-point parameters.

Network	N	$\log(Q)$	$\log(P_c,$	$P_w,$	$P_u,$	$P_m)$
LeNet-5-small	8192	240	30	16	15	8
LeNet-5-medium	8192	240	30	16	15	8
LeNet-5-large	16384	400	40	20	20	10
Industrial	32768	705	35	25	20	10
SqueezeNet-CIFAR	32768	940	30	20	20	10

that of our LeNet-5-medium. The average latency of image inference in their highly optimized implementation is 250 seconds (throughput is higher because they consider a larger batch size). In contrast, the latency of CHET-SEAL generated code that achieves similar accuracy is only 10.8 seconds.

2.7.3 Analysis of Compiler Optimizations

Parameter Selection: The encryption parameters N and Q selected by CHET are shown in Table 2.4 for HEAAN with the best data layout. The values of these parameters grow with the depth of the circuit. The last columns show the fixed-point scaling parameters that were used for the image (P_c), plaintext (P_w) and scalar weights (P_u), and masks (P_m). With these parameters, encrypted inference achieved the same accuracy as unencrypted inference of the same HE-compatible networks.

Data Layout Selection: It is time-consuming to evaluate all possible data layouts, so we evaluate only the pruned subset that CHET searches by default. The pruned data layouts that we evaluate are:

Table 2.5: Average latency (sec) with different data layouts using CHET-SEAL.

Network	HW	CHW	HW-conv CHW-rest	CHW-fc HW-before
LeNet-5-small	**2.5**	3.8	3.8	**2.5**
LeNet-5-medium	22.1	**10.8**	25.8	18.1
LeNet-5-large	64.8	**35.2**	64.6	61.2
Industrial	108.4	**56.4**	181.1	136.3
SqueezeNet-CIFAR	429.3	**164.7**	517.0	441.0

Table 2.6: Average latency (sec) with different data layouts using CHET-HEAAN.

Network	HW	CHW	HW-conv CHW-rest	CHW-fc HW-before
LeNet-5-small	8	12	8	**8**
LeNet-5-medium	82	91	52	**51**
LeNet-5-large	325	423	270	**265**
Industrial	330	**312**	379	381
SqueezeNet-CIFAR	**1342**	1620	1550	1342

1. HW: each ciphertext has all height and width elements of a single channel only.

2. CHW: each ciphertext can have multiple channels (all height and width elements of each).

3. HW-conv and CHW-rest: same has CHW, but move to HW before each convolution and back to CHW after each convolution.

4. CHW-fc and HW-before: same as HW, but switch to CHW during the first fully connected layer and CHW thereafter.

Tables 2.5 and 2.6 present the average latency of different layouts for SEAL

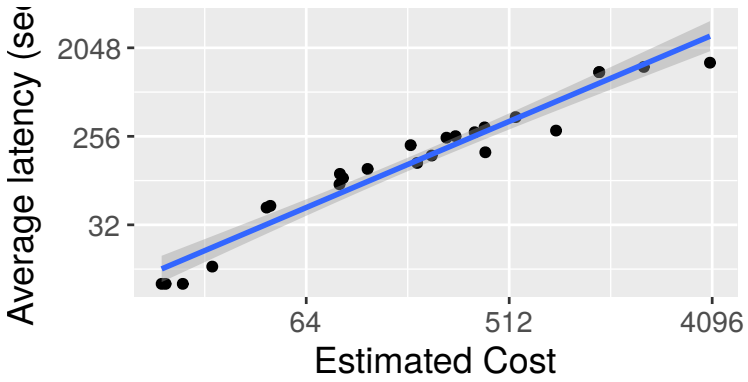


Figure 2.7: Estimated cost vs. observed average latency (log-log scale) for different layouts and networks in CHET.

and HEAAN. The best data layout depends on the network as well as the FHE scheme. For example, the time for convolutions in HEAAN could be more for CHW layout than HW layout because mulPlain is more time-consuming than mulScalar. On the other hand, mulPlain and mulScalar take similar time in SEAL. Hence, for the same network, CHW layout might be the best layout in SEAL but not in HEAAN. It is very difficult for the user to determine which is the best data layout and it is tedious to implement each network manually using a different data layout. For all these networks, CHET chooses the best performing data layout automatically based on its cost model of SEAL or HEAAN. Figure 2.7 plots the cost estimated by CHET and the observed average latency for each network and data layout. We observe that they are highly correlated, so our cost model is quite accurate.

Rotation Keys Selection: Figure 2.8 presents the speedup of using only CHET selected rotation keys over using power-of-2 rotation keys (by default).

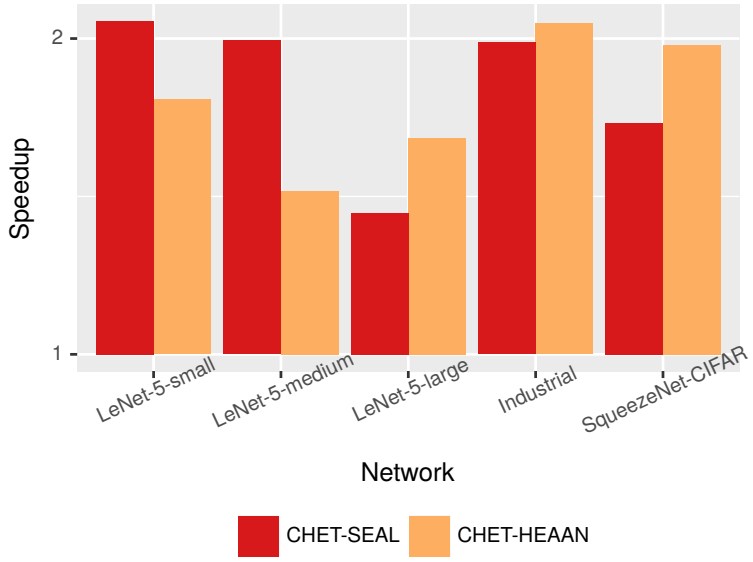


Figure 2.8: Speedup (log scale) of rotation keys selection optimization over default power-of-2 rotation keys.

The geometric mean speedup is 1.8× for all networks and FHE schemes. We observe that the rotation keys chosen by the compiler are a constant factor of $log(N)$ in every case. CHET thus significantly improves performance without consuming noticeably more memory.

2.8 Related Work

FHE is currently an active area of research [12, 14, 22, 25, 26, 37, 40–42, 44, 59, 64, 111, 160]. See Acar et al [8] for a survey of FHE schemes. CHET can be used to homomorphically evaluate tensor programs using any of these schemes. We demonstrated the utilities of CHET for the most recent FHE schemes, CKKS [42] and its RNS variant [41].

Many have observed the need and suitability of FHE for machine learning tasks [33, 67, 83, 96]. Cryptonets [67] was the first tool to demonstrate a fully-homomorphic inference of a CNN by replacing the (FHE incompatible) ReLU activations with a quadratic function. Our emphasis in this thesis is primarily on automating the manual and error-prone hand tuning required to ensure that the networks are secure, correct, and efficient.

Bourse et al. [24] use the TFHE library [44] for CNN inference. TFHE operates on bits and is thus slow for multi-bit integer arithmetic. To overcome this difficulty, Bourse et al. instead use a discrete binary neural network. Similarly, Cingulata [30, 31] is a compiler for converting C++ programs into a Boolean circuit, which is then evaluated using a backend FHE library. Despite various optimizations [30], these approaches are unlikely to scale for large CNNs.

Prior works [57, 155] have used partially homomorphic encryption schemes (which support addition or multiplication of encrypted data but not both) to determine the encryption schemes to use for different data items so as to execute a given program. While they are able to use computationally efficient schemes, such techniques are not applicable for evaluating CNNs that require both multiplication and addition to be done on the same input.

DeepSecure [145] and Chameleon [144] use secure multi-party computation techniques [73, 170] to perform CNN inference. Many frameworks like SecureML [129], MiniONN [117], and Gazelle [98] combine secure two-party computation protocols [170] with homomorphic encryption. Secure multi-

party computations require the client and the server to collaboratively perform computation, involving more communication and more computation on the client when compared to FHE. These tradeoffs between FHE and other secure computation techniques are beyond the scope of the thesis. Our focus is instead on optimizing FHE computations. Nevertheless, a compiler like CHET could possibly be useful in other secure computing techniques as well.

Chapter 3

EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation

Fully-Homomorphic Encryption (FHE) offers powerful capabilities by enabling secure offloading of both storage and computation, and recent innovations in schemes and implementation have made it all the more attractive. At the same time, FHE is notoriously hard to use with a very constrained programming model, a very unusual performance profile, and many cryptographic constraints. Existing compilers for FHE either target simpler but less efficient FHE schemes or only support specific domains where they can rely on expert-provided high-level runtimes to hide complications.

This thesis presents a new FHE language called Encrypted Vector Arithmetic (EVA)[1], which includes an optimizing compiler that generates correct and secure FHE programs, while hiding all the complexities of the target

[1]My main contributions to this work include the design and implementation of the language and compiler as well as the compiler optimizations. The full citation of the published version of this work is: "Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, New York, NY, USA, 2020. ACM".

FHE scheme. Bolstered by our optimizing compiler, programmers can develop efficient general-purpose FHE applications directly in EVA. For example, we have developed image processing applications using EVA, with a very few lines of code.

EVA is designed to also work as an intermediate representation that can be a target for compiling higher-level domain-specific languages. To demonstrate this, we have re-targeted CHET, an existing domain-specific compiler for neural network inference, onto EVA. Due to the novel optimizations in EVA, its programs are on average 5.3× faster than those generated by CHET. We believe that EVA would enable a wider adoption of FHE by making it easier to develop FHE applications and domain-specific FHE compilers.

3.1 Introduction

Fully-Homomorphic Encryption (FHE) allows arbitrary computations on encrypted data without requiring the decryption key. Thus, FHE enables interesting privacy-preserving capabilities, such as offloading secure storage and secure computation to untrusted cloud providers. Recent advances in FHE theory [41, 42] along with improved implementations have pushed FHE into the realm of practicality. For instance, with appropriate optimization, we can perform encrypted fixed-point multiplications within a few microseconds, which matches the speed of 8086 processors that jumpstarted the computing revolution. Future cryptographic innovations will further reduce the performance gap between encrypted and unencrypted computations.

Despite the availability of multiple open-source implementations [81, 92, 138, 149], programming FHE applications remains hard and requires cryptographic expertise, making it inaccessible to most programmers today. Furthermore, different FHE schemes provide subtly different functionalities and require manually setting encryption parameters that control correctness, performance, and security. We expect the programming complexity to only increase as future FHE schemes become more capable and performant. For instance, the recently invented CKKS scheme [42] supports fixed-point arithmetic operations by representing real numbers as integers with a fixed scaling factor, but requires the programmer to perform rescaling operations so that scaling factors and the cryptographic noise do not grow exponentially due to multiplications. Moreover, the so-called RNS-variant of the CKKS scheme [41] provides efficient implementations that can use machine-sized integer operations as opposed to multi-precision libraries, but imposes further restrictions on the circuits that can be evaluated on encrypted data.

To improve the developer friendliness of FHE, this thesis proposes a new general-purpose language for FHE computation called Encrypted Vector Arithmetic (EVA). EVA is also designed to be an intermediate representation that is a backend for other domain-specific compilers. At its core, EVA supports arithmetic on fixed-width vectors and scalars. The vector instructions naturally match the encrypted SIMD – or batching – capabilities of FHE schemes today. EVA includes an optimizing compiler that hides all the complexities of the target FHE scheme, such as encryption parameters and noise.

It ensures that the generated FHE program is correct, performant, and secure. In particular, it eliminates all common runtime exceptions that arise when using FHE libraries.

EVA implements FHE-specific optimizations, such as optimally inserting operations like rescaling and modulus switching. We have built a compiler incorporating all these optimizations to generate efficient programs that run using the Microsoft SEAL [149] FHE library which implements the RNS-variant of the CKKS scheme. We have built an EVA executor that transparently parallelizes the generated program efficiently, allowing programs to scale well. The executor also automatically reuses the memory used for encrypted messages, thereby reducing the memory consumed.

To demonstrate EVA's usability, we have built a Python frontend for it. Using this frontend, we have implemented several applications in EVA with a very few lines of code and much less complexity than in SEAL directly. One application computes the length of a path in 3-dimensional space, which can be used in secure fitness mobile applications. We have implemented some statistical machine learning applications. We have also implemented two image processing applications, Sobel filter detection and Harris corner detection. We believe Harris corner detection is one of the most complex programs that have been evaluated using CKKS.

In addition, we have built a domain-specific compiler on top of EVA for deep neural network (DNN) inference. This DNN compiler subsumes our prior domain-specific compiler called CHET [56]. This compiler takes programs

written in a higher-level language as input similar to CHET and generates EVA programs using a library of operations on higher-level constructs like tensors and images. Our DNN compiler uses the same tensor kernels as CHET, except that it generates EVA programs instead of generating SEAL programs. Nevertheless, the optimizing compiler in EVA is able to outperform CHET in DNN inference by 5.3× on average.

In summary, EVA is a general-purpose language and an intermediate representation that improves the programmability of FHE applications by guaranteeing correctness and security, while outperforming current methods.

The rest of this chapter is organized as follows. Section 3.2 gives background on FHE. Section 3.3 presents the EVA language. Section 3.4 gives an overview of the EVA compiler. We then describe transformations and analysis in the compiler in Sections 3.5 and 3.6 respectively. Section 3.7 briefly describes the domain-specific compilers we built on top of EVA. We present our evaluation in Section 3.8. Finally, Section 3.9 describes related work.

3.2 Background and Motivation

In this section, we describe FHE (Section 3.2.1) and the challenges in using it (Section 3.2.2). We also describe an implementation of FHE (Section 3.2.3). Finally, we present the threat model assumed in this thesis (Section 3.2.4).

3.2.1 Fully-Homomorphic Encryption (FHE)

An FHE scheme includes four stages: key generation, encryption, evaluation, and decryption. Most of the efficient FHE schemes, for example, BGV [26], BFV [59], and CKKS [42], are constructed on the Ring Learning with Errors (RLWE) problem [119]. At the time of key generation, a polynomial ring of degree N with integer coefficients modulo Q must be chosen to represent ciphertexts and public keys according to the security standard [9]. We call Q the ciphertext modulus. A message is encoded to a polynomial, and subsequently encrypted with a public key or a secret key to form a ciphertext consisting of two polynomials of degree up to $N-1$. Encryption also adds to a ciphertext a small random error that is later removable in decryption.

FHE schemes are malleable by design. From the perspective of the user, they offer a way to encrypt integers (or fixed-point numbers in CKKS — see the next section) such that certain arithmetic operations can be evaluated on the resulting ciphertexts. Evaluation primarily includes four operations: addition of ciphertexts, addition of a ciphertext and a plaintext, multiplication of ciphertexts, and multiplication of a ciphertext and a plaintext. Decrypting (with a secret key) and decoding reveals the message, as if the computation was performed on unencrypted data.

Many modern FHE schemes also include a SIMD-like feature known as *batching* which allows a vector of values to be encrypted as a single ciphertext ($N/2$ values in CKKS). With batching, arithmetic operations happen in an element-wise fashion. *Batching-compatible* schemes can evaluate rotations

which allow data movement inside a ciphertext. But evaluating each rotation step count needs a distinct public key.

3.2.2 Challenges in Using FHE

Programmers using FHE face significant challenges that must be overcome for correct, efficient, and secure computation. We discuss those challenges here to motivate our work.

Depth of Computation: Computations on ciphertexts increase the initially small error in them linearly on the number of homomorphic additions and exponentially on the multiplicative depth of the evaluation circuit. When the errors get too large, ciphertexts become corrupted and cannot be decrypted, even with the correct secret key. This bound is in turn determined by the size of the encryption parameter Q. Thus, to support efficient homomorphic evaluation of a circuit, one must optimize the circuit for low depth.

Relinearization: Each ciphertext consists of 2 or more polynomials (freshly encrypted ciphertexts consist of only 2 polynomials). Multiplication of two ciphertexts with k and l polynomials yields a ciphertext with $k + l + 1$ polynomials. To prevent the number of polynomials from growing indefinitely, an operation called relinearization is performed to reduce it back to 2. Relinearization from each distinct number of polynomials to 2 requires a distinct public key. Relinearization is costly and their optimal placement is an NP-hard problem [37].

CKKS and Approximate Fixed-Point: The CKKS [42] scheme introduced an additional challenge by only providing *approximate results* (but much higher performance in return). There are two main sources of error in CKKS: (i) error from the encoding of values to polynomials being lossy, and (ii) the noise added in every homomorphic operation being mixed with the message. To counter this, CKKS adopts a fixed-point representation by associating each ciphertext with an unencrypted scaling factor. Using high enough scaling factors allows the errors to be hidden.

CKKS further features an operation called *rescaling* that scales down the fixed-point representation of a ciphertext. Consider a ciphertext x that contains the encoding of 0.25 multiplied by the scale 2^{10} (a relatively low scale). x^2 encodes 0.0625 multiplied by the scale 2^{20}. Further powers would rapidly overflow modest values of the modulus Q, requiring impractically large encryption parameters to be selected. Rescaling the second power by 2^{10} will truncate the fixed-point representation to encode the value at a scale of 2^{10}.

Rescaling has a secondary effect of also dividing the ciphertext's modulus Q by the same divisor as the ciphertext itself. This means that there is a limited "budget" for rescaling built into the initial value of Q. The combined effect for CKKS is that $\log Q$ can grow linearly with the multiplicative depth of the circuit. It is common to talk about the *level* of a ciphertext as how much Q is left for rescaling.

A further complication arises from the ciphertext after rescaling being encrypted under fundamentally different encryption parameters. To apply

any binary homomorphic operations, two ciphertexts must be at the same level, i.e., have the same Q. Furthermore, addition and subtraction require ciphertexts to be encoded at the same scale due to the properties of fixed-point arithmetic. CKKS also supports a modulus switching operation to bring down the level of a ciphertext without scaling the message. *In our experience, inserting the appropriate rescaling and modulus switching operations to match levels and scales is a significantly difficult process even for experts in homomorphic encryption.*

In the most efficient implementations of CKKS (so called RNS-variants [41]), the truncation is actually performed by dividing the encrypted values by prime factors of Q. Furthermore, there is a fixed order to these prime factors, which means that from a given level (i.e., how many prime factors are left in Q) there is only one valid divisor available for rescaling. This complicates selecting points to rescale, as doing so too early might make the fixed-point representation so small that the approximation errors destroy the message.

Encryption Parameters: In CKKS, all of the concerns about scaling factors, rescaling, and managing levels are intricately linked with selecting encryption parameters. Thus, a typical workflow when developing FHE applications involves a lot of trial-and-error, and repeatedly tweaking the parameters to achieve both correctness (accuracy) and performance. While some FHE libraries warn the user if the selected encryption parameters are secure, but not all of them do, so a developer may need to keep in mind security-related

limitations, which typically means upper-bounding Q for a given N.

3.2.3 Microsoft SEAL

Microsoft SEAL [149] is a software library that implements the RNS variant of the CKKS scheme. In SEAL, the modulus Q is a product of several prime factors of bit sizes up to 60 bits, and rescaling of ciphertexts is always done by dividing away these prime factors. The developer must choose these prime factors and order them correctly to achieve the desired rescaling behavior. SEAL automatically validates encryption parameters for correctness and security.

3.2.4 Threat Model

We assume a semi-honest threat model, as is typical for homomorphic encryption. This means that the party performing the computation (i.e., the server) is curious about the encrypted data but is guaranteed to run the desired operations faithfully. This model matches for example the scenario where the server is trusted, but a malicious party has read access to the server's internal state and/or communication between the server and the client.

3.3 EVA Language

The EVA framework uses a single language as its input format, intermediate representation, and executable format. The EVA language abstracts *batching-compatible* FHE schemes like BFV [59], BGV [26], and CKKS [41, 42],

Table 3.1: Types of values.

Type	Description
Cipher	An encrypted vector of fixed-point values.
Vector	A vector of 64-bit floating point values.
Scalar	A 64-bit floating point value.
Integer	A 32-bit signed integer.

Table 3.2: Instruction opcodes and their signatures.

Opcode	Signature	Restrictions
NEGATE	**Cipher** $\rightarrow$ **Cipher**	
ADD	**Cipher** $\times$ (**Vector**|**Cipher**) $\rightarrow$ **Cipher**	
SUB	**Cipher** $\times$ (**Vector**|**Cipher**) $\rightarrow$ **Cipher**	
MULTIPLY	**Cipher** $\times$ (**Vector**|**Cipher**) $\rightarrow$ **Cipher**	
ROTATELEFT	**Cipher** $\times$ **Integer** $\rightarrow$ **Cipher**	
ROTATERIGHT	**Cipher** $\times$ **Integer** $\rightarrow$ **Cipher**	
RELINEARIZE	**Cipher** $\rightarrow$ **Cipher**	Not in input
MODSWITCH	**Cipher** $\rightarrow$ **Cipher**	Not in input
RESCALE	**Cipher** $\times$ **Scalar** $\rightarrow$ **Cipher**	Not in input

Table 3.3: Instruction opcodes and their semantics (see Section 3.2 for details on semantics of the restricted instructions).

Opcode	Description
NEGATE	Negate each element of the argument.
ADD	Add arguments element-wise.
SUB	Subtract right argument from left one element-wise.
MULTIPLY	Multiply arguments element-wise (and multiply scales).
ROTATELEFT	Rotate elements to the left by given number of indices.
ROTATERIGHT	Rotate elements to the right by given number of indices.
RELINEARIZE	Apply relinearization.
MODSWITCH	Switch to the next modulus in the modulus chain.
RESCALE	Rescale the ciphertext (and divide scale) with the scalar.

and can be compiled to target libraries implementing those schemes. Input programs use a subset of the language that omits details specific to FHE, such as when to rescale. In this section, we describe the input language and its semantics, while Section 3.4 presents an overview of the compilation to an executable EVA program.

Types and Values: Table 3.1 lists the types that values in EVA programs may have. The vector types **Cipher** and **Vector** have a fixed power-of-two size for each input program. The power-of-two requirement comes from the target encryption schemes.

We introduce some notation for talking about types and values in EVA. For **Vector**, a literal value with elements a_i is written $[a_1, a_2, \ldots, a_i]$ or as a comprehension $[a_i \text{ for } i = 1 \ldots i]$. For the ith element of **Vector** a, we write a_i. For the product type (i.e., tuple) of two EVA types A and B, we write $A \times B$, and write tuple literals as (a, b) where $a \in A$ and $b \in B$.

Instructions: Programs in EVA are Directed Acyclic Graphs (DAGs), where each node represents a value available during execution. Example programs are shown in Figures 3.2(a) and 3.3(a). Nodes with one or more incoming edges are called *instructions*, which compute a new value as a function of its *parameter* nodes, i.e., the parent nodes connected to it. For the ith parameter of an instruction n, we write $n.parm_i$ and the whole list of parameter nodes is $n.parms$. Each instruction n has an opcode $n.op$, which specifies the operation

to be performed at the node. Note that the incoming edges are ordered, as it corresponds to the list of arguments. Table 3.2 lists all the opcodes available in EVA and Table 3.3 lists their semantics. The first group are opcodes that frontends may generate, while the second group lists FHE-specific opcodes that are inserted by the compiler. The key to the expressiveness of the input language are the RotateLeft and RotateRight instructions, which abstract rotation (circular shift) in *batching-compatible* FHE schemes.

Inputs: A node with no incoming edges is called a *constant* if its value is available at compile time and an *input* if its value is only available at run time. For a constant n, we write $n.value$ to denote the value. Inputs may be of any type, while constants can be any type except **Cipher**. This difference is due to the fact that the **Cipher** type is not fully defined before key generation time, and thus cannot have any values at compile time. The type is accessible as $n.type$.

Program: A program P is a tuple $(M, Insts, Consts, Inputs, Outputs)$, where M is the length of all vector types in P; *Insts*, *Consts* and *Inputs* are list of all instruction, constant, and input nodes, respectively; and *Outputs* identifies a list of instruction nodes as outputs of the program.

Execution Semantics: Next, we define execution semantics for EVA. Consider a dummy encryption scheme *id* that instead of encrypting **Cipher** values

just stores them as **Vector** values. In other words, the encryption and decryption are the identity function. This scheme makes homomorphic computation very easy, as every plaintext operation is its own homomorphic counterpart. Given a map $I : \mathit{Inputs} \rightarrow \textbf{Vector}$, let $\mathcal{E}_{id}(n)$ be the function that computes the value for node n recursively by using $n.\mathit{value}$ or $I(n)$ if n is a constant or input respectively and using $n.\mathit{op}$ and $\mathcal{E}_{id}()$ on $n.\mathit{parms}$ otherwise. Now for a program P, we further define its reference semantic as a function P_{id}, which given a value for each input node maps each output node in P to its resulting value:

$$P_{id} : \times_{n_i \in \mathit{Inputs}} n_i.\mathit{type} \rightarrow \times_{n_o \in \mathit{Outputs}} \textbf{Vector}$$
$$P_{id}(I(n_i^1), \ldots, I(n_i^{|\mathit{Inputs}|})) = (\mathcal{E}_{id}(n_o^1), \ldots, \mathcal{E}_{id}(n_o^{|\mathit{Outputs}|}))$$

These execution semantics hold for any encryption scheme, except that output is also encrypted (i.e., **Cipher** type).

Discussion on Rotation and Vector Sizes: The EVA language restricts the size for any **Cipher** or **Vector** input to be a power-of-2 so as to support execution semantics of RotateLeft and RotateRight instructions.

The target encryption schemes use the same vector size s_e $(= N/2)$ for all **Cipher** values during execution. However, the vector size s_i for an input could be different from s_e. Nevertheless, the target encryption scheme and the EVA language enforce the vector sizes s_e and s_i, respectively, to be

powers-of-2. EVA chooses encryption parameters for the target encryption scheme such that for all inputs i, $s_i \leq s_e$ (because N can be increased trivially without hurting correctness or security — note that increasing N will hurt performance).

For a **Cipher** or **Vector** input i such that $s_i < s_e$, EVA constructs a vector (before encryption for **Cipher**) i' with s_e/s_i copies of the vector i contiguously such that $s'_i = s_e$, and replaces i with i' in the program. For example, if an input $a = [a_1, a_2]$ and $s_e = 4$, then EVA constructs $a' = [a_1, a_2, a_1, a_2]$ and replaces a with a'. RotateRight or RotateRight instruction on the original vector i and the constructed larger vector i' have the same result on their first s_i elements. This is feasible because s_i divides s_e. EVA thus ensures that the execution semantics of the EVA program holds for any encryption scheme.

Implementation: As shown in Figure 3.1, the EVA language has a serialized format defined using Protocol Buffers [75], a language and platform neutral data serialization format. Additionally, the EVA language has an in-memory graph representation that is designed for efficient analysis and transformation, which is discussed in Section 3.4.

3.4 Overview of EVA Compiler

In this section, we describe how to use the EVA compiler (Section 3.4.1). We then describe the constraints on the code generated by EVA (Section 3.4.2).

```
syntax = ''proto3'';

package EVA;

enum OpCode {
  UNDEFINED_OP = 0;
  NEGATE = 1;
  ADD = 2;
  SUB = 3;
  MULTIPLY = 4;
  SUM = 5;
  COPY = 6;
  ROTATE_LEFT = 7;
  ROTATE_RIGHT = 8;
  RELINEARIZE = 9;
  MOD_SWITCH = 10;
  RESCALE = 11;
  NORMALIZE_SCALE = 12;
}

enum ObjectType {
  UNDEFINED_TYPE = 0;
  SCALAR_CONST = 1;
  SCALAR_PLAIN = 2;
  SCALAR_CIPHER = 3;
  VECTOR_CONST = 4;
  VECTOR_PLAIN = 5;
  VECTOR_CIPHER = 6;
}

message Object {
  uint64 id = 1;
}

message Instruction {
  Object output = 1;
  OpCode op_code = 2;
  repeated Object args = 3;
}

message Vector {
  repeated double elements =
    1;
}

message Input {
  Object obj = 1;
  ObjectType type = 2;
  double scale = 3;
}

message Constant {
  Object obj = 1;
  ObjectType type = 2;
  double scale = 3;
  Vector vec = 4;
}

message Output {
  Object obj = 1;
  double scale = 2;
}

message Program {
  uint64 vec_size = 1;
  repeated Constant constants
    = 2;
  repeated Input inputs = 3;
  repeated Output outputs =
    4;
  repeated Instruction insts
    = 5;
}
```

Figure 3.1: The EVA language definition using Protocol Buffers.

Finally, we give an overview of the execution flow of the EVA compiler (Section 3.4.3).

3.4.1 Using the Compiler

In this thesis, we present the EVA compiler for the RNS variant of the CKKS scheme [41] and its implementation in the SEAL library [149]. Targeting EVA for other FHE libraries [81, 92, 138] implementing CKKS [41, 42] would be straightforward. The EVA compiler can also be adapted to support other *batching-compatible* FHE schemes like BFV [59] and BGV [26].

The EVA compiler takes a program in the EVA language as input. Along with the program, it needs the fixed-point scales or precisions for each input in the program and the desired fixed-point scales or precisions for each output in the program. The compiler then generates a program in the EVA language as output. In addition, it generates a vector of bit sizes that must be used to generate the encryption parameters as well as a set of rotation steps that must be used to generate the rotation keys. The encryption parameters and the rotations keys thus generated are required to execute the generated EVA program.

While the input and the output programs are in the EVA language, the set of instructions allowed in the input and the output are distinct, as listed in Table 3.2. The Relinearize, Rescale, and ModSwitch instructions require understanding the intricate details of the FHE scheme. Hence, they are omitted from the input program. Note that we can make these instructions

optional in the input and the compiler can handle it if they are present, but for the sake of exposition, we assume that the input does not have these instructions.

The input scales and the desired output scales affect the encryption parameters, and consequently, the performance and accuracy of the generated program. Choosing the right values for these is a trade-off between performance and accuracy (while providing the same security). Larger values lead to larger encryption parameters and more accurate but slower generated program, whereas smaller values lead to smaller encryption parameters and less accurate but faster generated program. Profiling techniques like those used in CHET [56] can be used to select the appropriate values.

3.4.2 Motivation and Constraints

There is a one-to-one mapping between instructions in the EVA language (Table 3.2) and instructions in the RNS-CKKS scheme. However, the input program cannot be directly executed. Firstly, encryption parameters are required to ensure that the program would be accurate. EVA can simply determine the bit sizes that is required to generate the parameters. However, this is insufficient to execute the program correctly because some instructions in the RNS-CKKS scheme have restrictions on their inputs. If these restrictions are not met, the instructions would just throw an exception at runtime.

Each ciphertext in the RNS-CKKS scheme has a coefficient modulus q

$(Q = \prod_{i=1}^{r} q_i)^2$ and a fixed-point *scale* associated with it. All freshly encrypted ciphertexts have the same q but they may have different *scale*. The following constraints apply for the binary instructions involving two ciphertexts:

$$n.parm_1.modulus = n.parm_2.modulus \quad \text{if } n.op \in \{\textsc{Add}, \textsc{Sub}, \textsc{Multiply}\} \tag{3.1}$$

$$n.parm_1.scale = n.parm_2.scale \quad \text{if } n.op \in \{\textsc{Add}, \textsc{Sub}\} \tag{3.2}$$

In the rest of this chapter, whenever we mention Add regarding constraints, it includes both Add and Sub.

Consider the example to compute x^2y^3 for ciphertexts x and y in Figure 3.2 (viewed as a dataflow graph). Constraint 3.2 is trivially satisfied because they are no Add instructions. Only Rescale and ModSwitch instructions modify q. Constraint 3.1 is also trivially satisfied due to the absence of these instructions. Nonetheless, without the use of Rescale instructions, the scales and the *noise* of the ciphertexts would grow exponentially with the multiplicative depth of the program (i.e., maximum number of Multiply nodes in any path) and consequently, the $\log_2$ of the coefficient modulus product Q required for the input would grow exponentially. Instead, using Rescale instructions ensures that $\log_2 Q$ would only grow linearly with the multiplicative depth of the program.

[2]In SEAL, if the coefficient modulus q is $\{q_1, q_2, ..., q_r\}$, then q_i is a prime close to a power-of-2. EVA compiler (and the rest of this chapter) assumes q_i is the corresponding power-of-2 instead. To resolve this discrepancy, when a Rescale instruction divides the scale by the prime, the scale is adjusted (by the EVA executor) as if it was divided by the power-of-2 instead.

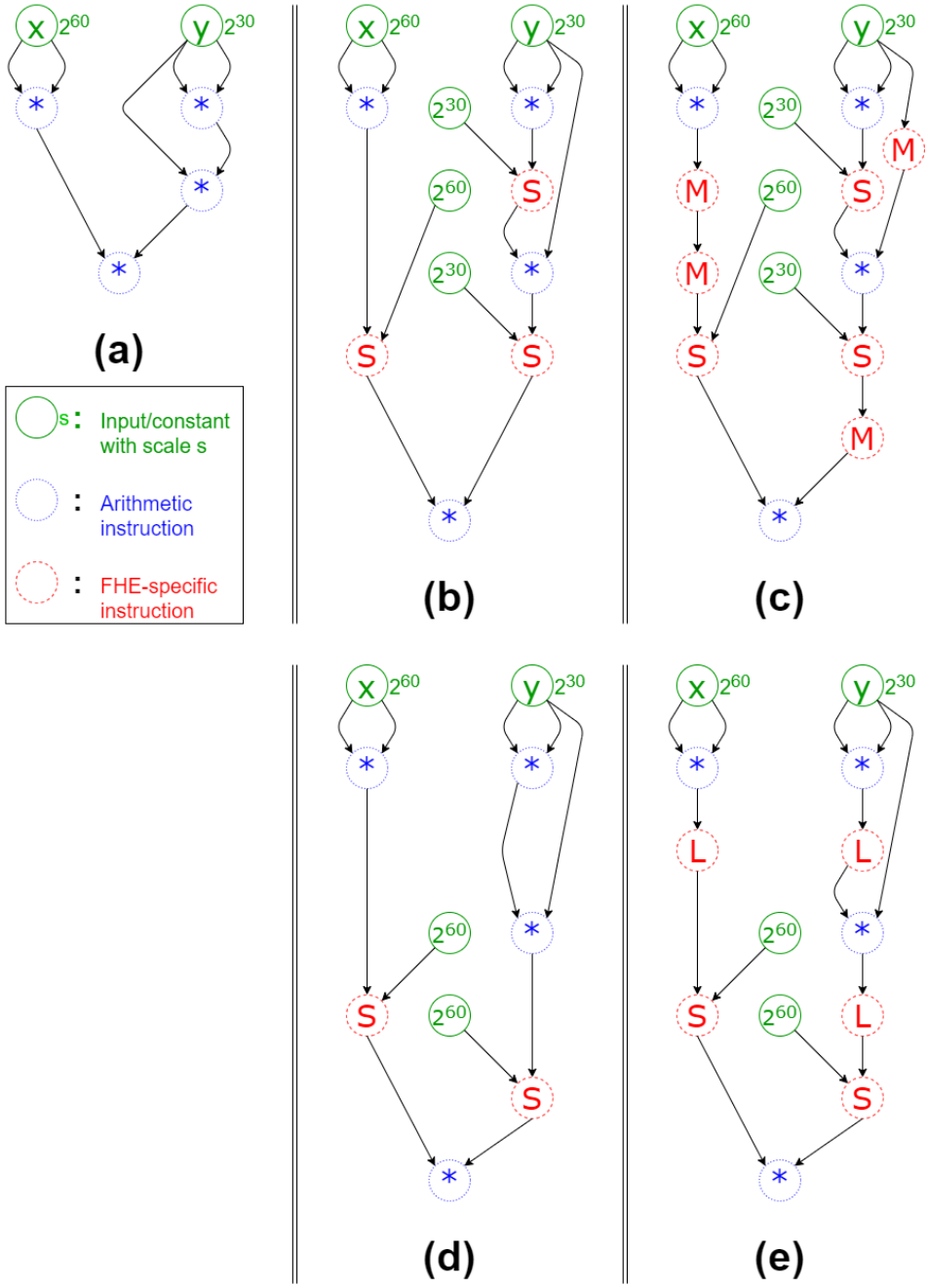


Figure 3.2: x^2y^3 example in EVA: (a) input; (b) after ALWAYS-RESCALE; (c) after ALWAYS-RESCALE & MODSWITCH; (d) after WATERLINE-RESCALE; (e) after WATERLINE-RESCALE & RELINEARIZE (S: RESCALE, M: MODSWITCH, L: RELINEARIZE).

In Figure 3.2, the output has a scale of $2^{60} * 2^{60} * 2^{30} * 2^{30} * 2^{30}$ (with $x.scale = 2^{60}$ and $y.scale = 2^{30}$). This would require Q to be at least $2^{210} * s_o$, where s_o is the user-provided desired output scale. To try to reduce this, one can insert RESCALE after every MULTIPLY, as shown in Figure 3.2(b). However, this yields an invalid program because it violates Constraint 3.1 for the last (bottom) MULTIPLY (and there is no way to choose the same q for both x and y). To satisfy this constraint, MODSWITCH instructions can be inserted, as shown in Figure 3.2(c). Both RESCALE and MODSWITCH drop the first element in their input q (or *consume the modulus*), whereas RESCALE also divides the scale by the given scalar (which is required to match the first element in q). The output now has a scale of $2^{60} * 2^{30}$. This would require choosing $q = \{2^{30}, 2^{30}, 2^{60}, 2^{60}, 2^{30}, s_o\}$. Thus, although Figure 3.2(c) executes more instructions than Figure 3.2(a), it requires the same Q. A better way to insert RESCALE instructions is shown in Figure 3.2(d). This satisfies Constraint 3.1 without MODSWITCH instructions. The output now has a scale of $2^{60} * 2^{30}$. We can choose $q = \{2^{60}, 2^{60}, 2^{30}, s_o\}$, so $Q = 2^{150} * s_o$. Hence, this program is more efficient than the input program.

If the computation was modified to $x^2 + y^3$ in Figure 3.2(d), then the last (bottom) MULTIPLY would be replaced by ADD and the program would violate Constraint 3.2 as ADD would have operands with scales 2^{60} and 2^{30}. Consider a similar but simpler example of $x^2 + x$ in Figure 3.3(a). One way to satisfy Constraints 3.1 and 3.2 is by adding RESCALE and MODSWITCH, as shown in Figure 3.3(b), which would require $q = \{2^{30}, 2^{30}, s_o\}$. Another

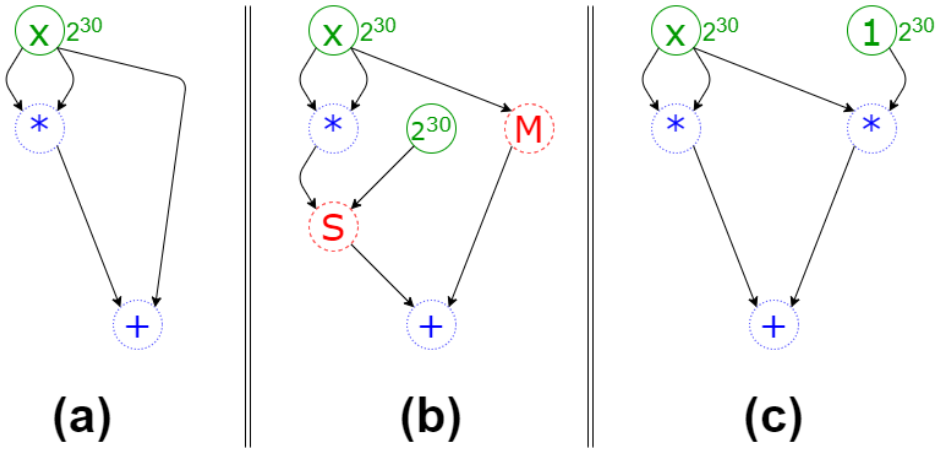


Figure 3.3: $x^2 + x$ example in EVA: (a) input; (b) after ALWAYS-RESCALE & MODSWITCH; (c) after MATCH-SCALE.

way is to introduce MULTIPLY of x and a constant 1 with 2^{30} scale to match the scale of ADD operands, as shown in Figure 3.3(c), which would require $q = \{2^{60}, s_o\}$. Although the product $Q = 2^{60} * s_o$ is same in both cases, the modulus length r is different. Hence, the program in Figure 3.3(c) is more efficient due to a smaller r.

MULTIPLY has another constraint. Each ciphertext consists of 2 or more polynomials. MULTIPLY of two ciphertexts with k and l polynomials yields a ciphertext with $k + l + 1$ polynomials. Nevertheless, fewer the polynomials faster the MULTIPLY, so we enforce it to be the minimum:

$$\begin{gathered} \forall i \; n.parm_i.num\_polynomials = 2 \\ \text{if } n.op \in \{\text{MULTIPLY}\} \end{gathered} \tag{3.3}$$

RELINEARIZE reduces the number of polynomials in a ciphertext to 2. This constraint guarantees that any relinearization in the program would reduce the number of polynomials in a ciphertext from 3 to 2, thereby ensuring

that only one public key is sufficient for all relinearizations in the program. RELINEARIZE can be inserted in the program in Figure 3.2(d) to satisfy this constraint, as shown in Figure 3.2(e).

Finally, we use s_f to denote the maximum allowed rescale value in the rest of this thesis ($\log_2 s_f$ is also the maximum bit size that can be used for encryption parameters), i.e.,

$$\begin{gathered} n.parm_2.value \leq s_f \\ \text{if } n.op \in \{\text{RESCALE}\} \end{gathered} \tag{3.4}$$

In the SEAL library, $s_f = 2^{60}$ (which enables a performant implementation by limiting scales to machine-sized integers).

To summarize, FHE schemes (or libraries) are tedious for a programmer to reason about, due to all their cryptographic constraints. Programmers find it even more tricky to satisfy the constraints in a way that optimizes performance. *The EVA compiler hides such cryptographic details from the programmer while optimizing the program.*

3.4.3 Execution Flow of the Compiler

As mentioned in Section 3.3, the in-memory internal representation of the EVA compiler is an **Abstract Semantic Graph**, also known as a **Term Graph**, of the input program. In the rest of this thesis, we will use the term *graph* to denote an Abstract Semantic Graph. In this in-memory representation, each node can access both its parents and its children, and for each output, a distinct leaf node is added as a child. It is straightforward to

Algorithm 1: Execution of EVA compiler.

Input : Program P_i in EVA language
Input : Scales S_i for inputs in P_i
Input : Desired scales S_d for outputs in P_i
Output: Program P_o in EVA language
Output: Vector B_v of bit sizes
Output: Set R_s of rotation steps

```
1 P_o = Transform(P_i, S_i)
2 if Validate(P_o) == Failed then
3 |   Throw an exception
4 end
5 B_v = DetermineParameters(P_o, S_i, S_d)
6 R_s = DetermineRotationSteps(P_i, S_i)
```

construct the graph from the EVA program and vice-versa, so we omit the details. We use the terms program and graph interchangeably in the rest of the thesis.

Algorithm 1 presents the execution flow of the compiler. There are four main steps, namely transformation, validation, parameters selection, and rotations selection. The transformation step takes the input program and modifies it to satisfy the constraints of all instructions, while optimizing it. In the next step, the transformed program is validated to ensure that no constraints are violated. If any constraints are violated, then the compiler throws an exception. By doing this, EVA ensures that executing the output program will never lead to a runtime exception thrown by the FHE library. Finally, for the validated output program, the compiler selects the bit sizes and the rotation steps that must be used to determine the encryption parameters and the public keys required for rotations respectively, before executing the

output program. The transformation step involves rewriting the graph, which is described in detail in Section 3.5. The other steps only involve traversal of the graph (without changing it), which is described in Section 3.6.

3.5 Transformations in EVA Compiler

In this section, we describe the key graph transformations in the EVA compiler. We first describe a general graph rewriting framework (Section 3.5.1). Then, we describe the graph transformation passes (Sections 3.5.2 and 3.5.3).

3.5.1 Graph Rewriting Framework

A graph transformation can be captured succinctly using graph *rewrite rules* (or term rewrite rules). These rules specify the conditional transformation of a subgraph (or an expression) and the graph transformation consists of transforming all applicable subgraphs (or expressions) in the graph (or program). The graph nodes have read-only properties like the opcode and number of parents. In a graph transformation, some state or data may be stored on each node in the graph and the rewrite rules may read and update the state.

The rewrite rules specify local operations on a graph and the graph transformation itself is composed of applying these operations wherever needed. The schedule in which these local operations are applied may impact the correctness or efficiency of the transformation. Consider two schedules:

1. Forward pass from roots to leaves of the graph: a node is scheduled for

rewriting only after all its parents have already been rewritten.

2. Backward pass from leaves to roots of the graph: a node is scheduled for rewriting only after all its children have already been rewritten.

Note that the rewriting operation may not do any modifications if its condition does not hold. In forward pass, state (or data) flows from parents to children. Similarly, in backward pass, state (or data) flows from children to parents. In general, multiple forward or backward passes may be needed to apply the rewrite rules until quiescence (no change).

EVA includes a graph rewriting framework for arbitrary rewrite rules for a subgraph that consists of a node along with its parents or children. The rewrite rules for each graph transformation pass in EVA are defined in Figure 3.4. A single backward pass is sufficient for EAGER-MODSWITCH, while a single forward pass is sufficient for the rest. The rewrite rules assume the passes are applied in a specific order: WATERLINE-RESCALE, EAGER-MODSWITCH, MATCH-SCALE, and RELINEARIZE (ALWAYS-RESCALE and LAZY-MODSWITCH are not used but defined only for clarity). For the sake of exposition, we will first describe RELINEARIZE pass before describing the other passes.

3.5.2 Relinearize Insertion Pass

Each ciphertext is represented as 2 or more polynomials. Multiplying two ciphertexts each with 2 polynomials yields a ciphertext with 3 polynomi-

$$\text{ALWAYS} - \text{RESCALE} \frac{\begin{array}{c} n \in \mathit{Insts} \\ n.op = \textsc{Multiply} \qquad N_{ck} = \{(n_c, k) \,|\, n_c.parm_k = n\} \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_s\} \qquad n_s.op \leftarrow \textsc{Rescale} \\ n_s.parm_1 \leftarrow n \qquad n_s.parm_2 \leftarrow \min(\forall j,\ n.parm_j.scale) \\ \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_s \end{array}}$$

$$\text{WATERLINE} - \text{RESCALE} \frac{\begin{array}{c} n \in \mathit{Insts} \\ n.op = \textsc{Multiply} \qquad N_{ck} = \{(n_c, k) \,|\, n_c.parm_k = n\} \\ n.scale \leftarrow n.parm_1.scale * n.parm_2.scale \\ (n.scale / s_f) \geq \max(\forall n_j \in \{\mathit{Consts}, \mathit{Inputs}\},\ n_j.scale) \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_s\} \qquad n_s.op \leftarrow \textsc{Rescale} \\ n_s.parm_1 \leftarrow n \qquad n_s.parm_2 \leftarrow s_f \\ \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_s \end{array}}$$

$$\text{LAZY} - \text{MODSWITCH} \frac{\begin{array}{c} n \in \mathit{Insts} \qquad n.op \in \{\textsc{Add}, \textsc{Sub}, \textsc{Multiply}\} \\ n.parm_i.level > n.parm_j.level \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_m\} \qquad n_m.op \leftarrow \textsc{ModSwitch} \\ n_m.parm_1 \leftarrow n.parm_j \qquad n.parm_j \leftarrow n_m \end{array}}$$

$$\text{EAGER} - \text{MODSWITCH} \frac{\begin{array}{c} n \in \{\mathit{Insts}, \mathit{Consts}, \mathit{Inputs}\} \qquad n_c^1.parm_i = n \\ n_c^2.parm_j = n \qquad n_c^1.parm_i.rlevel > n_c^2.parm_j.rlevel \\ N_{ck} = \{(n_c, k) \,|\, n_c.parm_k = n \wedge n_c.parm_k.rlevel = n_c^2.parm_j.rlevel\} \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_m\} \qquad n_m.op \leftarrow \textsc{ModSwitch} \\ n_m.parm_1 \leftarrow n \qquad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_m \end{array}}$$

$$\text{MATCH} - \text{SCALE} \frac{\begin{array}{c} n \in \mathit{Insts} \\ n.op \in \{\textsc{Add}, \textsc{Sub}\} \qquad n.parm_i.scale > n.parm_j.scale \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_t\} \qquad \mathit{Consts} \leftarrow \mathit{Consts} \cup \{n_c\} \\ n_c.value \leftarrow n.parm_i.scale / n.parm_j.scale \\ n_t.op \leftarrow \textsc{Multiply} \qquad n_t.parm_1 \leftarrow n.parm_j \\ n_t.parm_2 \leftarrow n_c \qquad n.parm_j \leftarrow n_t \end{array}}$$

$$\text{RELINEARIZE} \frac{\begin{array}{c} n \in \mathit{Insts} \\ n.op = \textsc{Multiply} \qquad n.parm_1.type = n.parm_2.type = \textbf{Cipher} \\ N_{ck} = \{(n_c, k) \,|\, n_c.parm_k = n\} \end{array}}{\begin{array}{c} \mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_l\} \qquad n_l.op \leftarrow \textsc{Relinearize} \\ n_l.parm_1 \leftarrow n \qquad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_l \end{array}}$$

Figure 3.4: Graph rewriting rules (each rule is a transformation pass) in EVA (s_f: maximum allowed rescale value).

als. The RELINEARIZE instruction reduces a ciphertext to 2 polynomials. To satisfy Constraint 3.3, EVA must insert RELINEARIZE after MULTIPLY of two nodes with **Cipher** type and before another such MULTIPLY.

The RELINEARIZE rewrite rule (Figure 3.4) is applied for a node n only if it is a MULTIPLY operation and if both its parents (or parameters) have **Cipher** type. The transformation in the rule inserts a RELINEARIZE node n_l between the node n and its children. In other words, the new children of n will be only n_l and the children of n_l will be the old children of n. For the example graph in Figure 3.2(d), applying this rewrite rule transforms the graph into the one in Figure 3.2(e).

Optimal placement of relinearization is an NP-hard problem [37]. Our relinearization insertion pass is a simple way to enforce Constraint 3.3. More advanced relinearization insertion, with or without Constraint 3.3, is left for future work.

3.5.3 Rescale and ModSwitch Insertion Passes

Goal: The RESCALE and MODSWITCH nodes (or instructions) must be inserted such that they satisfy Constraint 3.1, so the goal of the RESCALE and MODSWITCH insertion passes is to insert them such that the coefficient moduli of the parents of any ADD and MULTIPLY node are equal.

While satisfying Constraint 3.1 is sufficient for correctness, performance depends on where RESCALE and MODSWITCH are inserted (as illustrated in Section 3.4.2). Different choices lead to different coefficient modulus $q =$

$\{q_1, q_2, ..., q_r\}$, and consequently, different polynomial modulus N for the roots (or inputs) to the graph (or program). Larger values of N and r increase the cost of every FHE operation and the memory of every ciphertext. N is a non-decreasing function of $Q = \prod_{i=1}^{r} q_i$ (i.e., if Q grows, N either remains the same or grows as well). Minimizing both Q and r is a hard problem to solve. However, reducing Q is only impactful if it reduces N, which is unlikely as the threshold of Q, for which N increases, grows exponentially. Therefore, *the goal of EVA is to yield the optimal* r, which may or may not yield the optimal N.

Constrained-Optimization Problem: The only nodes that modify a ciphertext's coefficient modulus are RESCALE and MODSWITCH nodes; that is, they are the only ones whose output ciphertext has a different coefficient modulus than that of their input ciphertext(s). Therefore, the coefficient modulus of the output of a node depends only on the RESCALE and MODSWITCH nodes in the path from the root to that node. To illustrate their relation, we define the term *rescale chain*.

Definition 1. *Given a directed acyclic graph G = (V, E):*

For $n_1, n_2 \in V$, n_1 *is a* parent *of* n_2 *if* $\exists(n_1, n_2) \in E$.

A node $r \in V$ *is a* root *if* $r.type =$ **Cipher** *and* $\nexists n \in V$ *s.t.* n *is a parent of* r.

Definition 2. *Given a directed acyclic graph G = (V, E):*

A path *p from a node $n_0 \in V$ to a node $n \in V$ is a sequence of nodes $p_0, ..., p_l$ s.t. $p_0 = n_0$, $p_l = n$, and $\forall 0 \leq i < l, p_i \in V$ and p_i is a parent of p_{i+1}. A path p is said to be* simple *if $\forall 0 < i < l, p_i.op \neq$* RESCALE *and $p_i.op \neq$* MODSWITCH.

Definition 3. *Given a directed acyclic graph G = (V, E):*

A rescale path *p to a node $n \in V$ is a sequence of nodes $p_0, ..., p_l$ s.t. ($\forall 0 \leq i \leq l, p_i.op \in \{$*RESCALE, MODSWITCH*$\}$), $\exists$ a simple path from a root to p_0, $\exists$ a simple path from p_l to n, ($\forall 0 \leq i < l, \exists$ a simple path from p_i to p_{i+1}), and ($n.op =$* RESCALE *or $n.op =$* MODSWITCH*) $\implies$ ($p_l = n$) .*

A rescale chain *of a node $n \in V$ is a vector c s.t. $\exists$ a rescale path p to n and ($\forall 0 \leq i < |p|$, ($p_i.op =$* MODSWITCH *$\implies c_i = \infty$) and ($p_i.op =$* RESCALE *$\implies c_i = p_i.parm_2.value$)). Note that ∞ is used here to distinguish* MODSWITCH *from* RESCALE *in the rescale chain.*

A rescale chain c of a node n and c' of a node n' are equal *if ($|c| = |c'|$ and ($\forall 0 \leq i < |c|$, $c_i = c'_i$ or $c_i = \infty$ or $c'_i = \infty$)).*

A rescale chain c of a node $n \in V$ is conforming *if $\forall$ rescale chain c' of n, c is equal to c'.*

Note that all the roots in the graph have the same coefficient modulus. Therefore, for nodes n_1 and n_2, the coefficient modulus of the output of n_1 is equal to that of n_2 if and only if there exists conforming rescale chains for n_1 and n_2, and the conforming rescale chain of n_1 is equal to that of n_2. Thus, we need to solve two problems simultaneously:

- Constraints: Ensure the conforming rescale chains of the parents of any MULTIPLY or ADD node are equal.
- Optimization: Minimize the length of the rescale chain of every node.

Outline: In general, the constraints problem can be solved in two steps:

- Insert RESCALE in a pass (to reduce exponential growth of scale and noise).
- Insert MODSWITCH in another pass so that the constraints are satisfied.

The challenge is in solving this problem in this way, while yielding the desired optimization.

Always Rescale Insertion: A naive approach of inserting RESCALE is to insert it after every MULTIPLY of **Cipher** nodes. We call this approach as *always rescale* and define it in the ALWAY-RESCALE rewrite rule in Figure 3.4. Consider the example in Figure 3.2(a). Applying this rewrite rule on this example yields the graph in Figure 3.2(b). For some MULTIPLY nodes (e.g., the bottom one), the conforming rescale chains of their parents do not exist or do not match. To satisfy these constraints, MODSWITCH nodes can be inserted appropriately, as shown in Figure 3.2(c) (we omit defining this rule because it would require multiple forward passes). The conforming rescale chain length for the output is now more more than the multiplicative depth of the graph.

Thus, *always rescale* and its corresponding modswitch insertion may lead to a larger coefficient modulus (both in the number of elements and their product) than not inserting any of them.

Insight: Consider that all the roots in the graph have the same scale s. For example in Figure 3.2(a), let $x.scale = 2^{30}$ instead of 2^{60}. Then, after *always rescale* (replace 2^{60} with 2^{30} in Figure 3.2(b)), the only difference between the rescale chains of a node n would be their length and not the values in it. This is the case even when roots may have different scales as long as all RESCALE nodes rescale by the same value s. Then, a conforming rescale chain c_n for n can be obtained by adding MODSWITCH nodes in the smaller chain(s). Thus, $|c_n|$ would not be greater than the multiplicative depth of n. The first key insight of EVA is that using the same rescale value for all RESCALE nodes ensures that $|c_o|$ cannot be greater than the multiplicative depth of o (a tight upper bound). The multiplicative depth of a node n is not a tight lower bound for $|c_n|$, as shown in Figure 3.2(d). The second key insight of EVA is that using the maximum rescale value s_f (satisfying Constraint 3.4) for all RESCALE nodes minimizes $|c_o|$ because it minimizes the number of RESCALE nodes in any path.

Waterline Rescale Insertion: Based on our insights, the value to rescale is fixed to s_f ($= 2^{60}$ in SEAL). That does not address the question of when to insert RESCALE nodes. If the scale after RESCALE becomes too small, then

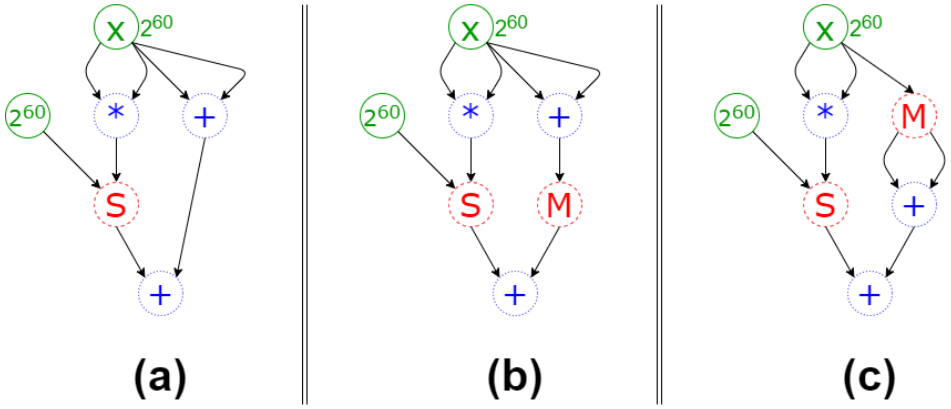


Figure 3.5: $x^2 + x + x$ in EVA: (a) after WATERLINE-RESCALE (b) after WATERLINE-RESCALE & LAZY-MODSWITCH; (c) after WATERLINE-RESCALE & EAGER-MODSWITCH.

the computed message may lose accuracy irrevocably. We call the minimum required scale as *waterline* and use s_w to denote it. We choose s_w to be maximum of the scales of all roots. Consider a MULTIPLY node n whose scale after multiplication is s_n. Then, a RESCALE in inserted between n and its children only if the scale after RESCALE is above the *waterline*, i.e., $(s_n/s_f) \geq s_w$. We call this approach as *waterline rescale* and define the WATERLINE-RESCALE rewrite rule in Figure 3.4. This rule (assuming $s_w = 2^{30}$ instead of 2^{60}) transforms the graph in Figure 3.2(a) to the one in Figure 3.2(d).

ModSwitch Insertion: For a node n, let $n.level$ denote its conforming rescale chain length. Let $n.rlevel$ denote the conforming rescale chain length of n in the transpose graph. A naive way to insert MODSWITCH is to find a ADD or MULTIPLY node for which *level* of the parents do not match and insert the appropriate number of MODSWITCH nodes between one of the parents

and the node. We call this *lazy* insertion and define the LAZY-MODSWITCH rewrite rule in Figure 3.4. We call inserting it at the earliest feasible edge in the graph as *eager* insertion. The EAGER-MODSWITCH rewrite rule (Figure 3.4) finds a node for which *rlevel* of the children do not match and inserts the appropriate number of ModSwitch nodes between some of the children and itself. If the *rlevel* of the roots do not match, then there is another rule (omitted in Figure 3.4 for brevity) that inserts the appropriate ModSwitch nodes between some of the roots and their children.

Consider the $x^2 + x + x$ example in Figure 3.5(a). Applying the LAZY-MODSWITCH and EAGER-MODSWITCH rewrite rules yields the graphs in Figures 3.5(b) and (c) respectively. The operands of Add after eager insertion use a smaller coefficient modulus than after lazy insertion, so Add would be faster if eager insertion is used. Thus, eager insertion leads to similar or more efficient code than lazy insertion.

Matching Scales: As illustrated in Section 3.4.2, it is easy to match scales of parents of Add by multiplying one of the parents and 1 with the appropriate scale. The MATCH-SCALE rewrite rule (Figure 3.4) takes this simple approach to satisfy Constraint 3.2 while avoiding introduction of any additional Rescale or ModSwitch. For the example graph in Figure 3.3(a), applying this rewrite rule transforms the graph into the one in Figure 3.3(c).

Optimality: EVA selects encryption parameters (see Section 3.6.2) s.t. $r = \max(\forall o \in \{Outputs\}, 1 + |c_o| + \lceil \frac{o.scale * s_o}{s_f} \rceil)$, where s_o is the desired scale for the output o. WATERLINE-RESCALE is the only pass that determines $|c_o|$ and *o.scale* for any output o (neither LAZY-MODSWITCH nor MATCH-SCALE modify that). If $|c_o|$ is decreased by 1 (an element s_f from c_o is removed), then *o.scale* would increase by at least s_f, so it would not decrease r. Due to *waterline rescale*, $o.scale < s_w * s_f$, so RESCALE cannot be inserted to reduce *o.scale* by at least s_f (because the minimum required scale is s_w). Thus, EVA yields the minimal or optimal r.

3.6 Analysis in EVA Compiler

In this section, we briefly describe our graph traversal framework (Section 3.6.1) and a few analysis passes (Section 3.6.2).

3.6.1 Graph Traversal Framework and Executor

EVA's graph traversal framework allows either a forward traversal or a backward traversal of the graph. In the forward traversal pass, a node is visited only after all its parents are visited. Similarly, in the backward traversal pass, a node is visited only after all its children are visited. Graph traversals do not modify the structure of the graph (unlike graph rewriting) but a state on each node can be maintained during the traversal. A single pass is sufficient to perform forward or backward data-flow analysis of the graph because the graph is acyclic. Execution of the graph is a forward traversal of the graph,

so uses the same framework.

Parallel Implementation: We implement an executor for the generated EVA program using the traversal framework. A node is said to be *ready* or *active* if all its parents (or children) in forward (or backward) pass have already been visited. These active nodes can be scheduled to execute in parallel as each active node only updates its own state (i.e., there are no conflicts). For example in Figure 3.2(e), the parents of the bottom MULTIPLY can execute in parallel. Each FHE instruction (node) can take a significant amount of time to execute, so it is useful to exploit parallelism among FHE instructions. The EVA executor automatically parallelizes the generated EVA program by implementing a parallel graph traversal using the Galois [2, 134] parallel library.

A node is said to *retire* if all its children (or parents) in forward (or backward) pass have already been visited. The state for the retired nodes will no longer be accessed, so it can be reused for other nodes. In Figure 3.2(e), the ciphertext for x^2 can be reused after the RELINEARIZE is executed. The EVA executor automatically reuses the memory used for encrypted messages, thereby reducing the memory consumed.

3.6.2 Analysis Passes

Validation Passes: We implement a validation pass for each of the constraints in Section 3.4.2. All are forward passes. The first pass computes the rescale chains for each node and asserts that it is conforming. It also asserts

that the conforming rescale chains of parents of ADD and MULTIPLY match, satisfying Constraint 3.1. The second and third passes compute a *scale* and *num_polynomials* for each node respectively and assert that Constraint 3.2 and 3.3 is satisfied respectively. If any assertion fails, an exception in thrown at compile-time. Thus, these passes elide runtime exceptions thrown by SEAL.

Encryption Parameter Selection Pass: Akin to encryption selection in CHET [56], the encryption parameter selection pass in EVA computes the conforming *rescale chain* and the scale for each node. For each leaf or output o after the pass, let c_o be the conforming rescale chains of o without ∞ in it and let $s'_o = s_o * o.scale$, where s_o is the desired output scale. s'_o is factorized into $s_0 * s_1 * ... * s_k$ such that s_k is a power-of-two, $s_k \leq s_f$ ($= 2^{60}$ in SEAL), and $\forall 0 \leq i < k, s_i = s_f$. Let $|s'_o|$ denote the number of factors of s'_o. Then EVA finds the output m with the maximum $|c_m| + |s'_m|$. The factors of s_m are appended to c_m and s_f (the *special prime* that is consumed during encryption) is inserted at the beginning of c_m. For each element s in c_m, $\log_2 s$ is applied to obtain a vector of bit sizes, which is then returned.

Rotation Keys Selection Pass: Similar to rotation keys selection in CHET [56], EVA's rotation keys selection pass computes and returns the set of unique step counts used among all ROTATELEFT and ROTATERIGHT nodes in the graph.

```
from EVA import *
def sqrt(x):
  return x*constant(scale, 2.214) +
    (x**2)*constant(scale, -1.098) +
    (x**3)*constant(scale, 0.173)
program = Program(vec_size=64*64)
scale = 30
with program:
  image = inputEncrypted(scale)
  F = [[-1, 0, 1],
       [-2, 0, 2],
       [-1, 0, 1]]
  for i in range(3):
    for j in range(3):
      rot = image << (i*64+j)
      h = rot * constant(scale, F[i][j])
      v = rot * constant(scale, F[j][i])
      first = i == 0 and j == 0
      Ix = h if first else Ix + h
      Iy = v if first else Iy + v
  d = sqrt(Ix**2 + Iy**2)
  output(d, scale)
```

Figure 3.6: PyEVA program for Sobel filtering 64×64 images. The sqrt function evaluates a 3rd degree polynomial approximation of square root.

3.7 Frontends of EVA

The various transformations described so far for compiling an input EVA program into an executable EVA program make up the *backend* in the EVA compiler framework. In this section, we describe two *frontends* for EVA, that make it easy to write programs for EVA.

3.7.1 PyEVA

We have built a general-purpose frontend for EVA as a DSL embedded into Python, called PyEVA. Consider the PyEVA program in Figure 3.6 for Sobel filtering, which is a form of edge detection in image processing. The `class Program` is a wrapper for the Protocol Buffer [75] format for EVA programs shown in Figure 3.1. It includes a context manager, such that inside a `with program:` block all operations are recorded in `program`. For example, the `inputEncrypted` function inserts an input node of type **Cipher** into the program currently in context and additionally returns an instance of `class Expr`, which stores a reference to the input node. The expression additionally overrides Python operators to provide the simple syntax seen here.

3.7.2 EVA for Neural Network Inference

CHET [56] is a compiler for evaluating neural networks on encrypted inputs. The CHET compiler receives a neural network as a graph of high-level tensor operations, and through its kernel implementations, analyzes and executes these neural networks against FHE libraries. CHET lacks a proper backend and operates more as an interpreter coupled with automatically chosen high-level execution strategies.

We modified CHET to use the EVA compiler as a backend. CHET uses an interface called *Homomorphic Instruction Set Architecture* (HISA) as a common abstraction for different FHE libraries. In order to make CHET generate EVA programs, we introduce a new HISA implementation that instead

of calling homomorphic operations inserts instructions into an EVA program. This decouples the generation of the program from its execution. We make use of CHET's data layout selection optimization, but not its encryption parameter selection functionality, as this is already provided in EVA. Thus, EVA subsumes CHET.

3.8 Experimental Evaluation

In this section, we first describe our experimental setup (Section 3.8.1). We then describe our evaluation of homomorphic neural network inference (Section 3.8.2) and homomorphic arithmetic, statistical machine learning, and image processing applications (Section 3.8.3).

3.8.1 Experimental Setup

All experiments were conducted on a 4 socket machine with Intel Xeon Gold 5120 2.2GHz CPU with 56 cores (14 cores per socket) and 190GB memory. Our evaluation of all applications uses GCC 8.1 and SEAL v3.3.1 [149], which implements the RNS variant of the CKKS scheme [41]. All experiments use the default 128-bit security level.

We evaluate a simple arithmetic application to compute the path length in 3-dimensional space. We also evaluate applications in statistical machine learning, image processing, and deep neural network (DNN) inferencing using the frontends that we built on top of EVA (Section 3.7). For DNN inferencing, we compare EVA with the prior state-of-the-art compiler for homomorphic

DNN inferencing, CHET [56], which has been shown to outperform hand-tuned codes. For the other applications, no suitable compiler exists for comparison. Hand-written codes also do no exist as it is very tedious to write them manually. We evaluate these applications using EVA to show that EVA yields good performance with little programming effort. For DNN inferencing, the accuracy reported is for all test inputs, whereas all the other results reported are an average over the first 20 test inputs. For the other applications, all results reported are an average over 20 different randomly generated inputs.

3.8.2 Deep Neural Network (DNN) Inference

Networks: We evaluate five deep neural network (DNN) architectures for image classification tasks that are summarized in Table 3.4:

- The three **LeNet-5** networks are all for the MNIST [113] dataset, which vary in the number of neurons. The largest one matches the one used in the TensorFlow's tutorials [154].
- **Industrial** is a network from an industry partner for privacy-sensitive binary classification of images.
- **SqueezeNet-CIFAR** is a network for the CIFAR-10 dataset [105] that uses 4 Fire-modules [47] and follows the SqueezeNet [91] architecture.

These networks are the same ones evaluated in CHET [56] (Section 2.7); i.e., LeNet-5 and SqueezeNet-CIFAR networks were made FHE-compatible

Table 3.4: Deep Neural Networks used in our evaluation.

Network	No. of layers			# FP	Accu-
	Conv	FC	Act	operations	racy(%)
LeNet-5-small	2	2	4	159960	98.45
LeNet-5-medium	2	2	4	5791168	99.11
LeNet-5-large	2	2	4	21385674	99.30
Industrial	5	2	6	-	-
SqueezeNet-CIFAR	10	0	9	37759754	79.38

Table 3.5: Programmer-specified input and output scaling factors used for both CHET and EVA, and the accuracy of homomorphic inference in CHET and EVA (all test inputs).

Model	Input Scale ($\log P$)			Output	Accuracy(%)	
	Cipher	Vector	Scalar	Scale	CHET	EVA
LeNet-5-small	25	15	10	30	98.42	98.45
LeNet-5-medium	25	15	10	30	99.07	99.09
LeNet-5-large	25	20	10	25	99.34	99.32
Industrial	30	15	10	30	-	-
SqueezeNet-CIFAR	25	15	10	30	79.31	79.34

using average-pooling and polynomial activations instead of max-pooling and ReLU activations. Table 3.4 lists the accuracies we observed for these networks using unencrypted inference on the test datasets. We evaluate encrypted image inference with a batch size of 1 (latency).

Scaling Factors: The scaling factors, or scales in short, must be chosen by the user. For each network (and model), we use CHET's profiling-guided optimization on the first 20 test images to choose the input scales as well as the desired output scale. There is only one output but there are many inputs.

Table 3.6: Average latency (s) of CHET and EVA on 56 threads.

Model	**CHET**	**EVA**	**Speedup from EVA**
LeNet-5-small	3.7	0.6	6.2
LeNet-5-medium	5.8	1.2	4.8
LeNet-5-large	23.3	5.6	4.2
Median	70.4	9.6	7.3
SqueezeNet-CIFAR	344.7	72.7	4.7

For the inputs, we choose one scale each for **Cipher**, **Vector**, and **Scalar** inputs. Both CHET and EVA use the same scales, as shown in Table 3.5. The scales impact both performance and accuracy. We evaluate CHET and EVA on all test images using these scales and report the accuracy achieved by fully-homomorphic inference in Table 3.5. There is negligible difference between their accuracy and the accuracy of unencrypted inference (Table 3.4). Higher values of scaling factors may improve the accuracy, but will also increase the latency of homomorphic inference.

Comparison with CHET Compiler: Table 3.6 shows that EVA is at least 4× faster than CHET on 56 threads for all networks. Note that the average latency of CHET is slower than that reported in Section 2.7 ([56]). This could be due to differences in the experimental setup. The input and output scales used in Section 2.7 are different, so is the SEAL version (3.1 vs. 3.3.1). We suspect the machine differences to be the primary reason for the slowdown because smaller number of heavier cores (16 3.2GHz cores vs. 56 2.2GHz cores) are used in Section 2.7. In any case, our comparison of CHET and

EVA is fair because both use the same input and output scales, SEAL version, Channel-Height-Width (CHW) data layout, and hardware. Both CHET and EVA perform similar encryption parameters and rotation keys selection. The differences between CHET and EVA are solely due to the benefits that accrue from EVA's low-level optimizations.

CHET relies on an expert-optimized library of homomorphic tensor kernels, where each kernel (1) includes FHE-specific instructions and (2) is explicitly parallelized. However, even experts cannot optimize or parallelize across different kernels as that information is not available to them. In contrast, EVA uses a library of vectorized tensor kernels and automatically (1) inserts FHE-specific instructions using global analysis and (2) parallelizes the execution of different instructions across kernels. Due to these optimizations, *EVA is on average* 5.3× *faster than CHET*. On a single thread (Figure 3.7), EVA is on average 2.3× faster than CHET and this is solely due to better placement of FHE-specific instructions. The rest of the improvement on 56 threads (2.3× on average) is due to better parallelization in EVA.

Both CHET and EVA have similar Relinearize placement. However, they differ in the placement of the other FHE-specific instructions — Rescale and ModSwitch. These instructions directly impact the encryption parameters (both CHET and EVA use a similar encryption parameter selection pass). We report the encryption parameters selected by CHET and EVA in Table 3.7. EVA selects much smaller coefficient modulus, both in terms of the number of elements r in it and their product Q. Consequently, the polynomial modulus

Table 3.7: Encryption parameters selected by CHET and EVA (where $Q = \prod_{i=1}^{r} Q_i$).

Model	**CHET**			**EVA**		
	$\log_2 N$	$\log_2 Q$	r	$\log_2 N$	$\log_2 Q$	r
LeNet-5-small	15	480	8	14	360	6
LeNet-5-medium	15	480	8	14	360	6
LeNet-5-large	15	740	13	15	480	8
Industrial	16	1222	21	15	810	14
SqueezeNet-CIFAR	16	1740	29	16	1225	21

N is one power-of-2 lower in all networks, except LeNet-5-large. Reducing N and r reduces the cost (and the memory) of each homomorphic operation (and ciphertext) significantly. In CHET, Rescale and ModSwitch used by the experts for a given tensor kernel may be sub-optimal for the program. On the other hand, EVA performs global (inter-procedural) analysis to minimize the length of the coefficient modulus, yielding much smaller encryption parameters.

To understand the differences in parallelization, we evaluated CHET and EVA on 1, 7, 14, 28, and 56 threads. Figure 3.7 shows the strong scaling. We omit LeNet-5-small because it takes too little time, even on 1 thread. It is apparent that EVA scales much better than CHET. The parallelization in CHET is within a tensor operation or kernel using OpenMP. Such static, *bulk-synchronous* schedule limits the available parallelism. In contrast, EVA dynamically schedules the directed acyclic graph of EVA (or SEAL) operations asynchronously. Thus, it exploits the parallelism available across tensor kernels, resulting in much better scaling. The average speedup of EVA on 56

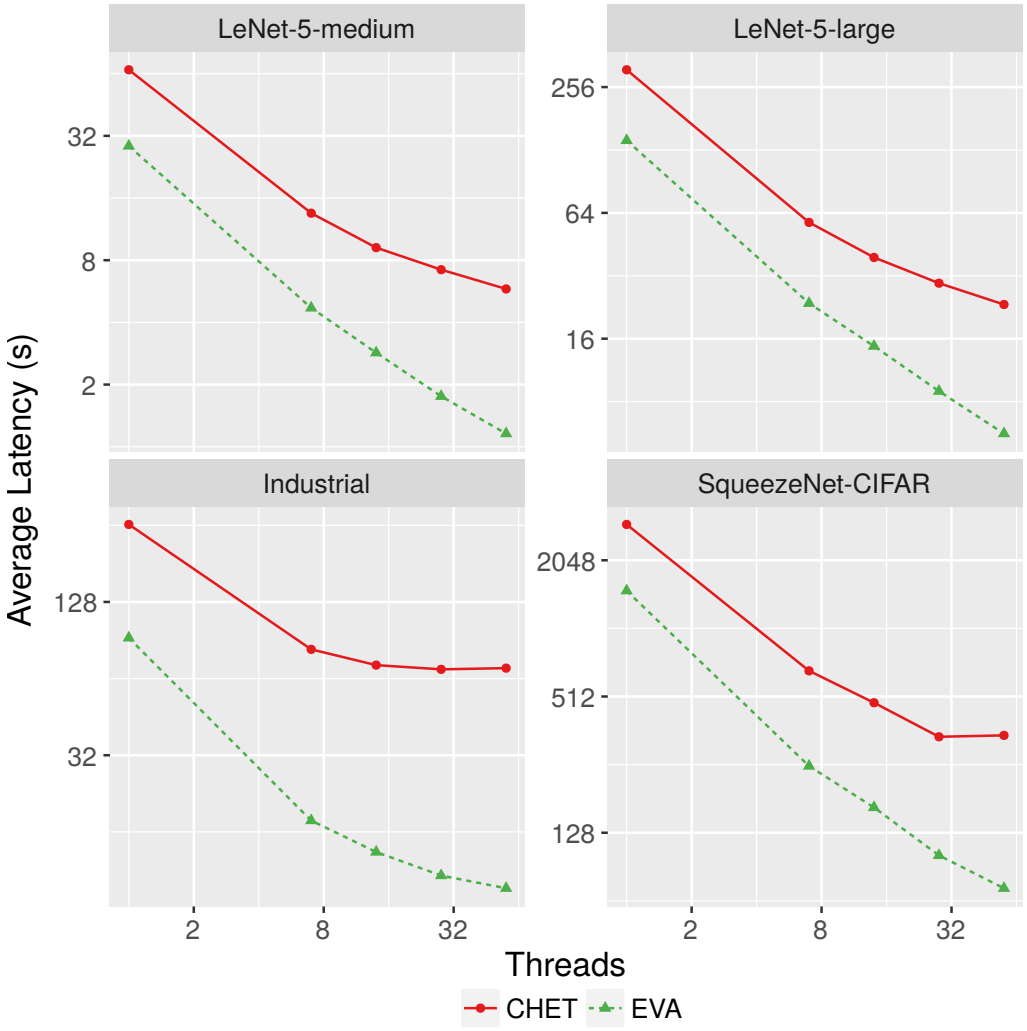


Figure 3.7: Strong scaling of CHET and EVA (log-log scale).

threads over EVA on 1 thread is 18.6× (excluding LeNet-5-small).

Comparison with Hand-Written LoLa: LoLa [27] implements hand-tuned homomorphic inference for neural networks, but the networks they implement are different than the ones we evaluated (and the ones in CHET). Nonetheless, they implement networks for the MNIST and CIFAR-10 datasets.

For the MNIST dataset, LoLa implements the highly-tuned CryptoNets [67] network (which is similar in size to LeNet-5-small). This implementation has an average latency of 2.2 seconds and has an accuracy of 98.95%. EVA takes only 1.2 seconds on a much larger network, LeNet-5-medium, with a better accuracy of 99.09%. For the CIFAR-10 dataset, LoLa implements a custom network which takes 730 seconds and has an accuracy of 74.1%. EVA takes only 72.7 seconds on a much larger network with a better accuracy of 79.34%.

LoLa uses SEAL 2.3 (which implements BFV [59]) which is less efficient than SEAL 3.3.1 (which implements RNS-CKKS [41]) but much more easier to use. EVA is faster because it exploits a more efficient FHE scheme which is much more difficult to manually write code for. Thus, *EVA outperforms even highly tuned expert-written implementations like LoLa with very little programming effort.*

Compilation Time: We present the compilation time, encryption context time, encryption time, and decryption time for all networks in Table 3.8. The encryption context time includes the time to generate the public key, the

Table 3.8: Compilation, encryption context (context), encryption, and decryption time for EVA.

Model	Time (s)			
	Compilation	Context	Encrypt	Decrypt
LeNet-5-small	0.14	1.21	0.03	0.01
LeNet-5-medium	0.50	1.26	0.03	0.01
LeNet-5-large	1.13	7.24	0.08	0.02
Industrial	0.59	15.70	0.12	0.03
SqueezeNet-CIFAR	4.06	160.82	0.42	0.26

Table 3.9: Evaluation of EVA for fully-homomorphic arithmetic, statistical machine learning, and image processing applications on 1 thread (LoC: lines of code).

Application	Vector Size	LoC	Time (s)
3-dimensional Path Length	4096	45	0.394
Linear Regression	2048	10	0.027
Polynomial Regression	4096	15	0.104
Multivariate Regression	2048	15	0.094
Sobel Filter Detection	4096	35	0.511
Harris Corner Detection	4096	40	1.004

secret key, the rotation keys, and the relinearization keys. This can take a lot of time, especially for large N, like in SqueezeNet-CIFAR. Compilation time, encryption time, and decryption time are negligible for all networks.

3.8.3 Arithmetic, Statistical Machine Learning, and Image Processing

We implemented several applications using PyEVA. To illustrate a simple arithmetic application, we implemented an application that computes the length of a given encrypted 3-dimensional path. This computation can be used

as a kernel in several applications like in secure fitness tracking on mobiles. For statistical machine learning, we implemented linear regression, polynomial regression, and multi-variate regression on encrypted vectors. For image processing, we implemented Sobel filter detection and Harris corner detection on encrypted images. All these implementations took very few lines of code (< 50), as shown in Table 3.9.

Table 3.9 shows the execution time of these applications on encrypted data using 1 thread. Sobel filter detection takes half a second and Harris corner detection takes only a second. The rest take negligible time. We believe *Harris corner detection is one of the most complex programs that have been evaluated using CKKS*. EVA enables writing advanced applications in various domains with little programming effort, while providing excellent performance.

3.9 Related Work

Libraries for FHE SEAL [149] implements RNS variants of two FHE schemes: BFV [59] and CKKS [41, 42]. HElib [81] implements two FHE schemes: BGV [26] and CKKS. PALISADE [138] is a framework that provides a general API for multiple FHE schemes including BFV, BGV, and CKKS. For BFV and CKKS, PALISADE is similar to SEAL as it only implements lower-level FHE primitives. On the other hand, EVA language abstracts batching-compatible FHE schemes like BFV, BGV, and CKKS while hiding cryptographic details from the programmer. Although EVA compiler currently generates code targeting only CKKS implementation in SEAL, it can

be adapted to target other batching-compatible FHE scheme implementations or FHE libraries.

General-Purpose Compilers for FHE To reduce the burden of writing FHE programs, general-purpose compilers have been proposed that target different FHE libraries. These compilers share many of the same goals as ours. Some of these compilers support general-purpose languages like Julia (cf. [10]), C++ (cf. [45]), and R (cf. [11]), whereas ALCHEMY [48] is the only one that provides its own general-purpose language. Unlike EVA, none of these languages are amenable to be a target for domain-specific compilers like CHET [56] because these languages do not support rotations on fixed power-of-two sized vectors. Nonetheless, techniques in these compilers (such as ALCHEMY's static type safety and error rate analysis) are orthogonal to our contributions in this thesis and can be incorporated in EVA.

All prior general-purpose compilers target (libraries implementing) either the BFV scheme [59] or the BGV scheme [26]. In contrast, EVA targets (libraries implementing) the recent CKKS scheme [41, 42], which is much more difficult to write or generate code for (compared to BFV or BGV). For example, ALCHEMY supports the BGV scheme and would require significant changes to capture the semantics (e.g., Rescale) of CKKS. ALCHEMY always inserts ModSwitch after every ciphertext-ciphertext multiplication (using local analysis), which is not optimal for BGV (or BFV) and would not be correct for CKKS. EVA is the first general-purpose compiler for CKKS and

it uses a graph rewriting framework to insert RESCALE and MODSWITCH operations correctly (using global analysis) so that the modulus chain length is optimal. These compiler passes in EVA can be incorporated in other general-purpose compilers (to target CKKS).

Domain-Specific Compilers for FHE Some prior compilers for DNN inferencing [16, 17, 56] target CKKS. CHET [56] is a compiler for tensor programs that automates the selection of *data layouts* for mapping tensors to vectors of vectors. The nGraph-HE [17] project introduced an extension to the Intel nGraph [50] deep learning compiler that allowed data scientists to make use of FHE with minimal code changes. The nGraph-HE compiler uses runtime optimization (e.g., detection of special plaintext values) and compile-time optimizations (e.g., use of ISA-level parallelism, graph-level optimizations). nGraph-HE2 [16] is an extension of nGraph-HE that uses a hybrid computational model – the server interacts with the client to perform non-HE compatible operations, which increases the communication overhead. Moreover, unlike CHET and EVA, neither nGraph-HE nor nGraph-HE2 automatically select encryption parameters.

To hide the complexities of FHE operations, all existing domain-specific compilers rely on a runtime of high-level kernels which can be optimized by experts. However, experts are limited to information within a single kernel (like convolution) to optimize insertion of FHE-specific operations and to parallelize execution. In contrast, EVA optimizes insertion of FHE-specific operations by

using global analysis and parallelizes FHE operations across kernels transparently. Therefore, CHET, nGraph-HE, and nGraph-HE2 can target EVA instead of the FHE scheme directly to benefit from such optimizations and we demonstrated this for CHET.

Compilers for MPC Multi-party computation (MPC) [73, 170] is another technique for privacy-preserving computation. The existing MPC compilers [80] are mostly general-purpose and even though it is possible to use them for deep learning applications, it is hard to program against a general-purpose interface. The EzPC compiler [35] is a machine learning compiler that combines arithmetic sharing and garbled circuits and operates in a two-party setting. EzPC uses ABY [128] as a cryptographic backend.

Privacy-Preserving Deep Learning CryptoNets [67], one of the first systems for neural network inference using FHE and the consequent work on LoLa [27], a low-latency CryptoNets, show the ever more practical use of FHE for deep learning. CryptoNets and LoLa however use kernels for neural networks that directly translate the operations to the cryptographic primitives of the FHE schemes. There are also other algorithms and cryptosystems specifically for deep learning that rely on FHE (CryptoDL [83], Chabanne et al. [33], Jiang et al. [96]), MPC (Chameleon [144], DeepSecure [145], SecureML [129]), oblivious protocols (MiniONN [117]), or on hybrid approaches (Gazelle [98], SecureNN [163]). None of these provide the flexibility and the optimizations of a compiler approach.

Chapter 4

Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics

This thesis introduces a new approach to building distributed-memory graph analytics systems that exploits heterogeneity in processor types (CPU and GPU), partitioning policies, and programming models. The key to this approach is Gluon[1], a communication-optimizing substrate.

Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run on heterogeneous clusters and optimizes communication in a novel way by exploiting structural and temporal invariants of graph partitioning policies.

To demonstrate Gluon's ability to support different programming mod-

[1]My main contributions to this work include the design of the Gluon substrate, the implementation of the Gluon substrate for GPUs, and the design and implementation of partition-aware communication optimizations. The full citation of the published version of this work is: "Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pages 752–768, New York, NY, USA, 2018. ACM".

els, we interfaced Gluon with the Galois and Ligra shared-memory graph analytics systems to produce distributed-memory versions of these systems named D-Galois and D-Ligra, respectively. To demonstrate Gluon's ability to support heterogeneous processors, we interfaced Gluon with IrGL, a state-of-the-art single-GPU system for graph analytics, to produce D-IrGL, the first multi-GPU distributed-memory graph analytics system.

Our experiments were done on CPU clusters with up to 256 hosts and roughly 70,000 threads and on multi-GPU clusters with up to 64 GPUs. The communication optimizations in Gluon improve end-to-end application execution time by $\sim 2.6\times$ on the average. D-Galois and D-IrGL scale well and are faster than Gemini, the state-of-the-art distributed CPU graph analytics system, by factors of $\sim 3.9\times$ and $\sim 4.9\times$, respectively, on the average.

4.1 Introduction

Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges, or the indexable Web graph, which has roughly 100 billion nodes and trillions of edges. Parallel computing is essential for processing graphs of this size in reasonable time. McSherry *et al.* [123] have shown that shared-memory graph analytics systems like Galois [134] and Ligra [150] process medium-scale graphs efficiently, but these systems cannot be used for graphs with billions of nodes and edges because main memory is limited to a few terabytes even on large servers.

One solution is to use clusters, which can provide petabytes of storage for in-memory processing of large graphs. Graphs are partitioned between the machines, and communication between partitions is implemented using a substrate like MPI. Existing systems in this space such as PowerGraph [74], PowerLyra [38], and Gemini [173] have taken one of two approaches regarding communication. Some systems build optimized communication libraries for a single graph partitioning strategy; for example, Gemini supports only edge-cut partitioning. However, the best-performing graph partitioning strategy depends on the algorithm, input graph, and number of hosts, so this approach is not sufficiently flexible. An alternative approach taken in systems like PowerGraph and PowerLyra is to support vertex-cut graph partitioning. Although vertex-cuts are general, communication is implemented using the generic gather-apply-scatter model [74], which does not take advantage of partitioning invariants to optimize communication. Since performance on large clusters is limited by communication overhead, a key challenge is to optimize communication while supporting heterogeneous partitioning policies. This is explained further in Section 4.2 using a simple example.

Another limitation of existing systems is that they are integrated solutions that come with their own programming models, runtime systems, and communication runtimes, which makes it difficult to reuse infrastructure to build new systems. For example, all existing GPU graph analytics systems such as Gunrock [139, 165], Groute [13], and IrGL [137] are limited to a single node, and there is no way to reuse infrastructure from existing distributed

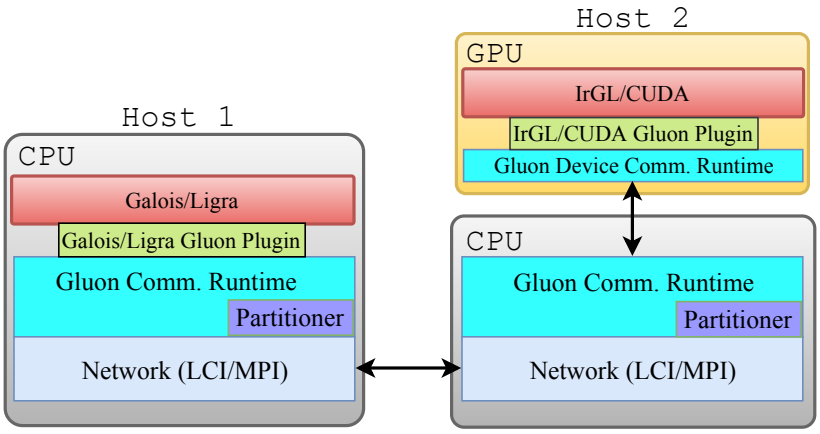


Figure 4.1: Overview of the Gluon Communication Substrate.

graph analytics systems to build GPU-based distributed graph analytics systems from these single-node systems.

This thesis introduces a new approach to building distributed-memory graph analytics systems that exploits heterogeneity in programming models, partitioning policies, and processor types (CPU and GPU). The key to this approach is Gluon, a communication-optimizing substrate.

Programmers write applications in a shared-memory programming system of their choice and interface these applications with Gluon using a lightweight API. Gluon enables these programs to run efficiently on heterogeneous clusters by partitioning the input graph using a policy that can be chosen at runtime and by optimizing communication for that policy.

To demonstrate Gluon's support for *heterogeneous programming models*, we integrated Gluon with the Galois [134] and Ligra [150] shared-memory

systems to build distributed graph analytics systems that we call *D-Galois* and *D-Ligra*, respectively. To demonstrate Gluon's support for *processor heterogeneity*, we integrated Gluon with IrGL [137], a single-GPU graph analytics system, to build *D-IrGL*, the first cluster-based multi-GPU graph analytics system.

Figure 4.1 illustrates a distributed, heterogeneous graph analytics system that can be constructed with Gluon: there is a CPU host running Galois or Ligra, and a GPU, connected to another CPU, running IrGL.

Another contribution of Gluon is that it incorporates novel communication optimizations that *exploit structural and temporal invariants of graph partitions to optimize communication*.

1. *Exploiting structural invariants*: We show how general graph partitioning strategies can be supported in distributed-memory graph analytics systems while still exploiting structural invariants of a given graph partitioning policy to optimize communication (Section 4.3).

2. *Exploiting temporal invariance*: The partitioning of the graph does not change during the computation. We show how this *temporal invariance* can be exploited to reduce both the overhead and the volume of communication compared to existing systems (Section 4.4).

Our evaluation in Section 4.5 shows that the CPU-only systems D-Ligra and D-Galois are faster than Gemini [173], the state-of-the-art distributed-memory CPU-only graph analytics system, on CPU clusters with up to 256

hosts and roughly 70,000 threads. The geomean speedup of D-Galois over Gemini is $\sim 3.9\times$ even though D-Galois, unlike Gemini, is not a monolithic system. We also show that D-IrGL is effective as a multi-node, multi-GPU graph analytics system on multi-GPU clusters with up to 64 GPUs, yielding a geomean speedup of $\sim 4.9\times$ over Gemini. Finally, we demonstrate that our communication optimizations result in a $\sim 2.6\times$ geomean improvement in running time compared to a baseline in which these optimizations are turned off.

The rest of this chapter is organized as follows. We present an overview of Gluon in Section 4.2. Sections 4.3 and 4.4 describe Gluon's communication optimizations that exploit structural and temporal invariants in partitioning policies respectively. We present our experimental results on distributed clusters of CPUs and GPUs in Section 4.5. Section 4.6 discusses related work.

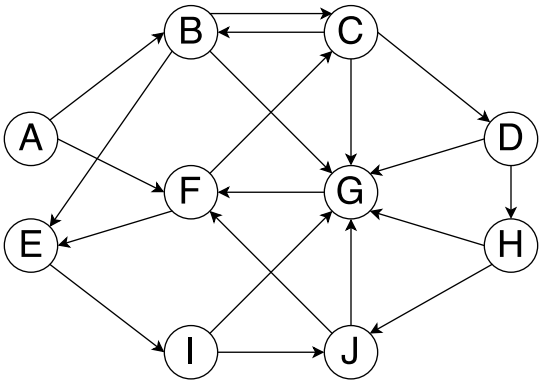
4.2 Overview of Gluon

Gluon can be interfaced with any shared-memory graph analytics system that supports the vertex programming model, which is described briefly in Section 4.2.1. Section 4.2.2 gives a high-level overview of how Gluon enables such systems to run on distributed-memory machines. Section 4.2.3 describes opportunities for optimizing communication compared to the baseline gather-apply-scatter model [74] of synchronization.

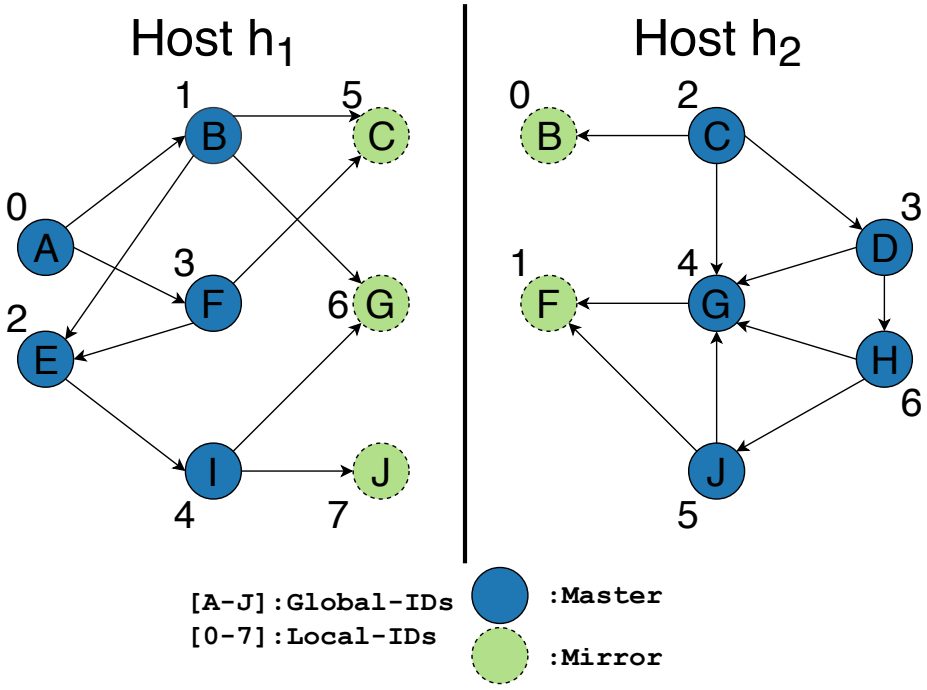
4.2.1 Vertex Programs

In the graph analytics applications considered in this thesis, each node v has a label $l(v)$ that is initialized at the beginning of the algorithm and updated during the execution of the algorithm until a global quiescence condition is reached. In some problems, edges also have labels, and the label of an edge (v, w) is denoted by $weight(v, w)$. We use the single-source shortest-path (sssp) problem to illustrate concepts. Vertex programs for this problem initialize the label of the source node to zero and the label of every other node to a large positive number. At the end of the computation, the label of each node is the distance of the shortest path to that node from the source node.

Updates to node labels are performed by applying a computation rule called an *operator* [141] to nodes in the graph. A *push-style* operator uses the label of a node to conditionally update labels of its immediate neighbors, while a *pull-style* operator reads the labels of the immediate neighbors and conditionally updates the label of the node where the operator is applied. For the sssp problem, the operator is known as the relaxation operator. A push-style relaxation operator sets the label of a neighbor w of the node v to the value $l(v) + weight(v, w)$ if $l(w)$ is larger than this value [46]. Shared-memory systems like Galois repeatedly apply operators to graph nodes until global quiescence is reached.



(a) Original graph



(b) Partitioned graph

Figure 4.2: An example of partitioning a graph for two hosts.

4.2.2 Distributed-Memory Execution

Distributed-memory graph analytics is more complicated since it is necessary to perform both computation and communication. The graph is partitioned between hosts at the start of the computation. Execution is done in rounds: in each round, a host applies the operator to graph nodes in its own partition and then participates in a global communication phase in which it exchanges information about labels of nodes at partition boundaries with other hosts. Since fine-grain communication is very expensive on current systems, execution models with coarse-grain communication, such as bulk-synchronous parallel (BSP) execution, are preferred [159].

To understand the issues that arise in coupling shared-memory systems on different (possibly heterogeneous) hosts to create a distributed-memory system for graph analytics, consider Figure 4.2(a), which shows a directed graph with ten nodes labeled `A` through `H` (the *global-IDs* of nodes). There are two hosts $\mathtt{h}_1$ and $\mathtt{h}_2$, and the graph is partitioned between them. Figure 4.2(b) shows the result of applying an *Outgoing Edge-Cut* (OEC) partitioning (described in Section 4.3.1): nodes {`A,B,E,F,I`} have been assigned to host $\mathtt{h}_1$ and the other nodes have been assigned to host $\mathtt{h}_2$. Each host creates a *proxy node* for the nodes assigned to it, and that proxy node is said to be a *master*: it keeps the canonical value of the node.

Some edges, such as edge `(B,G)` in Figure 4.2(a), connect nodes assigned to different hosts. For these edges, OEC partitioning creates a *mirror node* for the destination node (in the example, node `G`) on the host that owns the

source node (in the example, host $\texttt{h}_1$), and it creates an edge on $\texttt{h}_1$ between the proxy nodes for `B` and `G`. The following invariants hold in the partitioned graph of Figure 4.2(b).

a) Every node `N` in the input graph is assigned to one host $\texttt{h}_i$. Proxy nodes for `N` are created on this host and possibly other hosts. The proxy node on $\texttt{h}_i$ is called the master proxy for `N`, and the others are designated mirror proxies.

b) In the partitioned graph, all edges connect proxy nodes on the same host.

Consider a push-style sssp computation. Because of invariant (b), the application program running on each host is oblivious to the existence of other partitions and hosts, and it can execute its sssp code on its partition independently of other hosts using any shared-memory graph analytics system (*this is the key insight that allows us to interface such systems with a common communication-optimizing substrate*). Node `G` in the graph of Figure 4.2 has proxies on both hosts, and both proxies have incoming edges, so the labels on both proxies may be updated and read independently by the application programs running on the two hosts. Since mirror nodes implement a form of software-controlled caching, it is necessary to reconcile the labels on proxies at some point. In BSP-style synchronization, collective communication is performed among all hosts to reconcile the labels on all proxies. At the end of synchronization, computation resumes at each host, which is again agnostic of other partitions and hosts.

In the sssp example, the label of the mirror node for `G` on host $\mathtt{h}_1$ can be transmitted to h_2, and the label of the master node for `G` on h_2 can be updated to the minimum of the two labels. In general, a node may have several mirror proxies on different hosts. If so, the values on the mirrors can be communicated to the master, which reduces them and updates its label to the resulting value. This value can then be broadcast to the mirror nodes, which update their labels to this value. This general approach is called *gather-apply-scatter* in the literature [74].

4.2.3 Opportunities for Communication Optimization

Communication is the performance bottleneck in graph analytics applications [169], so communication optimizations are essential to improving performance.

The first set of optimizations exploit structural invariants of partitions to reduce the amount of communication compared to the default gather-apply-scatter implementation. In Figure 4.2(b), we see that mirror nodes do not have outgoing edges; this is an invariant of the OEC partitioning (explained in Section 4.3). This means that a push-style operator applied to a mirror node is a no-op and that the label on the mirror node is never read during the computation phase. Therefore, the volume of communication can be reduced in half by just resetting the labels of mirror nodes locally instead of updating them to the master's value during the communication phase, which obviates the need to communicate values from masters to the mirrors. The value to

reset the mirror nodes to depends on the operator. For example, for sssp, keeping labels of mirror nodes unchanged is sufficient whereas for push-style pagerank, the labels are reset to 0. In Section 4.3, we describe a number of important partitioning strategies, and we show how Gluon can be used to exploit their structural invariants to optimize communication.

The second set of optimizations, described in Section 4.4, reduces the memory and communication overhead by exploiting the temporal invariance of graph partitions. Once the graph has been partitioned, each host stores its portion of the graph using a representation of its choosing. Proxies assigned to a given host are given *local-IDs*, and the graph structure is usually stored in Compressed-Sparse-Row (CSR) format, which permits the proxies assigned to a given host to be stored contiguously in memory regardless of their global-IDs. Figure 4.2(b) shows an assignment of local-IDs (numbers in the figure) to the proxies.

Since local-IDs are used only in intra-host computation and have no meaning outside that host, communication between a master and its mirrors on different hosts requires reference to their common global-ID. In current systems, each host maintains a map between local-IDs and global-IDs. To communicate the label of a proxy on host h_1 to the corresponding proxy on host h_2, (i) the local-ID of the proxy on h_1 is translated to the global-ID, (ii) the global-ID and node label are communicated to host h_2, and (iii) the global-ID is translated to the corresponding local-ID on h_2. This approach has two overheads: conversion between global-IDs and local-IDs and communication

of global-IDs with labels.

Gluon implements an important optimization called *memoization of address translation* (Section 4.4.1), which obviates the need for the translation and for sending global-IDs. A second optimization (Section 4.4.2) tracks updates to proxies and avoids sending proxy values that have not changed since the previous round. While this optimization is implemented in other systems such as PowerGraph and Gemini, these systems send global-IDs along with proxy values. Implementing this optimization in a system like Gluon is more challenging for two reasons: it is not an integrated computation/communication solution, and it does not send global-IDs with proxy values. Section 4.4.2 describes how we address this problem.

4.3 Exploiting Structural Invariants to Optimize Communication

Section 4.3.1 describes partitioning strategies implemented in Gluon. Section 4.3.2 describes how communication can be optimized by using the structural invariants of these strategies. Section 4.3.3 describes the Gluon API calls that permit these optimized communication patterns to be plugged in with existing shared-memory graph analytical systems.

4.3.1 Partitioning Strategies

The partitioning strategies implemented by Gluon can be presented in a unified way as follows. The edges of the graph are distributed between

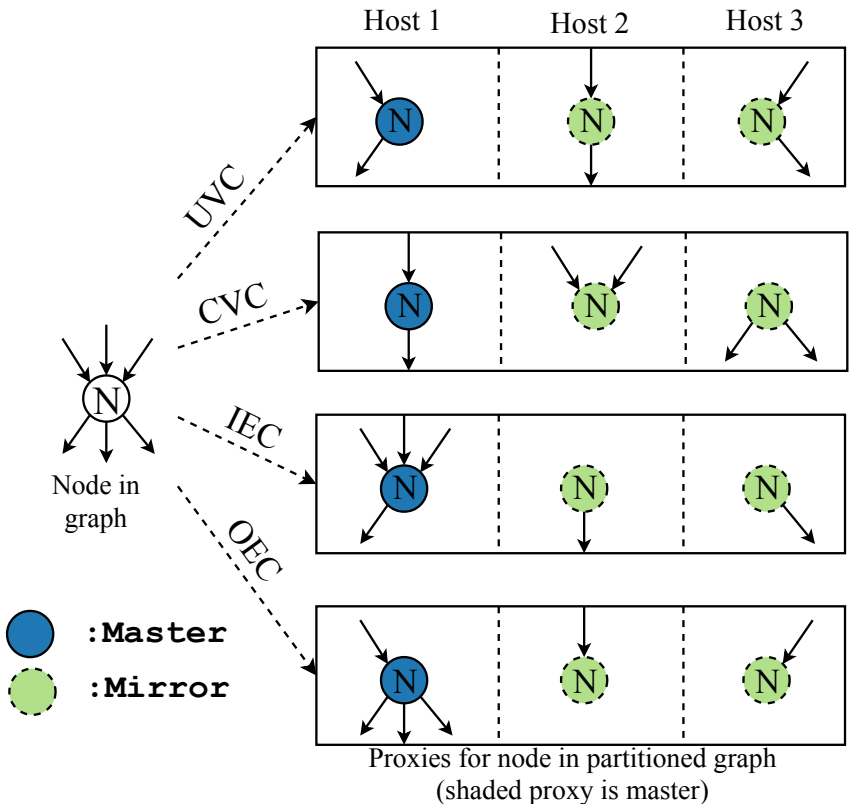


Figure 4.3: Different partitioning strategies.

hosts using some policy. If a host is assigned an edge (N_1, N_2), proxies are created for nodes N_1 and N_2 and are connected by an edge on that host. If the edges connected to a given node end up on several hosts, that node has several proxies in the partitioned graph. One proxy is designated the master for all proxies of that node, and the others are designated as mirrors. The master is responsible for holding and communicating the canonical value of the node during synchronization.

Partitioning policies can interact with the structure of the operator in the sense that some partitioning policies may not be legal for certain operators. To understand this interaction, it is useful to classify various policies (or heuristics) for graph partitioning into four categories or strategies. These are described below and illustrated in Figure 4.3. In this figure, a node N in the graph has three proxies on different hosts; the shaded proxy is the master.

- *Unconstrained Vertex-Cut* (*UVC*) can assign both the outgoing and incoming edges of a node to different hosts.
- *Cartesian Vertex-Cut* (*CVC*) is a constrained vertex-cut: only the master can have both incoming and outgoing edges, while mirror proxies can have either incoming or outgoing edges but not both.
- *Incoming Edge-Cut* (*IEC*) is more constrained: incoming edges are assigned only to the master while outgoing edges of a node are partitioned among hosts.

- *Outgoing Edge-Cut* (*OEC*) is the mirror image of IEC: outgoing edges are assigned only to the master while incoming edges are partitioned among hosts.

Some of these partitioning strategies can be used only if the operator has a special structure. For a pull-style operator, UVC, CVC, or OEC can be used only if the update made by the operator to the active node label is a reduction; otherwise, IEC must be used since all the incoming edges required for local computation are present at the master. For a push-style operator, UVC, CVC, or IEC can be used only if the node pushes the same value along its outgoing edges and uses a reduction to combine the values pushed to it on its incoming edges; otherwise, OEC must be used since master nodes have all the outgoing edges required for the computation. The benchmarks studied in this thesis satisfy all these conditions, so any partitioning strategy can be used for them.

4.3.2 Partitioning Invariants and Communication

Each partitioning strategy requires a different pattern of optimized communication to synchronize proxies, but they can be composed from two basic patterns supported by a thin API. The first is a *reduce* pattern in which values at mirror nodes are communicated to the host that owns the master and combined on that host using a reduction operation. The resulting value is written to the master. The second is a *broadcast* pattern in which data at the master is broadcast to the mirrors. For some partitioning strategies, only

a subset of mirrors may be involved in the communication.

Consider push-style operators that read the label of the active node and write to the labels of outgoing neighbors or pull-style operators that read the labels of incoming neighbors and write to the label of the active node. In these cases, the data flows from the source to the destination of an edge. We discuss only the synchronization patterns for this scenario in this section; synchronization patterns for other cases are complementary and are not discussed further.

Since the data flows from the source to the destination of an edge, a proxy's label is only read if it is the source node of an edge and only written if it is the destination node of an edge (a reduction is considered to be a write). Consequently, there are two invariants during execution:

(1) If a proxy node has no outgoing edges, its label will not be read during the computation phase.

(2) If a proxy node has no incoming edges, its label will not be written during the computation phase.

These invariants can be used to determine the communication patterns required for the different partitioning strategies.

- *UVC*: Since the master and mirrors of a node can have outgoing as well as incoming edges, any proxy can be written during the computation phase. At the end of the round, the labels of the mirrors are communicated to

the master and combined to produce a final value. The value is written to the master and broadcast to the mirror nodes. Therefore, both reduce and broadcast are required. This is the default gather-apply-scatter pattern used in systems like PowerGraph [74].

- *CVC*: Only the master can have both incoming and outgoing edges; mirrors can have either incoming or outgoing edges but not both. Consequently, mirrors either read from the label or write to the label but not both. At the end of the round, the set of mirrors that have only incoming edges communicate their values to the master to produce the final value. The master then broadcasts the value to the set of mirrors that have only outgoing edges. Like UVC, CVC requires both reduce and broadcast synchronization patterns, but each pattern uses only a particular subset of mirror nodes rather than all mirror nodes. This leads to better performance at scale for many programs.

- *IEC*: Only the master has incoming edges, so only its label can be updated during the computation. The master communicates this updated value to the mirrors for the next round. Therefore, only the broadcast synchronization pattern is required. This is the *halo exchange* pattern used in distributed-memory finite-difference codes.

- *OEC*: Only the master of a node has outgoing edges. At the end of the round, the values pushed to the mirrors are combined to produce the final result on the master, and the values on the mirrors can be reset

```
struct SSSP { /* SSSP edgemap functor */
  unsigned int* dist;
  SSSP(unsigned int* _dist) : dist(_dist) { }
  bool update(unsigned s, unsigned d, int edgeLen) {
    unsigned int new_dist = dist[s] + edgeLen;
    if (dist[d] > new_dist) {
      dist[d] = new_dist;
      return true;
    }
    return false;
  }
  ... /* other Ligra functor functions */
};
void Compute(...) { /* Main computation loop */
  ... /* setup initial local-frontier */
  do {
    edgeMap(LigraGraph, LocalFrontier, SSSP(dist), ...);
    gluon::sync<WriteAtDestination, ReadAtSource, ReduceDist
    , BroadcastDist>(LocalFrontier);
    /* update local-frontier for next iteration */
  } while (LocalFrontier is non-zero on any host);
}
```

Figure 4.4: Ligra plugged in with Gluon: sssp (D-Ligra).

to the (generalized) zero of the reduction operation for the next round. Therefore, only the reduce synchronization pattern is required.

4.3.3 Synchronization API

To interface programs with Gluon, a (blocking) synchronization call is inserted between successive parallel rounds in the program (*e.g.*, after edgeMap() in Ligra, do_all() in Galois, Kernel in IrGL). Figure 4.4 shows how Gluon can be used in an sssp application in Ligra to implement communication-optimized

```
struct ReduceDist { /* Reduce struct */
  static unsigned extract(unsigned localNodeID) {
    return dist[localNodeID];
  }
  static bool reduce(unsigned localNodeID, unsigned y)
  {
    if (y < dist[localNodeID]) { /* min operation */
      dist[localNodeID] = y;
      return true;
    }
    return false;
  }
  static void reset(unsigned localNodeID) {
    /* no-op */
  }
};
struct BroadcastDist { /* Broadcast struct */
  static unsigned extract(unsigned localNodeID) {
    return dist[localNodeID];
  }
  static void set(unsigned localNodeID, unsigned y) {
    dist[localNodeID] = y;
  }
};
```

Figure 4.5: Gluon synchronization structures: sssp (D-Ligra).

distributed sssp (D-Ligra). The synchronization call to the Gluon substrate shown in line 18 passes the reduce and broadcast API functions shown in Figure 4.5 to Gluon. Synchronization is field-sensitive: therefore, synchronization structures similar to the ones shown in Figure 4.5 are used for each label updated by the program. Depending on the partitioning strategy, Gluon uses reduce, broadcast, or both.

The functions in the reduce and broadcast structures specify how to access data that must be synchronized (there are also bulk-variants for GPUs). Broadcast has two main functions:

- *Extract*: Masters call this function to extract their node field values from the local graph to broadcast them to mirrors.
- *Set*: Mirrors call this function to update their node field values in the local graph to the canonical value received from masters.

Reduce has three main functions:

- *Extract*: Mirrors call this function to extract their node field values from the local graph to communicate them to the master.
- *Reduce*: Masters call this function to combine the partial values received from the mirrors to their node field values in the local graph.
- *Reset*: Mirrors call this function to reset their node field values in the local graph to the identity-value of the reduction operation for the next round.

Note that the application code does not depend on the particular partitioning strategy used by the programmer. Instead, Gluon composes the optimized communication pattern from the information in the `sync` call and its knowledge of the communication requirements of the particular partitioning strategy requested by the user at runtime. This permits users to explore a variety of partitioning strategies just by changing command-line flags, which permits auto-tuning.

The approach described in this section decouples the computation from communication, which enables the computation to run on any device or engine using any scheduling strategy. For each compute engine (Ligra, Galois, and IrGL), the broadcast and reduction structures can be supported through application-agnostic preprocessor templates. Each application only needs to identify the field(s) to synchronize, the reduction operation(s), and the point(s) at which to synchronize. This can be identified by statically analyzing the operator (*e.g.*, edgeMap() in Ligra, do_all() in Galois, Kernel in IrGL) and inserting the required `sync` call (we have implemented this for Galois and IrGL in a compiler called Abelian [68]). Communication is automatically specialized for the partitioning policy specified at runtime.

4.4 Exploiting Temporal Invariance to Optimize Communication

The optimizations in Section 4.3 exploit properties of the operator and partitioning strategy to reduce the volume of communication for synchronizing

the proxies of a *single* node in the graph. In general, there are many millions of nodes whose proxies need to be synchronized in each round. In this section, we describe two optimizations for reducing overheads when performing this bulk communication. The first optimization, described in Section 4.4.1, permits proxy values to be exchanged between hosts without sending the global-IDs of nodes. The second optimization, described in Section 4.4.2, tracks updates to proxies and avoids sending proxy values that have not changed since the previous round.

4.4.1 Memoization of Address Translation

As described in Section 4.3, synchronizing proxies requires sending values from mirrors to masters or vice versa. Mirrors and masters are on different hosts, so the communication needed for this is handled in current systems by sending node global-IDs along with values. Consider Figure 4.2(b): if host $\texttt{h}_2$ needs to send host $\texttt{h}_1$ the labels on the mirrors for nodes `B` and `F` in its local memory, it sends the global-IDs `B` and `F` along with these values. At the receiving host, these global-IDs are translated into local-IDs (in this case, `1` and `3`), and the received values are assimilated into the appropriate places in the local memory of that host.

This approach has both time and space overheads: communication volume and time increase because the global-IDs are sent along with values, and computational overhead increases because of the translation between global and local-IDs. To reduce these overheads, Gluon implements a key optimiza-

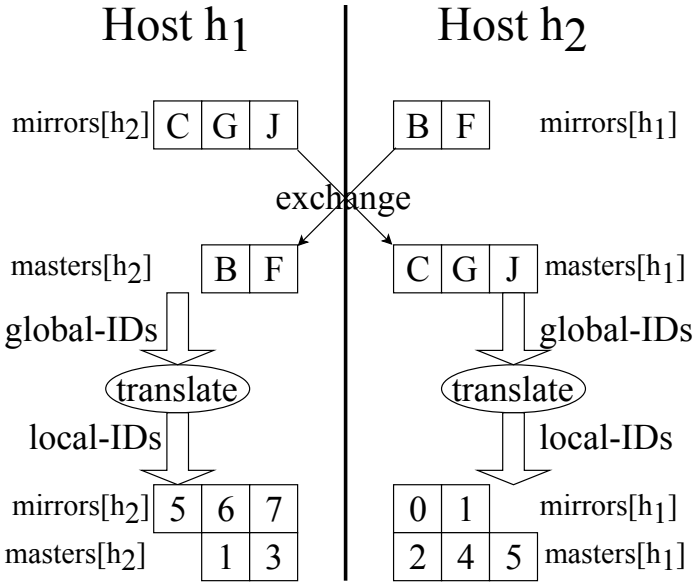


Figure 4.6: Memoization of address translation for the partitions in Figure 4.2.

tion called *memoization of address translation* that eliminates the sending of global-IDs as well as the translation between global and local-IDs.

In Gluon, as in existing distributed-memory graph analytics systems, graph partitions and the assignment of partitions to hosts do not change during the computation[2]. Gluon exploits this temporal invariance of partitions as follows. Before the computation begins, Gluon establishes an agreement between pairs of hosts $(\mathtt{h}_i, \mathtt{h}_j)$ that determines which proxies on $\mathtt{h}_i$ will send values to $\mathtt{h}_j$ and the order in which these values will be packed in the message.

This high-level idea is implemented as follows. We use Figure 4.6, which shows the memoization of address translation for the partitions in Figure 4.2,

[2]If the graph is re-partitioned, then memoization can be done soon after partitioning to amortize the communication costs until the next re-partitioning.

to explain the key ideas. In Gluon, each host reads from disk a subset of edges assigned to it and receives from other hosts the rest of the edges assigned to it. The end-points of these edges are specified using global-IDs. When building the in-memory (CSR) representation of its local portion of the graph, each host maintains a map to translate the global-IDs of its proxies to their local-IDs and a vector to translate the local-IDs to global-IDs.

After this, each host informs every other host about the global-IDs of its mirror nodes whose masters are owned by that other host. In the running example, host $\mathtt{h}_1$ informs $\mathtt{h}_2$ that it has mirrors for nodes with global-IDs {C,G,J}, whose masters are on $\mathtt{h}_2$. It does this by constructing the *mirrors* array shown in Figure 4.6 and sending it to $\mathtt{h}_2$. This exchange of mirrors provides the *masters* array shown in the figure. After the exchange, the global-IDs in the *masters* and *mirrors* arrays are translated to their local-IDs. This is the only point where the translation is performed unless the user code explicitly uses the global-ID of a node during computation (e.g., to set the source node in sssp).

During the execution of the algorithm, communication is either from masters to mirrors (during broadcast) or from mirrors to masters (during reduce). Depending on whether it is broadcast or reduce, the corresponding masters or mirrors array (respectively) of local IDs is used to determine what values to send to that host. Once the values are received, the corresponding mirrors or masters array (respectively) is used to update the proxies. Since the order of proxy values in the array has been established during memoization

and the messages sent by the runtime respect this ordering, there is no need for address translation.

In addition to reducing communication and address translation overheads, an important benefit of this optimization is that it enables Gluon to leverage existing shared-memory frameworks like Galois and Ligra out-of-the-box. Moreover, systems for other architectures like GPUs need not maintain memory-intensive address translation structures, thereby enabling Gluon to directly leverage GPU frameworks like IrGL.

4.4.2 Encoding of Metadata for Updated Values

In BSP-style execution of graph analytics algorithms, each round usually updates labels of only a small subset of graph nodes. For example, in the first round of sssp, only the neighbors of the source node have their labels updated. An important optimization is to avoid communicating the labels of nodes that have not been updated in a given round. If global-IDs are sent along with label values as is done in existing systems, this optimization is easy to implement [38, 74, 90, 173]. If, however, the memoization optimization described in Section 4.4.1 is used, it is not clear how to send only the updated values in a round since receivers will not know which nodes these values belong to.

One way to track the node labels that have changed is to keep a shadow copy of it in the communication runtime and compare the original against it to determine if the label has changed. To be more space efficient, Gluon requires

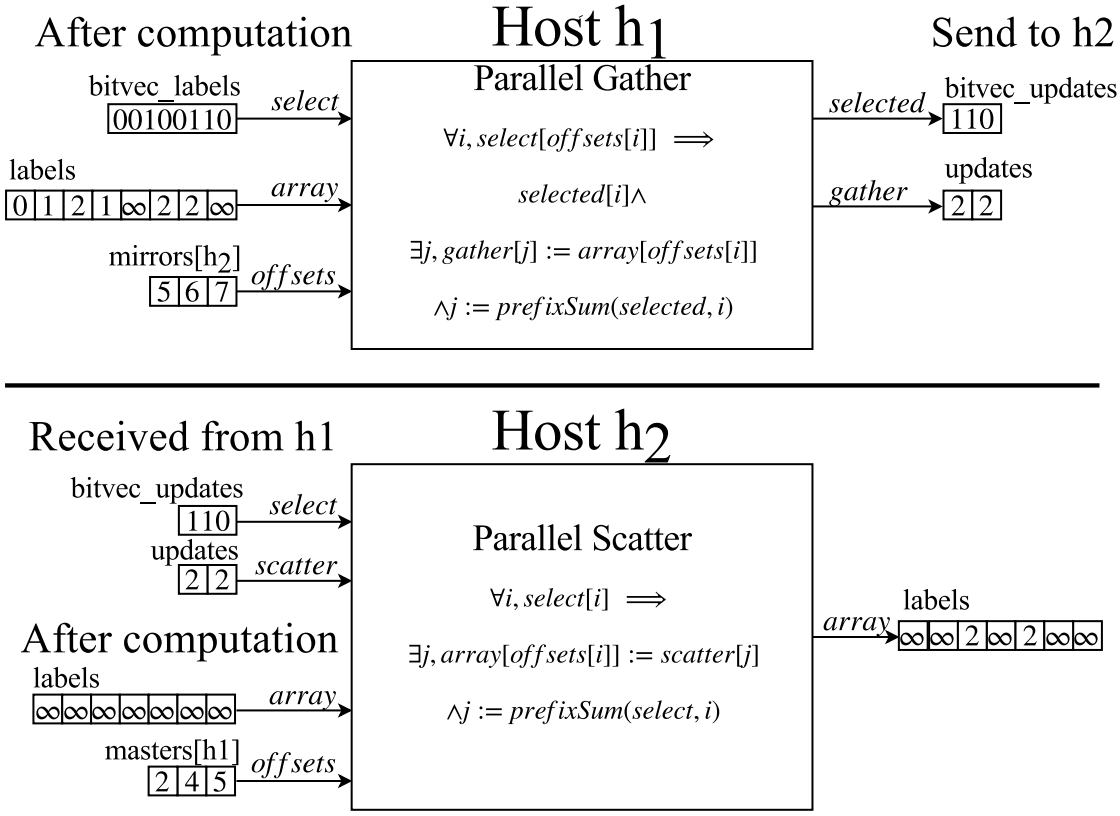


Figure 4.7: Communication from host h_1 to h_2 after second round of BFS algorithm with source A on the OEC partitions in Figure 4.2.

the shared-memory runtime to provide a field-specific bit-vector that indicates which nodes' labels have changed. This is passed to the synchronization call to Gluon (e.g., in Figure 4.4, `LocalFrontier` in line 18).

To illustrate this idea, we consider a level-by-level BFS algorithm on the partitions in Figure 4.2 with source A. Since these are OEC partitions, the mirrors need to be reduced to the master after each round (described in Section 4.3.2). In the first round, host h_1 updates the labels of B and F to

1. There is nothing to communicate since those nodes are not shared with h_2. After the second round, h_1 updates C, G, and E to 2. The updates to C and G need to be synchronized with h_2.

In the top left of Figure 4.7, host h_1 has the updated labels with its bit-vector indicating which local labels changed in the second round and the *mirrors* of h_2 from memoization (Figure 4.6). Instead of gathering all mirrors, only the mirrors that are set in the bit-vector are selected and gathered. Out of 5, 6, and 7 (C, G, and J, respectively), only 5 and 6 are set in the bit-vector. This yields the *updates* array of 2 and 2 shown in the top right. In addition, a bit-vector is determined which indicates which of the mirrors were selected. The *bitvec_updates* shows that mirror 0 and 1 (0-indexed) were selected from the mirrors. This bit-vector is sent along with the updates to h_2.

In the bottom left of Figure 4.7, host h_2 has the labels after the second round of computation and the *masters* of h_1 from memoization (Figure 4.6). It also has the bit-vector and the updates received from h_1. The bit-vector is used to select the masters, and the updates are scattered to the labels of those masters. Since 0 and 1 indices are set in the bit-vector, local-IDs 2 and 4 are selected, and their labels are updated with 2 and 2 respectively, yielding the updated labels shown on the bottom right.

When updates are very dense, sending all the labels of the mirrors or masters can obviate the need for the bit-vector metadata since most labels will be updated. To illustrate for the current running example, an array of $[2, 2, \infty]$ would be sent to h_2, and h_2, using its *masters* array, can correctly

scatter them to the C, G, and J. In cases when the updates are very sparse, sending just indices of the selected mirrors or masters instead of the bit-vector can further reduce the size of the metadata. If indices are sent for the example considered, an indices array of 0 and 1 is sent instead of the bit-vector 110. The indices are then used by the receiver to select the masters.

Gluon has the different modes described above to encode the metadata compactly. Simple rules to select the mode are as follows:

- When the updates are dense, do not send any bit-vector metadata, but send labels of all mirrors or masters.
- When the updates are sparse, send bit-vector metadata along with updated values.
- When the updates are very sparse, send the indices along with the updated values.
- When there are no updates, send an empty message.

The number of bits set in the bit-vector is used to determine which mode yields the smallest message size. A byte in the sent message indicates which mode was selected. Neither the rules nor the modes are exhaustive. Other compression or encoding techniques could be used to represent the bit-vector as long as they are deterministic (and sufficiently parallel).

4.4.3 Implementation

As shown in Figure 4.1, the Gluon communication runtime runs on each host and device. Each GPU is associated with a host CPU. The runtime on the host CPU manages the communication between the host and the device as well as the communication with other hosts. The device communication runtime prepares the messages to be sent to other hosts and processes the messages received from other hosts. Data is gathered and scattered on the device. The parallel gather and scatter operations for encoding metadata shown in Figure 4.7 are implemented using `cub::DeviceSelect` primitives. The host CPU only acts as a forwarding interface (or router) for the device. Similarly, the Gluon runtime on the CPU gathers and scatters data in parallel.

Inter-host communication is handled by a common network runtime, as shown in the Figure 4.1. MPI is the default communication runtime, but Gluon can be interfaced and used with other communication runtimes like Lightweight Communication Interface (LCI) [51], which operates directly on the underlying network hardware and provides light-weight interaction with threads.

4.5 Experimental Evaluation

Gluon can use either MPI or LCI [51] for message transport between hosts, as shown in Figure 4.1. We use LCI in our evaluation[3]. To evaluate

[3]Dang et al. [51] show the benefits of LCI over MPI for graph analytics.

Gluon, we interfaced it with the Ligra [150], Galois [134], and IrGL [137] shared-memory graph analytics engines to build three distributed-memory graph analytics systems that we call D-Ligra, D-Galois[4], and D-IrGL respectively. We compared the performance of these systems with that of Gunrock [139], the prior state-of-the-art single-node multi-GPU graph analytics system, and Gemini [173], the prior state-of-the-art distributed-memory CPU-only graph analytics system (the Gemini thesis shows that their system performs significantly better than other systems in this space such as PowerLyra [38], PowerGraph [74], and GraphX [168]).

We describe the benchmarks and experimental platforms (Section 4.5.1), graph partitioning times (Section 4.5.2), the performance of all systems at scale (Section 4.5.3), experimental studies of the CPU-only systems (Sections 4.5.4), and experimental studies of the GPU-only systems (Section 4.5.5). Section 4.5.6 gives a detailed breakdown of the performance impact of Gluon's communication optimizations.

4.5.1 Experimental Setup

All CPU experiments were conducted on the Stampede KNL Cluster (Stampede2) [153] at the Texas Advanced Computing Center [5], a distributed cluster connected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). Each node has a KNL processor with 68 1.4 GHz cores running four hardware threads per core, 96GB of DDR4 RAM, and 16GB of MC-DRAM.

[4]The Abelian system used in [51] is another name for D-Galois.

We used up to 256 CPU hosts. Since each host has 272 hardware threads, this allowed us to use up to 69632 threads. All code was compiled using g++ 7.1.0.

GPU experiments were done on the Bridges [135] supercomputer at the Pittsburgh Supercomputing Center [4, 157], another distributed cluster connected through Intel Omni-Path Architecture. Each machine has 2 Intel Haswell CPUs with 14 cores each and 128 GB DDR4 RAM, and each is connected to 2 NVIDIA Tesla K80 dual-GPUs (4 GPUs in total with) with 24 GB of DDR5 memory (12 GB per GPU). Each GPU host uses 7 cores and 1 GPU (4 GPUs share a single physical node). We used up to 64 GPUs. All code was compiled using g++ 6.3.0 and CUDA 9.0.

Our evaluation uses programs for breadth-first search (bfs), connected components (cc), pagerank (pr), and single-source shortest path (sssp). In D-Galois and D-IrGL, we implemented a pull-style algorithm for pagerank and push-style data-driven algorithms for the rest. Both push-style and pull-style implementations are available in D-Ligra due to Ligra's direction optimization. The source nodes for bfs and sssp are the maximum out-degree node. The tolerance for pr is 10^{-9} for rmat26 and 10^{-6} for the other inputs. *We run pr for up to 100 iterations; all the other benchmarks are run until convergence.* Results presented are for a mean of 3 runs.

Table 4.1 shows the properties of the six input graphs we used in our studies. Three of them are real-world web-crawls: the web data commons hyperlink graph [126, 127], wdc12, is the largest publicly available dataset;

Table 4.1: Inputs and their key properties.

	rmat26	twitter40	rmat28	kron30	clueweb12	wdc12
$\|V\|$	67M	41.6M	268M	1073M	978M	3,563M
$\|E\|$	1,074M	1,468M	4,295M	10,791M	42,574M	128,736M
$\|E\|/\|V\|$	16	35	16	16	44	36
max D_{out}	238M	2.99M	4M	3.2M	7,447	55,931
max D_{in}	18,211	0.77M	0.3M	3.2M	75M	95M

clueweb12 [18, 20, 142] is another large publicly available dataset; twitter40 [107] is a smaller dataset. rmat26, rmat28, and kron30 are randomized synthetically generated scale-free graphs using the rmat [34] and kron [115] generators (we used weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]).

4.5.2 Graph Partitioning Policies and Times

We implemented the four high-level partitioning strategies described in Section 4.3.1, using particular policies to assign edges to hosts for each one. For OEC and IEC, we implemented a chunk-based edge-cut [173] that partitions the node into contiguous blocks while trying to balance outgoing and incoming edges respectively. For CVC, we implemented a 2D graph partitioning policy [21]. For UVC, we implemented a hybrid vertex-cut (HVC) [38]. While Gluon supports a variety of partitioning policies, evaluating different partitioning policies in Gluon is not the focus of this work (see [70] and [94] for comparison of graph partitioning policies on distributed CPUs and GPUs respectively), so we experimentally determined good partitioning policies for the Gluon-based systems. For bfs, pr, and sssp using D-IrGL on clueweb12, we used the OEC policy. In all other cases for D-Ligra, D-Galois, and D-IrGL,

Table 4.2: Graph construction time (sec): time to load the graph from disk, partition it, and construct in-memory representation.

1 host	rmat26	twitter40	rmat28
Ligra	271.6	158.3	396.9
Galois	**64.9**	**51.8**	**123.9**
Gemini	854.3	893.5	3084.7

32 hosts	rmat28	kron30	clueweb12
D-Ligra	70.6	257.4	728.6
D-Galois	**68.6**	**244.4**	**600.8**
Gemini	328.0	1217.4	1539.0

256 hosts	rmat28	kron30	clueweb12	wdc12
D-Ligra	69.4	235.8	470.5	1515.9
D-Galois	**65.5**	**225.7**	**396.2**	**1345.0**
Gemini	231.0	921.8	1247.7	X

unless otherwise noted, we used the CVC policy since it performs well at scale. Note that Gemini and Gunrock support only outgoing edge-cuts.

Table 4.2 shows the time to load the graph from disk, partition it (which uses MPI), and construct the in-memory representation on 32 hosts and 256 hosts for D-Ligra, D-Galois, and Gemini. For comparison, we also present the time to load and construct small graphs on a single host for Ligra, Galois, and Gemini. Partitioning the graph on more hosts may increase inter-host communication, but it also increases the total disk bandwidth available, so the total graph construction time on 256 hosts for rmat28 is *better* than the total graph construction time on 1 host for all systems. Similarly, the graph construction time on 256 hosts is better than that on 32 hosts. D-Ligra and

Table 4.3: Fastest execution time (sec) of all systems using the best-performing number of hosts: distributed CPUs on Stampede and distributed GPUs on Bridges (number of hosts in parenthesis; "-" means out-of-memory; "X" means crash).

Bench-mark	Input	CPUs			GPUs
		D-Ligra	D-Galois	Gemini	D-IrGL
bfs	rmat28	1.0 (128)	**0.8 (256)**	2.3 (8)	**0.5 (64)**
	kron30	1.6 (256)	**1.4 (256)**	5.0 (8)	**1.2 (64)**
	clueweb12	65.3 (256)	**16.7 (256)**	44.4 (16)	**10.8 (64)**
	wdc12	1995.3 (64)	**380.8 (256)**	X	-
cc	rmat28	1.4 (256)	**1.3 (256)**	6.6 (8)	**1.1 (64)**
	kron30	2.7 (256)	**2.5 (256)**	14.6 (16)	**2.5 (64)**
	clueweb12	52.3 (256)	**8.1 (256)**	30.2 (16)	**23.8 (64)**
	wdc12	176.6 (256)	**75.3 (256)**	X	-
pr	rmat28	**19.7 (256)**	24.0 (256)	108.4 (8)	**21.6 (64)**
	kron30	**74.2 (256)**	102.4 (256)	190.8 (16)	**70.9 (64)**
	clueweb12	821.1 (256)	**67.0 (256)**	257.9 (32)	**215.1 (64)**
	wdc12	663.1 (256)	**158.2 (256)**	X	-
sssp	rmat28	2.1 (256)	**1.4 (256)**	6.3 (4)	**1.1 (64)**
	kron30	3.1 (256)	**2.4 (256)**	13.9 (8)	**2.3 (64)**
	clueweb12	112.5 (256)	**28.8 (128)**	128.3 (32)	**15.8 (64)**
	wdc12	2985.9 (256)	**574.9 (256)**	X	-

D-Galois use the Gluon partitioner but construct different in-memory representations, so there is a difference in their times. Both systems are faster than Gemini, which uses the simpler edge-cut partitioning policy. On 128 and 256 hosts, the replication factor (average number of proxies per vertex) in Gemini's partitions for different inputs vary from ~ 4 to ~ 25 while the replication factor in Gluon's partitions (CVC) is smaller and varies from ~ 2 to ~ 8. This is experimental evidence that Gluon keeps the replication factor relatively low compared to Gemini.

4.5.3 Best Performing Versions

Table 4.3 shows the execution times of all systems considering their best performance using any configuration or number of hosts across the platforms on all graphs, except the small graphs rmat26 and twitter40 (Gunrock runs out of memory for all other graphs). The configuration or number of hosts that yielded the best execution time is included in parenthesis (it means that the system did not scale beyond that). D-Galois clearly outperforms Gemini, and it can run large graphs like wdc12, which Gemini cannot. D-Ligra does not perform well on very large graphs like clueweb12 and wdc12. For the other graphs, D-Ligra performs better than Gemini. For graphs that fit in 64 GPUs, D-IrGL not only outperforms Gemini but also is generally comparable or better than D-Galois using up to 256 CPUs. We attribute this to the compute power of GPUs and their high memory bandwidth. Comparing the best-performing configurations on different systems, we see that D-Ligra, D-

Table 4.4: Execution time (sec) on a single node of Stampede ("-" means out-of-memory).

Input	twitter40				rmat28			
Benchmark	**bfs**	**cc**	**pr**	**sssp**	**bfs**	**cc**	**pr**	**sssp**
Ligra	**0.31**	2.75	175.67	2.60	**0.77**	17.56	542.51	-
D-Ligra	0.44	3.16	188.70	2.92	1.21	18.30	597.30	-
Galois	0.68	2.73	**43.47**	5.55	2.54	13.20	**116.50**	21.42
D-Galois	1.03	**1.04**	86.53	**1.84**	4.05	**7.02**	326.88	**5.47**
Gemini	0.85	3.96	80.23	3.78	3.44	20.34	351.65	41.77

Galois, and D-IrGL give a geomean speedup of $\sim 2\times$, $\sim 3.9\times$, and $\sim 4.9\times$ over Gemini, respectively. These results show that the flexibility and support for heterogeneity in Gluon do not come at the expense of performance compared to the state-of-the-art.

4.5.4 Strong Scaling of Distributed CPU Systems

We first consider the performance of different systems on a *single* host using twitter40 and rmat28, which fit in the memory of one host. The goal of this study is to understand the overheads introduced by Gluon on a single host compared to shared-memory systems like Ligra (March 19, 2017 commit) and Galois (v4.0).

Table 4.4 shows the single-host results. Note that we used the implementations in the Lonestar [3] benchmark suite (v4.0) for Galois, which may not be vertex programs and may use asynchronous execution. D-Galois uses bulk-synchronous vertex programs as do Ligra and other existing distributed graph analytical systems. Ligra and Gemini use a direction-optimizing al-

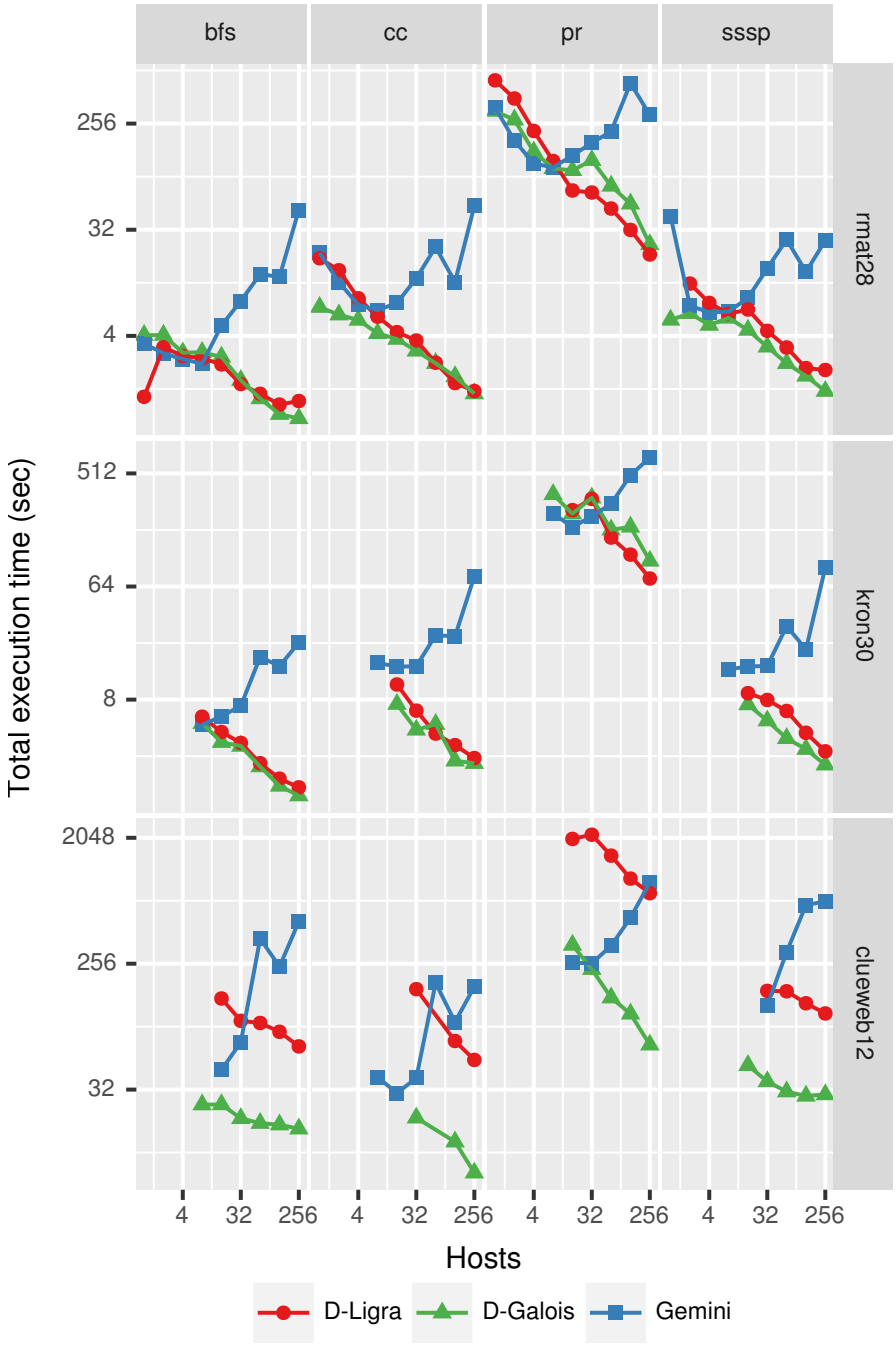


Figure 4.8: Strong scaling (log-log scale) of D-Ligra, D-Galois, and Gemini on the Stampede supercomputer (each host is a 68-core KNL node).

gorithm for bfs, so they outperform both Galois and D-Galois for bfs. The algorithms in Lonestar and D-Galois have different trade-offs; e.g., Lonestar cc uses pointer-jumping which is optimized for high-diameter graphs while D-Galois uses label-propagation which is better for low-diameter graphs. The main takeaway from this table is that D-Galois and D-Ligra are competitive with Lonestar-Galois and Ligra respectively for power-law graphs on one host for all benchmarks, which illustrates that the overheads introduced by the Gluon layer are small.

Figure 4.8 compares the strong scaling of D-Ligra, D-Galois, and Gemini up to 256 hosts[5]. The first high-level point is that for almost all applications, input graphs, and numbers of hosts, D-Galois performs better than Gemini. The second high-level point is that Gemini generally does not scale beyond 16 hosts for any benchmark and input combination. In contrast, D-Ligra and D-Galois scale well up to 256 hosts in most cases. D-Galois on 128 hosts yields a geomean speedup of $\sim 1.7\times$ over itself on 32 hosts.

For the most part, D-Ligra and D-Galois perform similarly for rmat28 and kron30 on all applications and number of hosts. However, their performance for very large graphs are significantly different. D-Ligra performs level-by-level bfs, cc, and sssp in which updates to labels of vertices in the current round are only visible in the next round; in contrast, D-Galois propagates such updates in the same round within the same host (like chaotic relaxation

[5]rmat26 and twitter40 are too small; wdc12 is too large to fit on fewer hosts. Missing point indicates graph loading or partitioning ran out-of-memory.

Figure 4.9: Communication volume (log-log scale) of D-Ligra, D-Galois, and Gemini on the Stampede supercomputer (each host is a 68-core KNL node).

in sssp). Consequently, for these algorithms, D-Ligra has $2-4\times$ more rounds and synchronization (implicit) barriers, resulting in much more communication overhead. *These results suggest that for large-scale graph analytics, hybrid solutions that use round-based algorithms across hosts and asynchronous algorithms within hosts might be the best choice.*

Since the execution time of distributed-memory graph analytics applications is dominated by communication time, we measured the bytes sent from one host to another to understand the performance differences between the systems. Figure 4.9 shows the total volume of communication. The main takeaway here is that D-Ligra and D-Galois, which are both based on Gluon, communicate similar volumes of data. Since D-Galois updates vertices in the same round (in an asynchronous manner), it sends more data than D-Ligra for bfs on clueweb12. Both D-Ligra and D-Galois send an order of magnitude less data than Gemini due to the communication optimizations in Gluon and the more flexible partitioning strategies supported by this system. The only exception is pr on clueweb12. In this case, D-Galois outperforms Gemini because CVC sends fewer messages and has fewer communication partners than the edge-cut on Gemini even though the communication volume is greater. Thus, D-Galois is an order of magnitude faster than Gemini on 128 or more hosts.

To analyze load imbalance, we measure the computation time of each round on each host, calculate the maximum and mean across hosts of each round, and sum them over rounds to determine the maximum computation time and mean computation time, respectively. The value of maximum-by-

Table 4.5: Execution time (sec) on a single node of Bridges with 4 K80 GPUs ("-" means out-of-memory).

Input	**rmat26**				**twitter40**			
Benchmark	**bfs**	**cc**	**pr**	**sssp**	**bfs**	**cc**	**pr**	**sssp**
Gunrock	-	1.81	51.46	1.42	0.88	1.46	37.37	2.26
D-IrGL(OEC)	3.61	5.72	55.72	4.13	1.03	1.57	62.81	1.99
D-IrGL(IEC)	**0.72**	7.88	**7.65**	**0.84**	**0.73**	1.55	**35.03**	**1.44**
D-IrGL(HVC)	0.82	**1.53**	8.54	0.95	1.08	1.58	44.35	2.04
D-IrGL(CVC)	2.11	4.22	46.91	2.24	0.87	**1.39**	46.86	2.32

mean computation time yields an estimate of the overhead due to load imbalance: the higher the value, the more the load imbalance. On 128 and 256 hosts for cc and pr in D-Galois with clueweb12 and wdc12, the overhead value ranges from ~ 3 to ~ 8, indicating a heavily imbalanced load. The overhead value in D-Ligra for the same cases ranges from ~ 3 to ~ 13, indicating that it is much more imbalanced; D-Ligra is also heavily imbalanced for sssp. In all other cases on 128 and 256 hosts, both D-Galois and D-Ligra are relatively well balanced with the overhead value being less than ~ 2.5.

4.5.5 Strong Scaling of Distributed GPU System

Gluon enables us to build D-IrGL, which is the first multi-node, multi-GPU graph analytics system. We first evaluate its single-node performance by comparing it with Gunrock (March 11, 2018 commit), the prior state-of-the-art single-node multi-GPU graph analytics system, for rmat26 and twitter40 (Gunrock runs out-of-memory for rmat28 or larger graphs) on a platform with four GPUs sharing a physical node. Like other existing multi-GPU graph an-

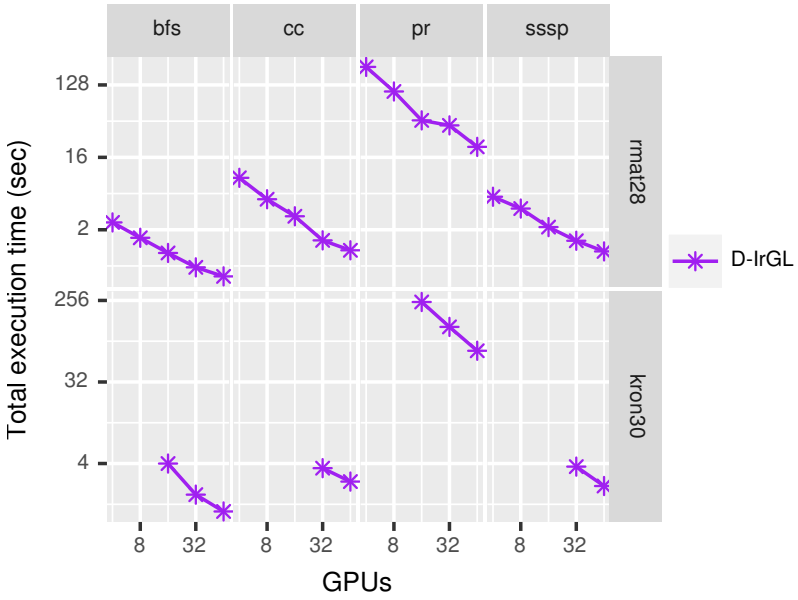


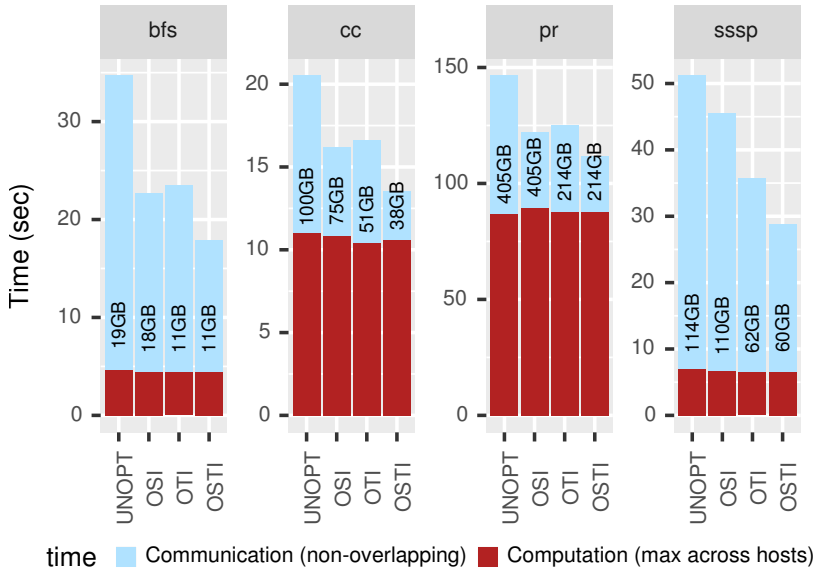
Figure 4.10: Strong scaling (log-log scale) of D-IrGL on the Bridges supercomputer (4 K80 GPUs share a physical node).

alytical systems [13, 172], Gunrock can handle only outgoing edge-cuts[6]. We evaluated D-IrGL with the partitioning policies described in Section 4.5.2. Table 4.5 shows the results. Although Gunrock performs better than D-IrGL using OEC in most cases, D-IrGL outperforms Gunrock by using more flexible partitioning policies. Considering the best-performing partitioning policy, D-IrGL gives a geomean speedup of 1.6× over Gunrock.

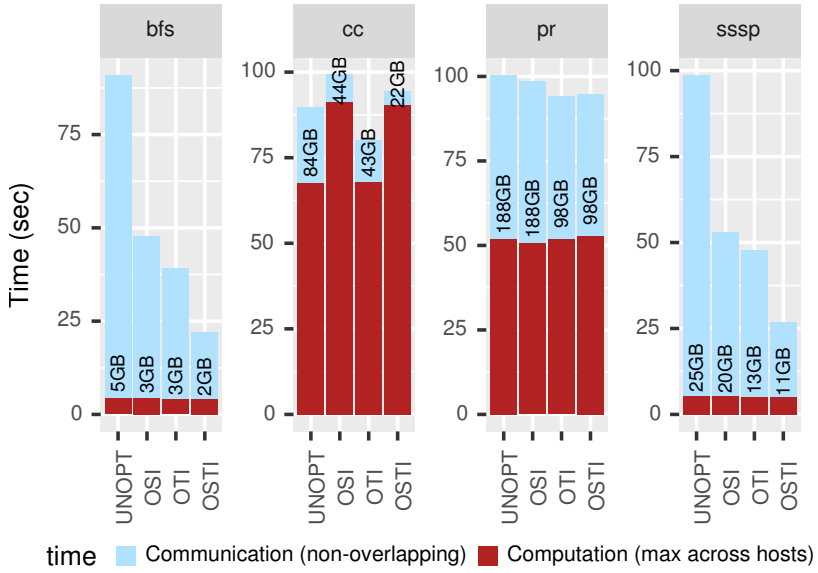
Figure 4.10 shows the strong scaling of D-IrGL[7]. For rmat28, D-IrGL on 64 GPUs yields a geomean speedup of ∼ 6.5× over that on 4 GPUs. The key

[6]We evaluated the different OEC policies in Gunrock and chose the best performing one for each benchmark and input (mostly, random edge-cut).

[7]rmat26 and twitter40 are too small while clueweb12 and wdc12 are too large to fit on fewer GPUs. Missing point indicates out-of-memory.

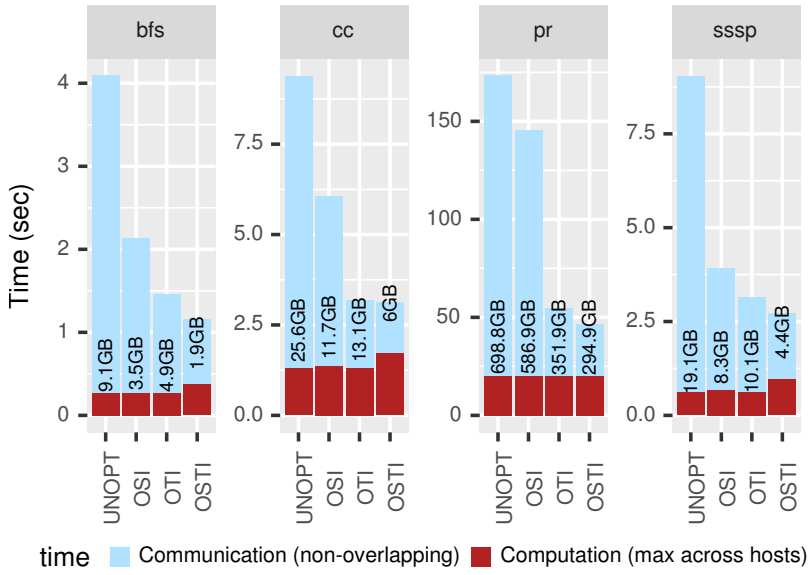


(a) clueweb12 with CVC

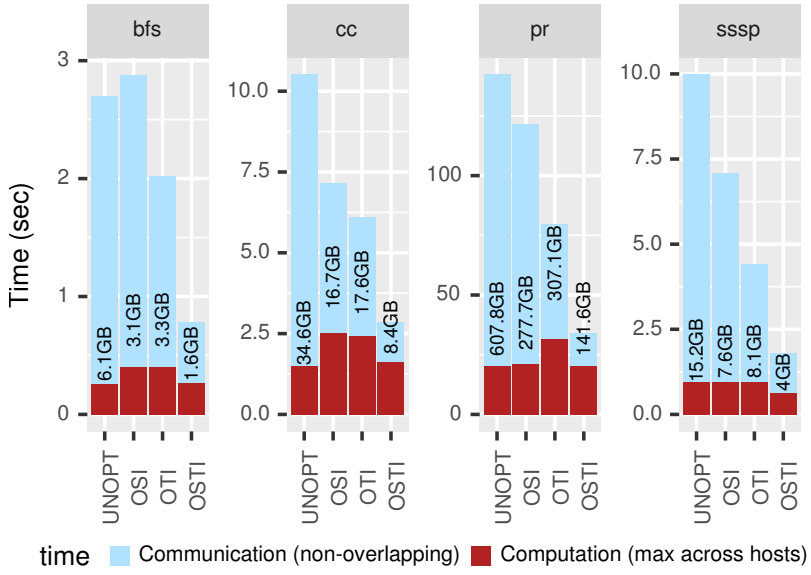


(b) clueweb12 with OEC

Figure 4.11: Gluon's communication optimizations (O) for D-Galois on 128 hosts of Stampede: SI - structural invariants, TI - temporal invariance, STI - both SI and TI.

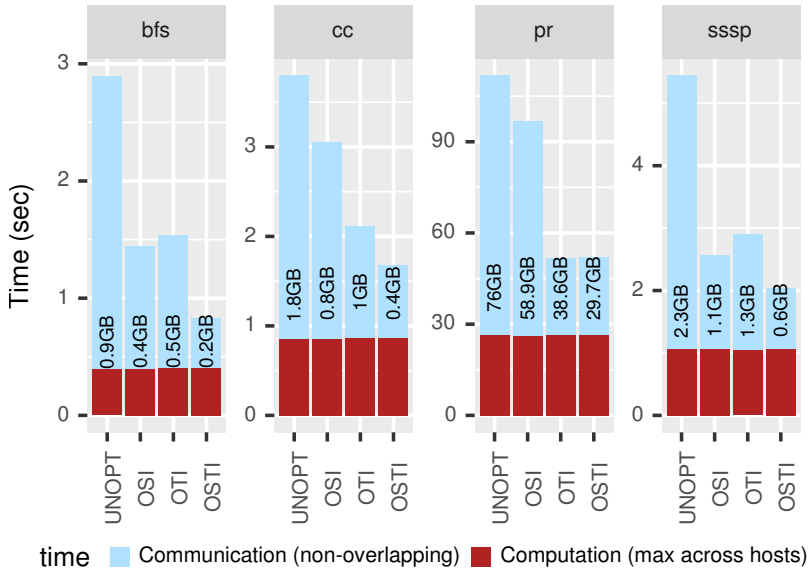


(a) rmat28 with CVC

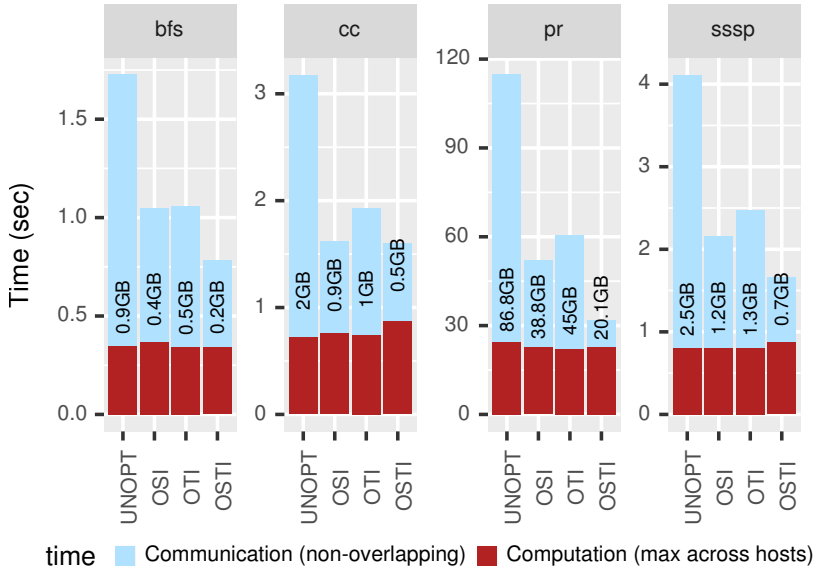


(b) rmat28 with IEC

Figure 4.12: Gluon's communication optimizations (O) for D-IrGL on 16 GPUs (4 nodes) of Bridges: SI - structural invariants, TI - temporal invariance, STI - both SI and TI.



(a) twitter40 with CVC



(b) twitter40 with IEC

Figure 4.13: Gluon's communication optimizations (O) for D-IrGL on 4 GPUs (1 node) of Bridges: SI - structural invariants, TI - temporal invariance, STI - both SI and TI.

takeaway is that Gluon permits D-IrGL to scale well like Gluon-based CPU systems.

4.5.6 Analysis of Gluon's Communication Optimizations

To understand the impact of Gluon's communication optimizations, we present a more detailed analysis of D-Galois on 128 KNL nodes of Stampede for clueweb12 using CVC and OEC, D-IrGL on 16 GPUs of Bridges for rmat28 using CVC and IEC, and D-IrGL on 4 GPUs of Bridges for twitter40 using CVC and IEC. Figures 4.11, 4.12, and 4.13 shows their execution time with several levels of communication optimization. In UNOPT, optimizations that exploit structural invariants (Section 4.3) and temporal invariance (Section 4.4) are disabled. Optimizations exploiting structural invariants and optimizations exploiting temporal invariance are enabled in OSI and OTI, respectively, while OSTI is the standard Gluon system with both turned on[8]. Each bar shows the breakdown into computation time and communication time, and within each bar, we show the communication volume. We measured the computation time of each round on each host, take the maximum across hosts in each round, and sum them over rounds, which is reported as the (maximum) computation time in Figures 4.11, 4.12, and 4.13. The rest of the execution time is reported as the (non-overlapping) communication time. As a consequence, the load imbalance would be incorporated in the computation time.

[8]In these D-IrGL experiments, we introduce cudaDeviceSynchronize(), after each computation kernel, to measure computation time precisely, so OSTI results for D-IrGL might differ slightly from that of the standard D-IrGL.

The first high-level observation is that, as expected, communication time is usually a significant portion of the overall execution time in all benchmarks even with all communication optimizations enabled. For cc and pr for clueweb12 on 128 KNL nodes, the computation time seems to be more than the communication time, but this is due to the load imbalance in those applications as explained earlier. The second high-level observation is that OTI has a significant impact on reducing communication volume. UNOPT sends 32-bit global-IDs along with the data, which is 32-bit in all cases. In OTI, memoization permits Gluon to send a bit-vector instead of global-IDs, reducing the communication volume by $\sim 2\times$. UNOPT also has the time overhead of translating to and from global-IDs during communication; this has more impact on the GPUs since this is done on the host CPU. OSI plays a significant role in reducing communication costs too. On 128 hosts using the CVC policy, UNOPT results in broadcasting updated values to at the most 22 hosts while OPT broadcasts it to at the most 7 hosts only. The overhead of these optimizations (memoization) is small: the mean runtime overhead is $\sim 4\%$ of the execution time, and the mean memory overhead is $\sim 0.5\%$. Due to these optimizations, OSTI yields a geomean speedup of $\sim 2.6\times$ over UNOPT.

4.5.7 Discussion

Systems like Gemini or Gunrock can be interfaced with Gluon to improve their communication performance. Although Gluon supports heterogeneous devices, we do not report distributed CPUs+GPUs because the 4 GPUs

on a node on Bridges outperform the CPU by a substantial margin. Nonetheless, Gluon enables plugging-in IrGL and Ligra or Galois to build distributed, heterogeneous graph analytics systems in which the device-optimized computation engine can be chosen at runtime.

4.6 Related Work

Shared-memory CPU and single-GPU graph analytics systems. Galois [79, 134, 141], Ligra [150], and Polymer [171] are state-of-the-art graph analytics frameworks for multi-core NUMA machines which have been shown to perform much better than existing distributed graph analytics systems when the graph fits in the memory of a node [123]. Gluon is designed to scale out these efficient shared-memory systems to distributed-memory clusters. As shown in Table 4.3, Gluon scales out Ligra (D-Ligra) and Galois (D-Galois) to 256 hosts. Single-GPU frameworks [77, 87, 89, 103, 137, 165] and algorithm implementations [29, 124, 130–132] have shown that the GPU can be efficiently utilized for irregular computations.

Single-node multi-GPUs and heterogeneous graph analytics systems. Several frameworks or libraries exist for graph processing on multiple GPUs [13, 58, 125, 139, 172] and multiple GPUs with a CPU [43, 65, 120]. Groute [13] is asynchronous; the rest of them use BSP-style synchronization. The most important limitation of these systems is that they are restricted to a single machine, so they cannot be used for clusters in which each machine is a multi-

GPU system. This limits the size of graphs that can be run on these systems. In addition, they only support outgoing edge-cut partitions. D-IrGL is the first system for graph analytics on clusters of multi-GPUs.

Distributed in-memory graph analytics systems. Many systems [28, 38, 72, 74, 88, 90, 93, 102, 110, 122, 133, 162, 166, 168, 169, 173] exist for distributed CPU-only graph analytics. All these systems are based on variants of the vertex programming model. Gemini [173] is the prior state-of-the-art, but it does not scale well since it does not optimize the communication volume like Gluon is able to, as seen in Figure 4.8. Moreover, computation is tightly coupled with communication in most of these systems, precluding them from using existing efficient shared-memory systems as their computation engine. Some of them like Gemini and LFGraph [90] only support edge-cut partitioning policies, but as we see in our evaluation, vertex-cut partitioning policies might be needed to scale well. Although the others handle unconstrained vertex-cuts, they do not optimize communication using structural or temporal invariants in the partitioning. Gluon's communication optimizations can be used in all these systems to build faster systems.

Out-of-core graph analytics systems. Many systems [104, 108, 121, 146, 147, 174] exist for processing graphs directly from secondary storage. GTS [104] is the only one which executes on GPUs. Chaos [146] is the only one which runs on a distributed cluster. Although they process graphs that do not fit in memory, their solution is orthogonal to ours.

Graph partitioning. Gluon makes the design space of existing partitioning policies [7, 21, 23, 32, 38, 74, 100, 101, 112, 140, 152, 158] easily available to the graph applications. Cartesian vertex-cut is a novel class of partitioning policies we identified that can reduce the communication volume and time over more general vertex-cut partitioning policies.

Chapter 5

Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics

Distributed graph analytics systems for CPUs, like D-Galois and Gemini, and for GPUs, like D-IrGL and Lux, use a bulk-synchronous parallel (BSP) programming and execution model. BSP permits bulk-communication and uses large messages which are supported efficiently by current message transport layers, but bulk-synchronization can exacerbate the performance impact of load imbalance because a round cannot be completed until every host has completed that round. Asynchronous distributed graph analytics systems circumvent this problem by permitting hosts to make progress at their own pace, but existing systems either use global locks and send small messages or send large messages but do not support general partitioning policies such as vertex-cuts. Consequently, they perform substantially worse than bulk-synchronous systems. Moreover, none of their programming or execution models can be easily adapted for heterogeneous devices like GPUs.

In this thesis, we design and implement a lock-free, non-blocking, bulk-

asynchronous runtime called Gluon-Async[1] for distributed and heterogeneous graph analytics. The runtime supports any partitioning policy and uses bulk-communication. We present the bulk-asynchronous parallel (BASP) model which allows the programmer to utilize the runtime by specifying only the abstract communication required. Applications written in this model are compared with the BSP programs written using (1) D-Galois and D-IrGL, the state-of-the-art distributed graph analytics systems (which are bulk-synchronous) for CPUs and GPUs, respectively, and (2) Lux, another (bulk-synchronous) distributed GPU graph analytical system. Our evaluation shows that programs written using BASP-style execution are on average $\sim 1.5\times$ faster than those in D-Galois and D-IrGL on real-world large-diameter graphs at scale. They are also on average $\sim 12\times$ faster than Lux. The distributed GPU graph analytics system that we built using Gluon-Async is the first graph analytics systems that supports asynchronous execution on multi-host multi-GPU platforms.

[1]My main contributions to this work include the design and implementation of the bulk-asynchronous communication runtime for distributed CPUs and GPUs as well as the design and implementation of the non-blocking termination detector. The full citation of the published version of this work is: "Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Vishwesh Jatala, V. Krishna Nandivada, Marc Snir, and Keshav Pingali. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019. IEEE".

5.1 Introduction

Present-day graph analytics systems have to handle large graphs with billions of nodes and trillions of edges [114]. Since graphs of this size may not fit in the main memory of a single machine, systems like Pregel [122], PowerGraph [74], Gemini [173], D-Galois [52], D-IrGL [52], and Lux [95] use distributed-memory clusters. In these distributed graph analytics systems, the graph is partitioned [21, 100, 151] so that each partition fits in the memory of one host in the cluster, and the *bulk-synchronous parallel* (BSP) programming model [159] is used. In this model, the program is executed in rounds, and each round consists of computation followed by communication. In the computation phase, each host updates node labels in its partition. In the communication phase, boundary node labels are reconciled so all hosts have a consistent view of labels. The algorithm terminates when a round is performed in which no label is updated on any host.

One drawback of the BSP model is that it can exacerbate the performance impact of load imbalance because a round cannot be completed until every host has completed that round. This happens frequently in graph analytics applications for two reasons: (1) unstructured power-law graphs are difficult to partition evenly, and (2) efficient graph analytics algorithms are *data-driven* algorithms that may update different subsets of nodes in each round [141], making static load balancing difficult.

One solution is to use *asynchronous* programming models and systems [60, 118, 156, 162, 164, 167], which take advantage of the fact that

many graph analytics algorithms are robust to stale reads. Here, the notion of rounds is eliminated, and a host performs computation at its own pace while an underlying messaging system ingests messages from remote hosts and incorporates boundary node label updates into the local partition of the graph. Asynchronous algorithms for particular problems like single-source shortest-path (sssp) [99] and graph coloring [61] have also been implemented. Some of these systems or implementations use global locks or send small messages, but current communication substrates in large clusters are engineered for large message sizes. The other systems send large messages but either do not handle general partitioning policies like vertex-cuts [70, 85] or do not optimize communication [52]. Consequently, the performance of these systems is not competitive with BSP systems like Gemini [173] or D-Galois [52]. In addition, it is not straightforward to extend these asynchronous programming or execution models to execute on heterogeneous devices like GPUs.

In this thesis, we explore a novel lock-free, non-blocking, asynchronous programming model that we call *bulk*-asynchronous parallel (BASP), which aims to combine the advantages of bulk communication in BSP models with the computational progress advantages of asynchronous models. BASP retains the notion of a round, but a host is not required to wait for other hosts when computation in a round is completed; instead, it sends and receives messages (if available) and moves on to the next round. One advantage of the BASP model is that it is relatively easy to modify BSP programs to BASP programs. It is also easy to modify BSP-based graph analytics systems for CPUs or GPUs

to implement this model.

In our study, we use D-Galois and D-IrGL [52], the state-of-the-art distributed CPU and GPU graph analytics systems, respectively. Both these systems are built using our prior communication-optimizing substrate, Gluon [52]. In this chapter, we refer to that substrate as Gluon-Sync. We modified Gluon-Sync to support the BASP model and we name this substrate Gluon-Async. Like Gluon-Sync, Gluon-Async can be used to extend or compile [68] existing shared-memory CPU-only or GPU-only graph analytical systems for distributed and heterogeneous execution. By using Gluon-Async instead of Gluon-Sync, we modified D-Galois and D-IrGL to develop the first asynchronous, distributed, heterogeneous graph analytics system. For large-diameter real-world web-crawls, D-IrGL using Gluon-Async is on an average $\sim 1.4\times$ faster than D-IrGL using Gluon-Sync on 64 GPUs and D-Galois using Gluon-Async is on an average $\sim 1.6\times$ faster than D-Galois using Gluon-Sync on 128 hosts. Furthermore, D-IrGL using Gluon-Async is $\sim 12\times$ faster than Lux, another BSP-style distributed GPU graph analytics system.

The rest of this chapter is organized as follows. Section 5.2 gives an overview of BSP-style distributed graph analytics and introduces the BASP model. Section 5.3 shows how Gluon [52], the state-of-the-art BSP-style distributed and heterogeneous graph analytics system, can be converted to BASP-style execution, and we believe similar modifications can be made to other BSP-based systems. Section 5.4 gives experimental results on Stampede2, a large CPU cluster, and on Bridges, a distributed multi-GPU cluster. Sec-

tion 5.5 describes related work.

5.2 Bulk-Asynchronous Parallel Model

This section introduces the BASP model. We start with an overview of the BSP model before describing BASP.

5.2.1 Overview of Bulk-Synchronous Parallel (BSP) Execution

At the start of the computation, the graph is partitioned among the hosts using one of many partitioning policies [85]. Figure 4.2 shows a graph that has been partitioned between two hosts. The edges of the graph are partitioned between hosts, and proxy nodes are created on each host for the end-points of its edges. Since the edges connected to a given vertex may be mapped to different hosts, a given vertex in the graph may have proxies on several hosts. One of these proxies is designated the master, and the others are designated as mirrors. During computation, the master holds the canonical value of the vertex, and it communicates that value to the mirrors when needed. In Figure 4.2, host h_1 has masters for nodes {A,B,E,F,I} and mirrors for nodes {C,G,J}.

Execution of the program occurs in rounds. In each round, a host computes independently on its partition of the graph. Most existing systems use the *vertex programming model* in which nodes either update the labels of their neighbors (push-style operator) or update their own labels using the labels of their neighbors (pull-style operator) until quiescence is reached. Since

a vertex in the original graph can have proxies on several hosts, the labels of these proxies may be updated differently on different hosts. For example, in a push-style breadth-first search (BFS) computation on the graph of Figure 4.2 rooted at vertex A, the mirror vertex for G on host h_1 may get the label 2 from B while the master vertex for G on host h_2 remains at the initial value ∞.

To reconcile these differences, it is necessary to perform inter-host communication. A key property of many graph analytics algorithms is that the differences among the labels of vertices can be reconciled by communicating the labels of all mirrors to the master, reducing them using an application-dependent operation, and broadcasting the result to all mirrors (as each edge is present on only one host, updates to edge labels do not involve communication). In the BFS example considered above, the value 2 will be sent to the master for vertex G on host h_2 where it is reduced with the master's label using the "minimum" operation, and the result 2 is used to update the labels of the master and mirrors. This pattern of reconciling labels using a reduction operation at the master followed by broadcast to mirrors can be used for any partitioning strategy [52]. It can also be used to offload the computation to any device [52].

In the BSP model, this reconciliation of node labels by inter-host communication is performed in each round of execution, and a host must send and ingest *all* updates from other hosts in that round before it can proceed to the next round. As a consequence, the slowest, or *straggler*, host in a

round determines when all hosts complete that round. This may increase the *idle* time of the other hosts and lead to load imbalance among hosts. This is exacerbated when the algorithm requires 100s of bulk-synchronous rounds to converge. Large real-world graph datasets have non-trivial diameter which may lead to graph analytics algorithms executing several rounds in the BSP model. This is turn may result in load imbalance among hosts, hurting performance (we analyze this in Section 5.4.4). One way to overcome this is to relax the bulk-synchronization required in each round.

5.2.2 Overview of Bulk-Asynchronous Parallel (BASP) Execution

The *bulk-asynchronous parallel* (BASP) execution model is based on the following intuition: *when a host completes its computation in a round, it can send messages to other hosts and ingest messages from other hosts, but it can go on to the next round of computation without waiting for messages from any stragglers.* Conceptually, the barrier at the end of each BSP round becomes a point at which each host sends and ingests messages without waiting for all other hosts to reach that point. The correctness of this execution strategy depends on the fact that graph analytics algorithms are resilient to stale reads: as long as there are no lost updates, execution will complete correctly.

Since hosts perform communication only at the end of a round, the BASP execution model permits the message transport layer to use large messages, which is advantageous on current systems since they do not handle small messages efficiently. In contrast, the asynchronous model in GraphLab [118]

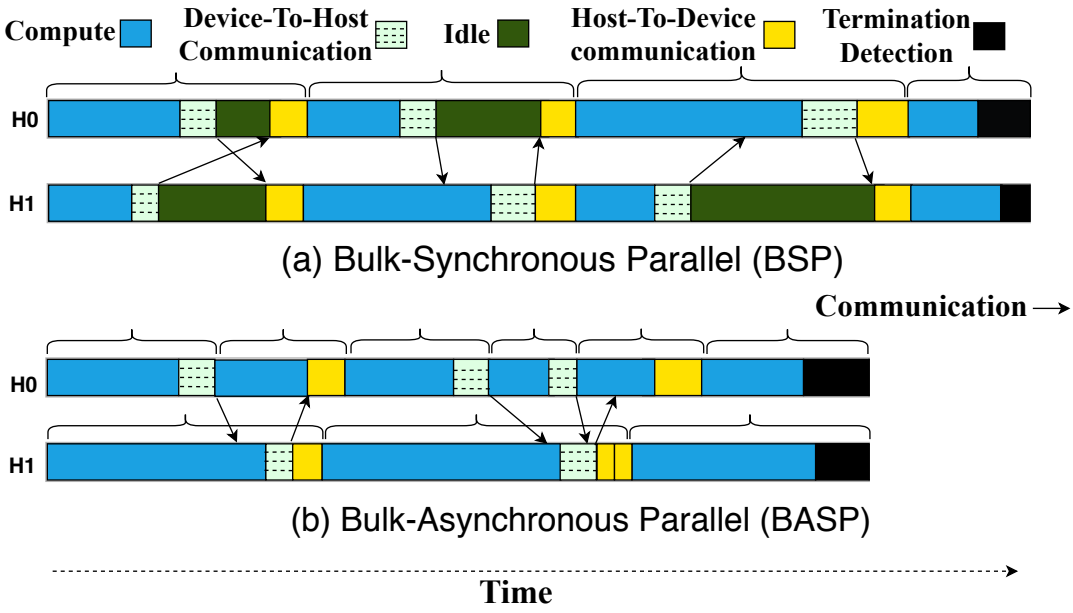


Figure 5.1: BSP vs. BASP execution.

uses small messages (along with locks) to interleave inter-host communication with computation, which is difficult to support efficiently on current systems.

Figure 5.1(a) shows a timeline for BSP-style computation on two GPUs. Each GPU is assumed to be a device that is connected to a host that performs inter-host communication. In each round, a GPU performs computation, transfers data to its host, and gets data from its host when that host receives it from the remote host. One feature of efficient graph analytics algorithms is that the amount of computation in each round in a given partition can vary unpredictably between rounds, so balancing computational load statically is difficult. This means that in each BSP round, some GPUs may be idle for long periods of time waiting for overloaded GPUs to catch up. This is shown in the

second BSP round in Figure 5.1(a): device H1 has more computation to do than device H0 in some rounds (and vice-versa), so in those rounds, one host must idle or wait for the other host to finish and send its data. Figure 5.1(b) illustrates the same computation under the BASP-model: here, the idle time has been completely eliminated.

While BASP exploits the resilience of graph analytics programs to stale reads to compensate for lack of load balance, stale reads may result in wasted computation. For example, under BSP execution, a host may ingest an update from another host and compute immediately with that value in the next round, whereas under BASP execution, the host may miss the update, compute with the stale value, and see the update only in a later round at which point it will need to repeat the computation with the updated value. Therefore, if load is already well-balanced under BSP execution, BASP execution may not be advantageous. We study these trade-offs by building and analyzing a BASP system.

5.3 Adapting Bulk-Synchronous Systems for Bulk-Asynchronous Execution

In this section, we describe how we adapted a BSP-style distributed and heterogeneous graph analytics system for BASP execution using the state-of-the-art communication substrate Gluon [52]. We first describe the changes required to Gluon application programs to make them amenable to BASP execution (Section 5.3.1). We then describe changes to Gluon to support BASP-style

execution (Section 5.3.2). We use the terms Gluon-Sync and Gluon-Async to denote BSP-style and BASP-style Gluon, respectively. Finally, we present a non-blocking termination detection algorithm that is required for BASP-style execution (Section 5.3.3). Based on our experience, we believe that other BSP systems can also be easily adapted to BASP.

5.3.1 Bulk-Asynchronous Programs

D-Galois [52] is the state-of-the-art distributed graph analytical system for CPUs. D-Galois programs are shared-memory Galois [134] programs that make calls to the Gluon(-Sync) communication substrate to synchronize distributed-memory computation. Figure 5.2 shows a code snippet for single-source-shortest-path (sssp) application. Each host processes its partition of the graph in rounds: computation is followed by communication. The compute phase (shown at Line 24) processes the vertices in the partitioned graph using a *push-style* operator (shown at Line 9) to compute and update the new distance values for their neighbors. The communication phase uses Gluon's communication interface, *i.e.*, the *sync()* method (shown at Line 25). Gluon is responsible for coordinating the communication among all hosts; at the end of this phase, all hosts have a consistent view of node labels. The application terminates when there is a round in which no host updates a node label. This can be detected using Gluon's distributed accumulator to determine the number of updates among all hosts in a round.

Figure 5.3 shows the same sssp application in the BASP programming

```
Graph* g;
struct GNode { // data on each node
  uint32_t dist_old;
  uint32_t dist_cur;
};
gluon::DistAccumulator<unsigned int> terminator;
...// sync structures
struct SSSP {
  void operator()(GNode src) const {
    if (src.dist_old > src.dist_cur) {
      terminator += 1; // do not terminate
      src.dist_old = src.dist_cur;
      for (auto dst : g->neighbors(src)) {
        uint32_t new_dist;
        new_dist = src.dist_cur + g->weight(src,dst);
        atomicMin(dst.dist_cur, new_dist);
      }
    }
  }
};
...// initialization, 1st round for source
do { // filter-based data-driven rounds
  terminator.reset();
  galois::do_all(g->begin(), g->end(), SSSP{&g});
  gluon::sync< .../* sync structures */ >();
} while(terminator.reduce());
```

Figure 5.2: Single source shortest path (sssp) application in BSP programming model.

```
1  Graph* g;
2  struct GNode { // data on each node
3    uint32_t dist_old;
4    uint32_t dist_cur;
5  };
6  gluon::DistTerminator<unsigned int> terminator;
7  ...// sync structures
8  struct SSSP {
9    void operator()(GNode src) const {
10     if (src.dist_old > src.dist_cur) {
11       terminator += 1; // do not terminate
12       src.dist_old = src.dist_cur;
13       for (auto dst : g->neighbors(src)) {
14         uint32_t new_dist;
15         new_dist = src.dist_cur + g->weight(src,dst);
16         atomicMin(dst.dist_cur, new_dist);
17       }
18     }
19   }
20 };
21 ...// initialization, 1st round for source
22 do { // filter-based data-driven rounds
23   terminator.reset();
24   galois::do_all(g->begin(), g->end(), SSSP{&g});
25   gluon::try_sync< ... /* sync structures */ >();
26 } while(terminator.cannot_terminate());
```

Figure 5.3: sssp application in BASP programming model. The modifications with respect to Figure 5.2 are highlighted.

model using Gluon-Async. The changes to the application are highlighted. The *try_sync* (*non-blocking*) call is responsible for coordinating the communication of labels among the hosts asynchronously. It ensures that each host eventually receives all the expected messages; in other words, it ensures that the hosts have a consistent view of node labels eventually. However, the challenge for each host then is to detect the termination of an application. This is handled efficiently using the *cannot_terminate()* method. The *cannot_terminate* (*non-blocking*) call is responsible for terminating if and only if no node labels can be updated on any host[2]. It ensures that no host terminates as long as some host has some computation or communication left to be completed. Since *try_sync*() and *cannot_terminate()* methods are non-blocking in nature, a host that performs synchronization can proceed to next round of computation phase without waiting for the communication process to complete. Thus, it may improve the performance.

While we explain these changes using D-Galois, the changes to other Gluon-based systems are similar because the only lines of code that changed are those related to Gluon. For example, in D-IrGL, the state-of-the-art distributed GPU graph analytical system, an IrGL compiler-generated CUDA kernel is called instead of `galois::do_all`, and the sync structures have CUDA kernels instead of CPU code. None of this needs to be changed to make the program amenable to BASP execution.

[2]The value set to *DistTerminator* on each host determines whether "no node labels are updated" or another quiescence condition is the termination criteria.

All programs that can be run asynchronously in existing distributed graph frameworks like PowerSwitch [167] and GRAPE+ [60] can use BASP. In addition, if a program can be run asynchronously in shared-memory, then it can use BASP on distributed-memory. In shared-memory, BSP programs can be made asynchronous if the program is resilient to stale reads and if computation is independent of the BSP round number. The same condition acts as a pre-requisite for changing BSP programs to BASP programs. For example, betweenness centrality [86] uses round number in its computation and requires BSP-style execution for correctness, so it cannot be changed for BASP-style execution. Most other BSP graph programs that have been used in the evaluation of distributed graph processing systems [52, 68, 74, 95, 173] can be changed to BASP-style execution by changing only a few lines of code.

5.3.2 Bulk-Asynchronous Communication

Recall from Section 5.2 that algorithm execution in both Gluon-Sync and Gluon-Async is done in local rounds where each round performs bulk-computation followed by bulk-communication. The bulk-communication itself involves a reduce phase followed by a broadcast phase. Thus, each round has 3 phases: computation, reduce, and broadcast. The computation phase is identical in Gluon-Sync and Gluon-Async, but the other phases differ.

The reduce and broadcast phases are blocking in Gluon-Sync and non-blocking in Gluon-Async. In Gluon-Sync, hosts exchange messages in each phase (even if the message is empty) and hosts wait to receive these messages;

this acts like an implicit barrier. Messages are sent in the reduce or broadcast phase of Gluon-Async only if there are updates to mirror nodes (empty messages are not required due to relaxation of synchronization barriers) and no host waits to receive a message. The action for the received messages in Gluon-Async depend on whether they were sent in the reduce or broadcast phase. As there are two phases and messages could be delivered out-of-order, we distinguish between messages sent in reduce and broadcast phases using tags. We describe this more concretely next.

Let host h_i have the set of mirror proxies P_i for which the set of master proxies P_j are on host h_j. Let U_i be the set of mirror proxies on h_i that are updated in round r (by definition, $U_i \subseteq P_i$). Let U_j be the master proxies on h_j that are updated in round r, during either computation or reduce phases (by definition, $U_j \subseteq P_j$).

In Gluon-Sync, the Gluon substrate performs the following operations for every pair of h_i and h_j ($h_i \neq h_j$):

- Reduce phase for h_i: Sends one message m_R containing values of U_i to h_j (if $U_i = \emptyset$, then an empty message is sent) and resets the values of U_i to the identity element of the reduction operation.
- Reduce phase for h_j: *Waits* to receive m_R from h_i and, once received, uses the reduction operator and the values in m_R to update the corresponding master proxies in P_j.

- Broadcast phase for h_j: Sends one message m_B containing values of U_j to h_i (if $U_j = \emptyset$, then an empty message is sent).

- Broadcast phase for h_i: *Waits* to receive m_B from h_j and, once received, uses the values in m_B to set the corresponding mirror proxies in P_i.

To support BASP-style execution of Gluon-Async, we modified the Gluon communication-optimizing substrate to perform the following operations (instead of the above) for every pair of h_i and h_j ($h_i \neq h_j$):

- Reduce phase for h_i: If $U_i \neq \emptyset$, sends a reduce-tagged message m_R containing values of U_i to h_j and resets the values of U_i to the identity element of the reduction operation.

- Reduce phase for h_j: For every reduced-tagged message m_R received from h_i, uses the reduction operator and the values in m_R to update the corresponding master proxies in P_j.

- Broadcast phase for h_j: If $U_j \neq \emptyset$, sends a broadcast-tagged message m_B containing values of U_j to h_i.

- Broadcast phase for h_i: For every broadcast-tagged message m_B received from h_j, uses *the reduction operator* and the values in m_B to update the corresponding mirror proxies in P_i.

If the reduction operator is not used in the broadcast phase of Gluon-Async, algorithms may not yield correct results (or even converge). To illus-

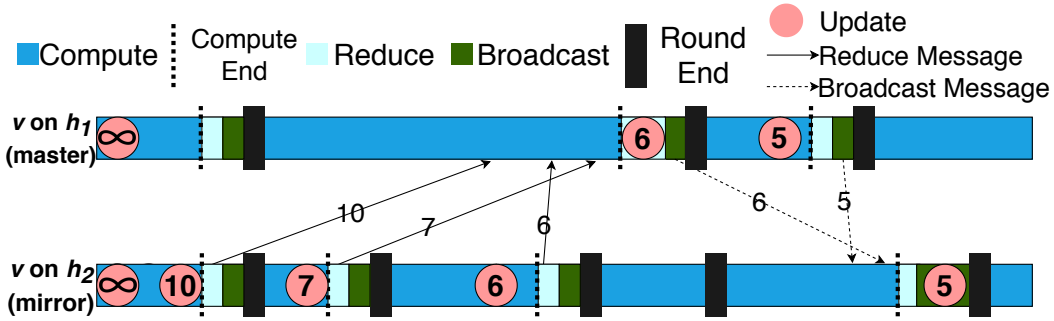


Figure 5.4: Illustration of communication in Gluon-Async.

trate this with an example, we show the synchronization of proxies in Figure 5.4 for the single-source shortest path (sssp) code in Gluon-Async (shown in Figure 5.3). The label `dist_current`. shortened as d_c, is reduced during computation using the "minimum" operation. Consider a vertex v with proxies on hosts h_1 and h_2, where the master proxy is on h_1 and the mirror proxy is on h_2. The label d_c is initialized to ∞ on both proxies. Say host h_2 sends values 10, 7, and 6 after executing its local rounds 1, 2, and 3, respectively. Say host h_1 receives all these values in the order 10, 6, and 7 at the end of its round 2. Host h_1, which still has ∞ value for its proxy, reduces the received values one-by-one, yielding the update 6, and broadcasts this value to h_2. Host h_1 reduces its proxy value during computation to 5 and broadcasts it to h_2 after its round 3. Host h_2 receives both these values in the order of 5 and 6. The mirror proxy value on h_2 is 6 (because reset is a *no-op* for minimum operation). If host h_2 had set the received values (in order) like in Gluon-Sync, then the final value of h_2 would be 6, which would be incorrect. Host h_2 instead reduces the received values one-by-one yielding the update 5.

The proxies on both hosts are not updated thereafter and thus, both proxies have the same values.

An important point to note is that if the message is not empty, then Gluon-Sync and Gluon-Async send the same message. Gluon-Async thus retains the underlying advantages of Gluon-Sync. Gluon-Async supports any partitioning policy and performs bulk-communication, thereby utilizing Gluon's communication optimizations that exploit structural and temporal invariants in partitioning policies [52]. Gluon-Async can be plugged into different CPU or GPU graph analytics systems to build distributed-memory versions of those systems that use BASP-style execution. As shown in Figure 5.1, communication between a GPU device and its host is a local operation. Gluon-Async treats this as a blocking operation like Gluon-Sync. While this can be made non-blocking too, it is outside the scope of this thesis.

We showed that BASP-style execution can be used in Gluon-Async without any blocking or waiting operations among hosts. The messages, if any, will be eventually delivered. The key to this is that hosts must not terminate until there are messages left to be delivered. This requires non-blocking termination detection, which we explain next.

5.3.3 Non-blocking Termination Detection

BASP-style execution requires a more complicated termination algorithm than BSP-style execution. We describe a non-blocking termination detection algorithm that uses snapshots to implement a distributed consensus

protocol [106] that does not rely on message delivery order.

The algorithm is based on a state machine maintained on each host. At any point of time, a host is in one of five states: Active (A), Idle (I), Ready-to-Terminate$_1$ (RT_1), Ready-to-Terminate$_2$ (RT_2), and Terminate (T). The goal of termination detection is that a host should move to T if and only if every other host will move to T. We describe state transitions and actions for ensuring this.

Hosts coordinate with each other by taking non-blocking snapshots that are numbered. When a host takes a snapshot n, it broadcasts its current state to other hosts (non-blocking). Once a host h takes the snapshot n, it cannot take the next snapshot $n+1$ until h knows that every other host has taken the snapshot n. In other words, before h takes the next snapshot $n+1$, h should not only have completed the broadcast it initiated for n but also have received broadcast messages from every other host for n. Thus, eventually, every host will know the states that all other hosts took their snapshots from. For example, all hosts will know whether all hosts took the snapshot n from the same state RT_2 or not. We use this knowledge to transition between states.

Each host has a dedicated communication thread that is started when the program begins (and terminated when program ends). It receives messages throughout program execution. Every host takes a (dummy) snapshot initially. Subsequent snapshots are taken by a host h only if h is ready to terminate. Intuitively, hosts can terminate only if every host knows that "every host knows that every host wants to terminate". This requires two consecutive snapshots

to be taken with all hosts indicating that they are ready-to-terminate (RT). We use RT1 and RT2 to distinguish between two consecutive snapshots of RT.

On each host h, the termination detection algorithm is invoked at the end of each local round r; all the state transitions occur only at this point in the program. Note that r is incremented each time *cannot_terminate()* is invoked (see Figure 5.3 for example). Let n be the last snapshot that h has taken. When the termination detection algorithm is invoked, we first check if h is *inactive*, *inspected*, or *affirmed*.

A host h is considered to be *inactive* if the following conditions hold:

1. No label was updated in round r in computation, reduce, or broadcast phases.
2. All non-blocking sends initiated on this host are complete.
3. All non-blocking receives initiated on this host are complete.

The first condition checks whether work was done in r while the other conditions check whether any work is still pending. These conditions must hold for h to take the next snapshot $n + 1$.

A host h is considered to be *inspected* if it knows that all the hosts have taken the previous snapshot n. This condition must hold for h to take the next snapshot $n + 1$. Similarly, a host h is considered to be *affirmed* if (i) h has been *inspected* and (ii) it knows that all the hosts have taken the

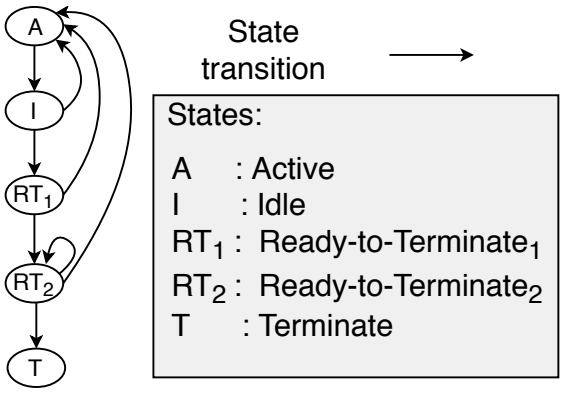


Figure 5.5: State transition diagram for termination detection.

previous snapshot n from state RT_2 (that is, other hosts have also affirmed their readiness to terminate). This condition must hold for h to terminate.

Initially, every host is in state A. Figure 5.5 shows the possible state transition on a single host. Table 5.1 shows the conditions that must hold for each state transition and the action, if any, taken after the state transition. No action is taken with transitions to states A and I. When h transitions to RT_1 or RT_2, it takes a snapshot. When h transitions to T, h decides to terminate (returns false in Line 26 in Figure 5.3). A host moves from A to I only if the host is *inactive*. If a host is not *inactive*, then it moves to A from the I, RT_1, or RT_2 states. If h is *inspected* and is in I, then it moves to RT_1. If h is *inspected* and is in RT_1, it moves to RT_2. If h is *affirmed*, then it moves from RT_2 to T.

Consider an example with two hosts, h_1 and h_2. Initially, both of them initiate (dummy) snapshot n_0. When h_2 becomes *inactive*, it moves to I. As both hosts initiated the previous snapshot n_0, h_2 moves to RT_1 and initiates

Start State	**Condition for state transition (boolean formula)**	**End State**	**Action**
A	inactive	I	
I	¬inactive	A	
I	inactive ∧ inspected	RT_1	Snapshot
RT_1	¬inactive	A	
RT_1	inactive ∧ inspected	RT_2	Snapshot
RT_2	¬inactive	A	
RT_2	inactive ∧ inspected ∧ ¬affirmed	RT_2	Snapshot
RT_2	affirmed	T	Terminate

Table 5.1: Conditions required for state transitions during termination detection.

the next snapshot n_1. Meanwhile, h_1 sends a message to h_2, becomes inactive, and moves to I. As n_0 has ended, h_1 moves to RT_1 and initiates n_1. In the next round, h_1 detects that h_2 also has initiated n_1. Note that it would be incorrect for h_1 to terminate at this point, although both h_1 and h_2 initiated n_1 from RT_1. Our algorithm uses two RT states to detect this, so h_1 moves to RT_2 instead of terminating and initiates the next snapshot n_2. During this time, h_2 received the message from h_1 which made it *active* and moved it to A. Later, it moves to I and then RT_1 to initiate n_2. In the next round, h_2 observes that n_2 has ended, so it moves to RT_2 and initiates n_3. h_1 also observes that n_2 has ended and initiates n_3 while remaining in RT_2. Now, in the next round on both hosts, each host observes that n_3 has ended and that the other host has initiated n_3 from RT_2, so both hosts *affirm* to terminate and move to T.

To implement our termination detection algorithm in Gluon-Async (Line 26 in Figure 5.3), we use non-blocking collectives to take a snapshot.

For the reduce and broadcast phases, we modify the communication substrate to send messages in *synchronous* mode instead of *standard* mode. In *standard* communication mode of MPI or LCI [51], a send (call) may complete before a matching receive is invoked. Hence, both the sender and the receiver may become inactive and terminate while the message is still in-flight. In contrast, in *synchronous* mode, a send is considered complete only if the receiver has initiated receive. Consequently, when a message is in-flight, either the sender or the receiver is in active state A. Thus, *synchronous* communication mode sends are necessary for our termination detection protocol. Note that our protocol does not rely on the order of message delivery of Gluon or the underlying communication substrate such as MPI or LCI [51].

Note that goal of termination detection is that a host should move to T if and only if every other host will move to T. We now argue how our termination detection algorithm satisfies this property. A non-active, non-terminated host h can move back to state A only if it receives data from another host – in this case, the *inactive* flag will become false. Since the program is correct, at least one host will not reach the RT_2 state until the final value(s) are computed (no false detection of termination). A host h can reach the state RT_2 from RT_1 or RT_2 only if it is *inspected* and *inactive*, which means that h did not update any labels and did not send nor receive data. If every host took the snapshot from RT_2, then no host computed, sent, or received data between two snapshots. Consequently, no host can receive a message and move to A after that, so every host must terminate.

5.4 Experimental Evaluation

In this section, we evaluate the benefits of Bulk-Asynchronous Parallel (BASP) execution over Bulk-Synchronous Parallel (BSP) execution using D-Galois [52] and D-IrGL [52], the state-of-the-art graph analytics systems for distributed CPUs and distributed GPUs, respectively. Both these systems are built on top of a Gluon [52]. In this section, we use the name Gluon-Sync to refer to these two systems. We modified D-Galois and D-IrGL BSP programs as described in Section 5.3.1 to make them amenable for BASP-style execution. As described in Sections 5.3.2 and 5.3.3, we modified Gluon to support BASP-style execution for both systems. In this section, we use the name Gluon-Async to refer to both systems (source code is publicly available [2]).

We also compare the performance of Gluon-Async with that of Lux [95], which is a multi-host multi-GPU graph analytical framework that uses BSP-style execution; note that there are no asynchronous distributed GPU graph analytical systems to compare against. GRAPE+ [60] and PowerSwitch [167] are asynchronous distributed CPU-only graph systems, and we compare them with Gluon-Async.

We first describe our experimental setup (Section 5.4.1). We then present our evaluation on distributed GPUs (Section 5.4.2) and distributed CPUs (Section 5.4.3). Finally, we analyze BASP and BSP (Section 5.4.4) and summarize our results (Section 5.4.5).

Table 5.2: Input graphs and their key properties (we classify graphs with estimated diameter > 200 as high-diameter graphs).

	Small graphs			
	twitter50	**rmat27**	**friendster**	**uk07**
$\|V\|$	51M	134M	66M	106M
$\|E\|$	1,963M	2,147M	1,806M	3,739M
$\|E\|/\|V\|$	38	16	28	35
Max OutDegree	779,958	453M	5,214	15,402
Max InDegree	3.5M	21,806	5,214	975,418
Estimated Diameter	12	3	21	115
Size (GB)	16	18	28	29

	Large graphs				
	gsh15	**clueweb12**	**uk14**	**wdc14**	**wdc12**
$\|V\|$	988M	978M	788M	1,725M	3,563M
$\|E\|$	33,877M	42,574M	47,615M	64,423M	128,736M
$\|E\|/\|V\|$	34.3	43.5	60.4	37	36
Max OutDegree	32,114	7,447	16,365	32,848	55,931
Max InDegree	59M	75M	8.6M	46M	95M
Estimated Diameter	95	498	2,498	789	5,274
Size (GB)	260	325	361	493	986

5.4.1 Experimental Setup

We conducted all the GPU experiments on the Bridges cluster [135] at the Pittsburgh Supercomputing Center [4, 157]. Each machine in the cluster is configured with 2 NVIDIA Tesla P100 GPUs and 2 Intel Broadwell E5-2683 v4 CPUs with 16 cores per CPU, DDR4-2400 128GB RAM, and 40MB LLC. The machines are interconnected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). We use up to 64 GPUs (32 machines). All benchmarks were compiled using CUDA 9.2, GCC 7.3, and MVAPICH2 2.3b.

All the CPU experiments were run on the Stampede2 [153] cluster lo-

cated at the Texas Advanced Computing Center. Each machine is equipped with 2 Intel Xeon Platinum 8160 "Skylake" CPUs with 24 cores per CPU, DDR4 192GB RAM, and 66MB LLC. The machines in the cluster are interconnected through Intel Omni-Path Architecture (peak bandwidth of 100Gbps). We use 48 threads on each machine and up to 128 machines (6144 cores or threads). Benchmarks were compiled with GCC 7.1 and IMPI 17.0.3.

Table 5.2 shows the input graphs along with their key properties: twitter50 [18, 20] and friendster [116] are social network graphs; rmat27 is a randomized synthetically generated graph using with an RMAT generator [34]; uk07, gsh15, clueweb12 [142], uk14 [18–20], wdc14, and wdc12 [126] are among the largest public web-crawls (wdc12 is the largest publicly available graph). Table 5.2 splits the graphs into two categories: small and large. Small graphs are only used for comparison with Lux, GRAPE+, and PowerSwitch (we could not run these systems using the large graphs), while we use large graphs for all other experiments. We also classify the graphs based on their estimated (observed) diameter. All small graphs are low-diameter graphs with diameter < 200, while all large graphs, except gsh15, are high-diameter graphs with diameter > 200.

We evaluated our framework with 5 benchmarks: breadth-first-search (bfs), connected components (cc), k-core (kcore), pagerank (pr), and single source shortest path (sssp). For pr, we used a tolerance of 10^{-6}. For bfs and sssp, we considered the vertex with maximum out-degree as the source. For kcore, we use a k of 100. All benchmarks are executed until convergence. We

report the total execution time, excluding the graph loading, partitioning, and construction time. The reported results are a mean over three runs.

For Gluon-Sync and Gluon-Async, the partitioning policy is configurable as it uses the CuSP streaming partitioner [85]. Based on the recommendations of a large-scale study [70], we choose the Cartesian Vertex Cut (CVC) [21, 52] for all our experiments[3]. We use LCI [51] instead of MPI for message transport among hosts[4].

For Lux, we only present results for cc and pr as the other benchmarks are not available or produce incorrect output. pr in Lux does not have a convergence criterion, so we executed it for the same number of rounds as that of Gluon-Sync[5] (Gluon-Async might execute more rounds to converge). Note that Lux uses an edge-cut partitioning policy and dynamically re-partitions the graph to balance the load.

GRAPE+ [60] is not publicly available. We present results used in their paper (and provided by the authors). They use a total of 196 cores in their study; to compare with them, we use 12 machines of Stampede with 16 threads (196 cores). They use partitions provided by XtraPulp [151]. They present results only for cc, pr, and sssp on friendster. When comparing with them, we use the same partitioning policy, we use the same source nodes for sssp (5506215, 6556728, 1752217, 3391590, 782658), and we use the same tolerance

[3]sssp, clueweb12, GPUs uses Outgoing Edge Cut due to memory limits.
[4]Dang et al. [51] show the benefits of LCI over MPI for graph applications.
[5]Both Gluon-Sync and Lux are BSP-style and use the same algorithm.

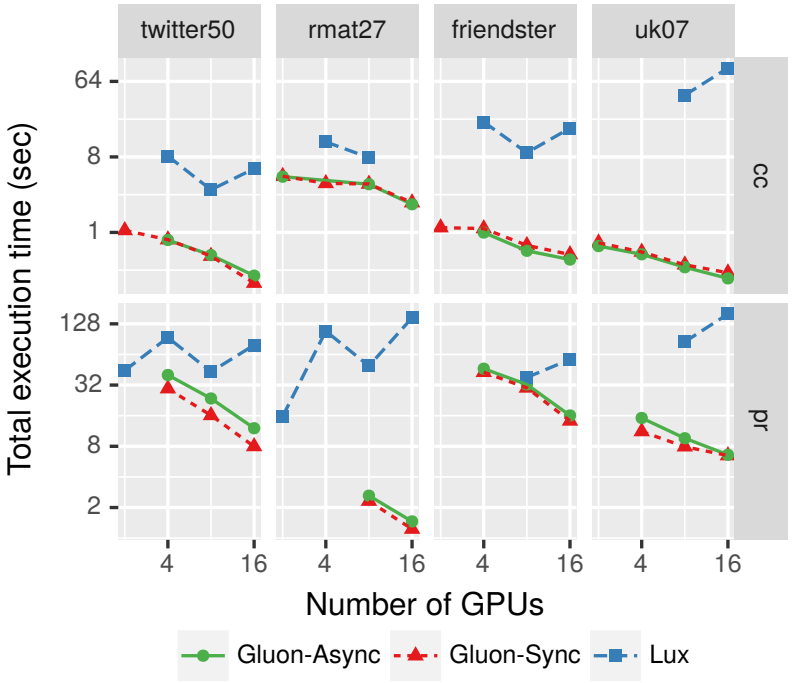


Figure 5.6: Strong scaling (log-log scale) of Lux, Gluon-Sync, and Gluon-Async for small graphs on Bridges (2 P100 GPUs share a physical machine).

for pr (10^{-3}). For a relative comparison, we also present the corresponding PowerSwitch [167] results from their paper [60]. We do not evaluate PowerSwitch ourselves because it is an order of magnitude slower.

5.4.2 Distributed GPUs

Small graphs: Figure 5.6 shows the total execution time of Gluon-Async, Gluon-Sync, and Lux on small graphs using up to 16 GPUs. Missing points indicate that the system ran out of memory (except for Lux with cc on rmat27 using 16 GPUs, which failed due to a crash). The major trend in the figure is that both Gluon-Async and Gluon-Sync always outperform Lux and scale bet-

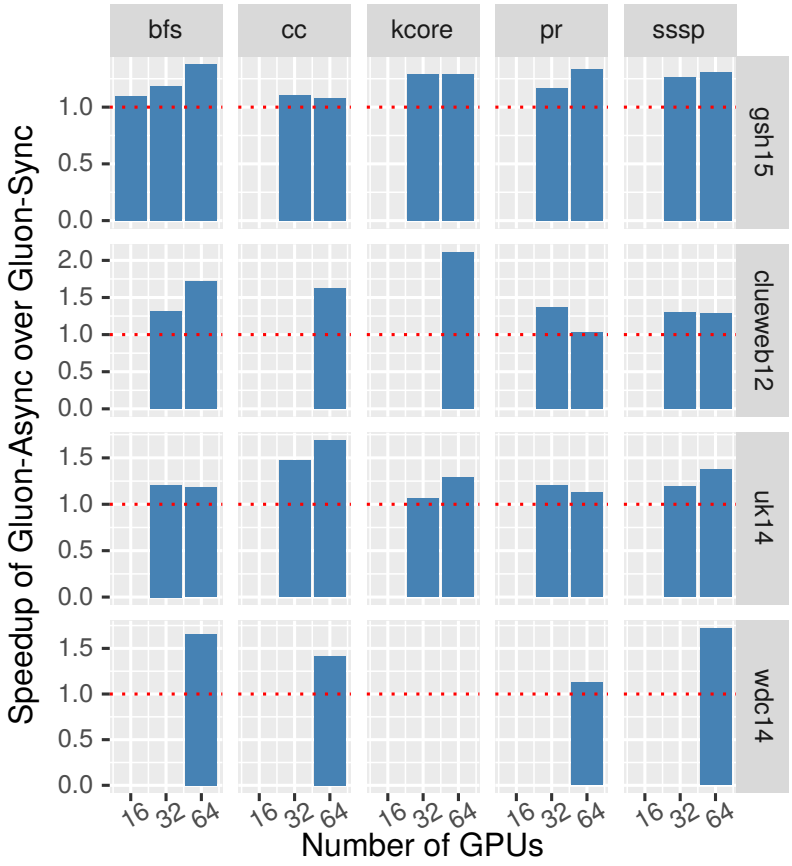


Figure 5.7: Speedup of Gluon-Async over Gluon-Sync for large graphs on Bridges (2 P100 GPUs share a physical machine).

ter. It is also clear that Gluon-Async and Gluon-Sync perform quite similarly. In some cases, Gluon-Async is also noticeably slower (pr on twitter50). We do not expect Gluon-Async to perform better than Gluon-Sync for low-diameter graphs like these because most benchmarks in Gluon-Sync execute very few (< 100) rounds for these. We will analyze this later using larger graphs (Section 5.4.4). Nevertheless, both Gluon-Async and Gluon-Sync are on average $\sim 12\times$ faster than Lux.

Table 5.3: Total execution time of Gluon-Sync and Gluon-Async on 192 cores of Stampede; PowerSwitch and GRAPE+ on 192 cores of a different HPC cluster [60].

Benchmark	Input	PowerSwitch	GRAPE+	Gluon-Sync	Gluon-Async
cc	**friendster**	61.1	10.4	1.7	1.7
pr		85.1	26.4	21.3	21.9
sssp		32.5	12.7	5.8	5.5

Large Graphs: Figure 5.7 shows the speedup in total execution time of Gluon-Async over Gluon-Sync for large graphs using up to 64 GPUs (Lux runs out of memory for all the large graphs, even on 64 GPUs). Missing points indicate that either Gluon-Sync or Gluon-Async ran out of memory (almost always, if one runs out of memory, the other also does; only in a couple of cases, Gluon-Async runs out of memory but Gluon-Sync does not because Gluon-Async may use more communication buffers). 64 GPUs are insufficient to load wdc12 as input, partition it, and construct it in memory; so both Gluon-Sync and Gluon-Async run out of memory. It is apparent that Gluon-Async always outperforms Gluon-Sync for large graphs. We observe that the speedup depends on both the input graph and the benchmark. Typically, speedup is better for clueweb12 and wdc14 than gsh15. The speedup is also usually lower for pr than for other benchmarks. We also see that in most cases, the speedup of Gluon-Async over Gluon-Sync increases with an increase in the number of GPUs. This indicates that Gluon-Async scales better than Gluon-Sync. For high-diameter graphs on 64 GPUs, Gluon-Async is on average $\sim 1.4\times$ faster than Gluon-Sync.

5.4.3 Distributed CPUs

Small graphs: Table 5.3 shows the total execution time of PowerSwitch, GRAPE+, Gluon-Sync, and Gluon-Async for friendster with 192 threads. Note that Gluon-Sync and Gluon-Async used machines on Stampede, whereas PowerSwitch and GRAPE+ used machines on a different HPC cluster. Similar to GPUs, the performance differences between Gluon-Async and Gluon-Sync are negligible because friendster is a low-diameter graph. Although both GRAPE+ and PowerSwitch are asynchronous systems, they are much slower than Gluon-Sync and Gluon-Async. Both Gluon-Sync and Gluon-Async are on average $\sim 2.5\times$ and $\sim 9.3\times$ faster than GRAPE+ and PowerSwitch, respectively. This shows that a well-optimized existing bulk-synchronous system (Gluon-Sync) beats the existing asynchronous systems and that it is challenging to reap the benefits of asynchronous execution. Gluon-Sync uses Galois [134] computation engine and Gluon [52] communication engine. Both have several optimizations that help Gluon-Sync outperform PowerSwitch and GRAPE+. It is not straightforward to incorporate these optimizations in PowerSwitch and GRAPE+ due to the way they perform asynchronous communication. Gluon-Async introduces a novel way for asynchronous execution while retaining all the performance benefits of on-device computation engines like Galois and IrGL [137] and the inter-device communication engine, Gluon. While Gluon-Sync and Gluon-Async perform similarly for small graphs, we show that on large graphs, Gluon-Async can be much faster than Gluon-Sync.

Large graphs: Figure 5.8 shows the speedup in total execution time of Gluon-

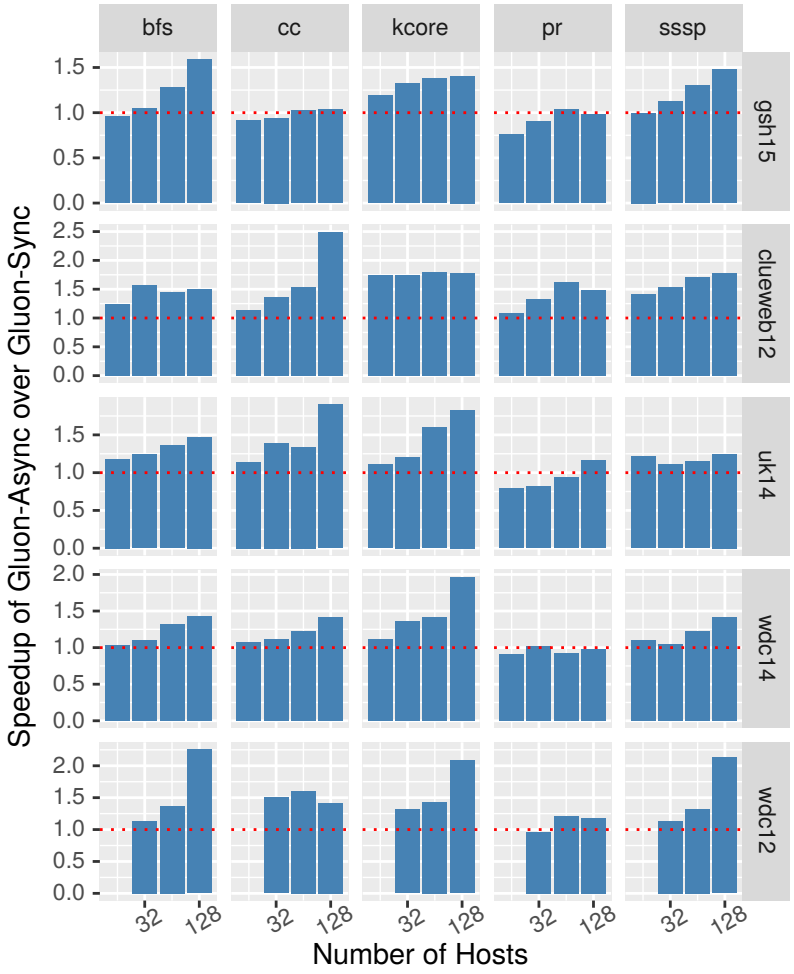


Figure 5.8: Speedup of Gluon-Async over Gluon-Sync for large graphs on Stampede (each host is a 48-core Skylake machine).

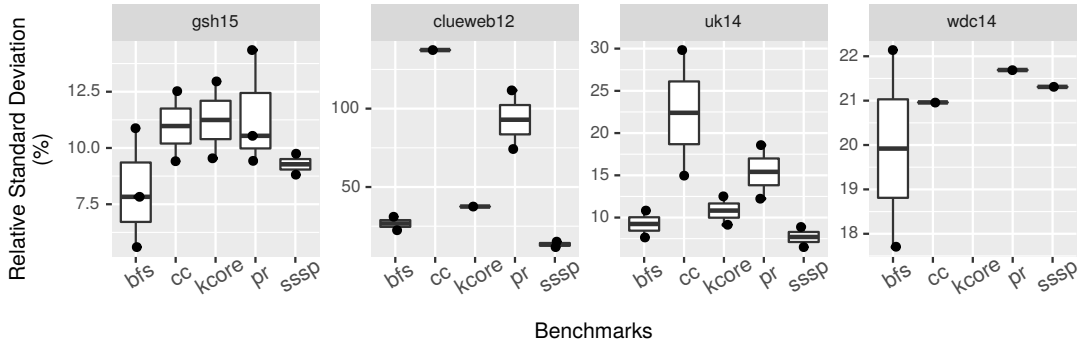
Table 5.4: Minimum BSP-rounds for Gluon-Sync on CPUs.

Input	Estimated	Minimum Number of Rounds				
	Diameter	bfs	cc	kcore	pr	sssp
gsh15	95	61	11	239	172	62
clueweb12	498	184	25	696	161	200
uk14	2,498	1,825	80	443	161	1,976
wdc14	789	503	196	146	180	507
wdc12	5,274	2,672	401	277	183	3,953

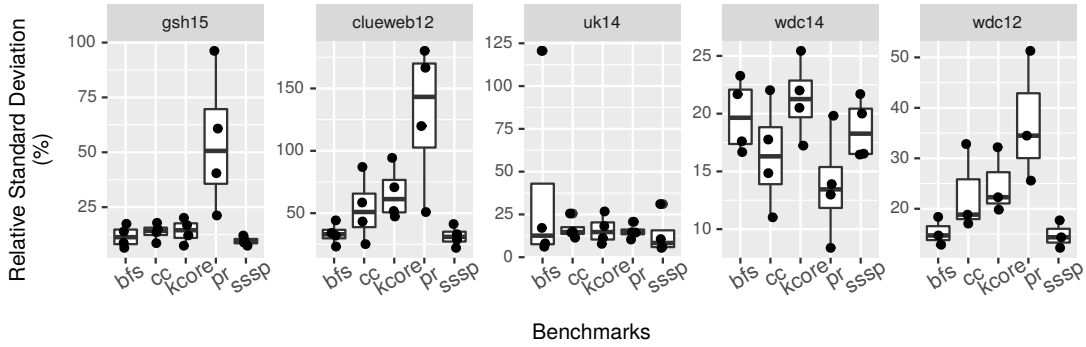
Async over Gluon-Sync for large graphs using up to 128 Skylake machines or hosts. Missing points indicate that either Gluon-Sync or Gluon-Async ran out of memory. The trends are similar to those on GPUs. The speedup depends on both the input graph and the benchmark. Gluon-Async mostly outperforms Gluon-Sync; its performance is similar or lower than that of Gluon-Sync on 64 or fewer hosts in some cases for pr or in some cases for the input gsh15. The speedup of Gluon-Async over Gluon-Sync increases with the increasing number of hosts indicating that on distributed CPUs also, Gluon-Async scales better than Gluon-Sync. For high-diameter graphs on 128 CPUs, Gluon-Async is on average $\sim 1.6\times$ faster than Gluon-Sync.

5.4.4 Analysis of BASP and BSP

Using Gluon-Async and Gluon-Sync, we now analyze the performance difference between BASP-style and BSP-style execution, respectively, on both distributed GPUs and CPUs. Specifically, we focus on: (1) *why* the difference arises (load imbalance), (2) *where* the difference exists (idle time), and (3) *how* the difference manifests itself (rounds executed).



(a) GPUs on Bridges



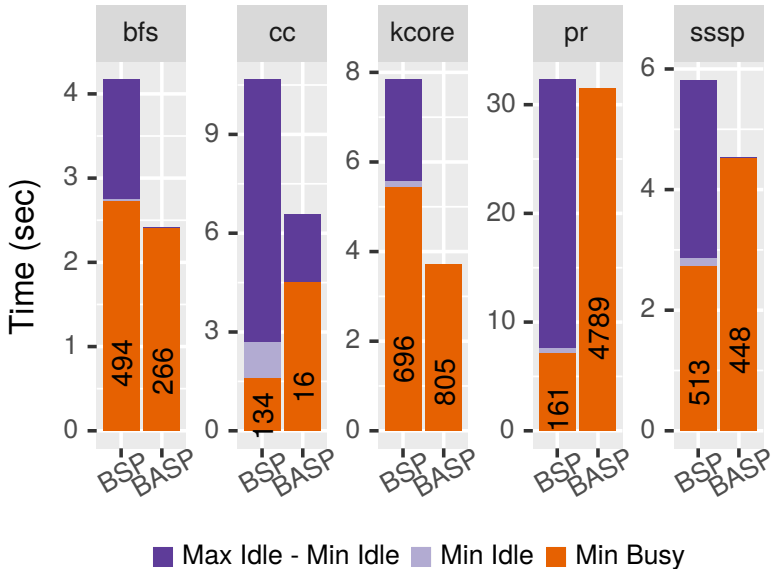
(b) CPU hosts on Stampede

Figure 5.9: Load imbalance in Gluon-Sync (presented as relative standard deviation in computation times among devices).

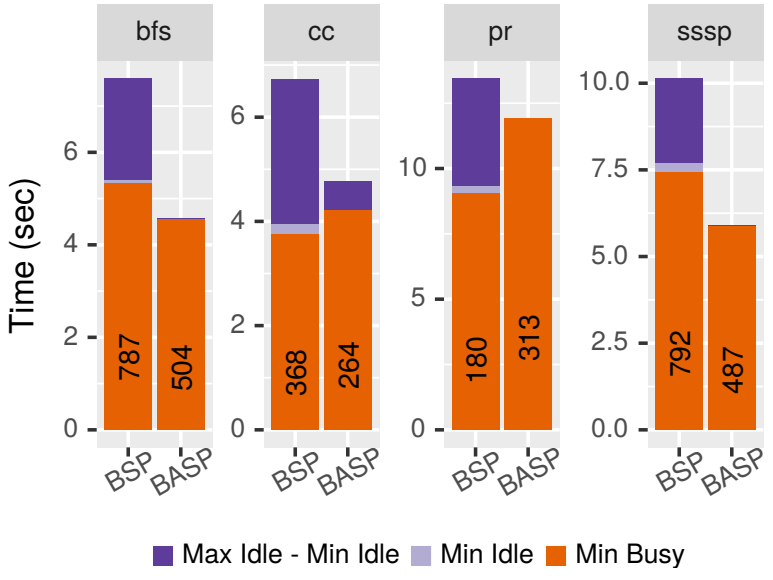
Load imbalance: Table 5.4 shows the number of rounds executed by benchmarks in Gluon-Sync for the large graphs. It can be observed that higher diameter graphs are likely to execute more rounds, except for pr. We next measure the load imbalance by calculating the total time spent by each host in computation and determine the relative standard deviation (standard deviation by mean) of these values. Figures 5.9(a) and 5.9(b) presents these values for Gluon-Sync as a box-plot[6] for all the number of devices (CPUs or GPUs) for each benchmark and input graph on Bridges and Stampede, respectively. Each point in a box-plot is a value for a distinct configuration of the number of devices (CPUs or GPUs) for that benchmark and input graph. The load imbalance and the number of rounds can be used to tell whether Gluon-Sync can benefit from switching to BASP-style execution. As cc on gsh15 is well balanced and executes very few rounds, it does not benefit much from BASP-style execution. In contrast, benchmarks using clueweb12 are more imbalanced and benefit significantly from BASP-style execution, even if it executes very few rounds like in cc. For high-diameter graphs, load balance is difficult to achieve in efficient *data-driven* graph applications [141] because different subsets of nodes may be updated in different rounds. We show that Gluon-Async circumvents this by using BASP-style execution.

Idle time: We define busy time of a host as the time spent in computation, serialization (for packing messages to be sent), deserialization (for unpacking

[6]The box for an input graph and benchmark represents the range of 50% of these values for that input graph and benchmark; the line dividing the box is the median of those values and the circles are outliers.

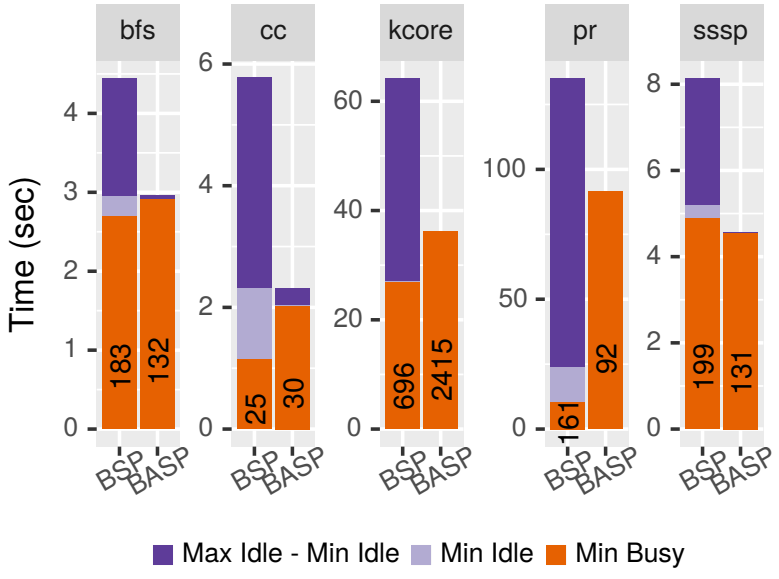


(a) clueweb12

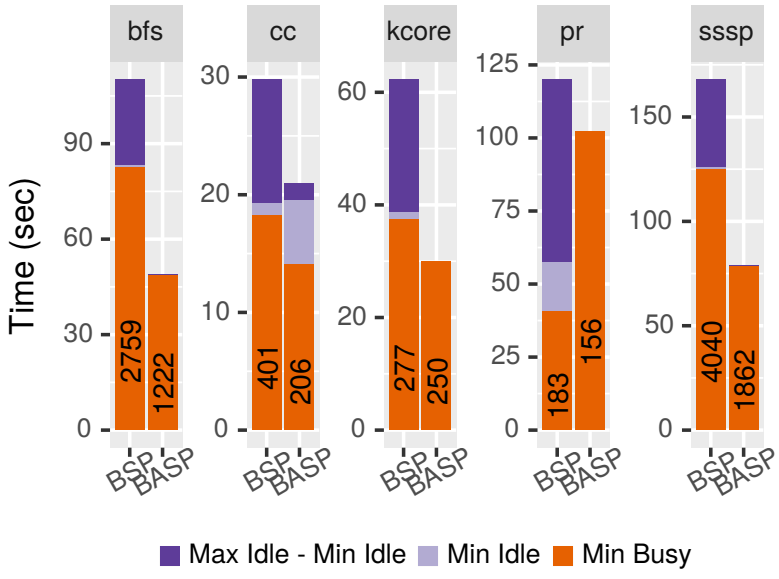


(b) wdc14

Figure 5.10: Breakdown of execution time (sec) on 64 P100 GPUs of Bridges; the minimum number of rounds executed among hosts is shown on each bar.



(c) clueweb12



(d) wdc12

Figure 5.11: Breakdown of execution time (sec) on 128 hosts of Stampede; the minimum number of rounds executed among hosts is shown on each bar.

and applying received messages), and communication between host and device. The rest of the total time is the idle time; in BASP, idle time includes the time to detect termination. Different hosts can have different busy and idle times (stragglers have smaller idle times), so we consider the minimum and maximum across hosts. Figure 5.11 show the breakdown of execution time into minimum busy time, minimum idle time, and the difference between maximum and minimum idle time. As expected, BSP has high maximum idle time due to load imbalance and BASP reduces idle time, which is one of the main advantages of having bulk-asynchronous execution. However, this reduction in idle time could lead to a corresponding increase in busy time because the host could be doing redundant or useless *work* by operating on stale values instead of being idle. This depends on the input graph and the benchmark. In some cases like pr, the busy time increases even though the idle time is reduced. In most other cases, the busy time does not increase by much. Nevertheless, it is clear that the difference between BASP and BSP is in the idle time, and the total execution time will be reduced only if the idle time is reduced without an excessive increase in busy time.

Rounds executed: All hosts execute the same number of rounds in BSP (Table 5.4), whereas different hosts may execute different numbers of local rounds in BASP. The minimum rounds executed in BSP and BASP are shown on each bar in Figure 5.11. We use the minimum local rounds among hosts to estimate the critical path in the execution. We count the number of edges processed (locally) on each host and use the maximum among hosts to estimate the

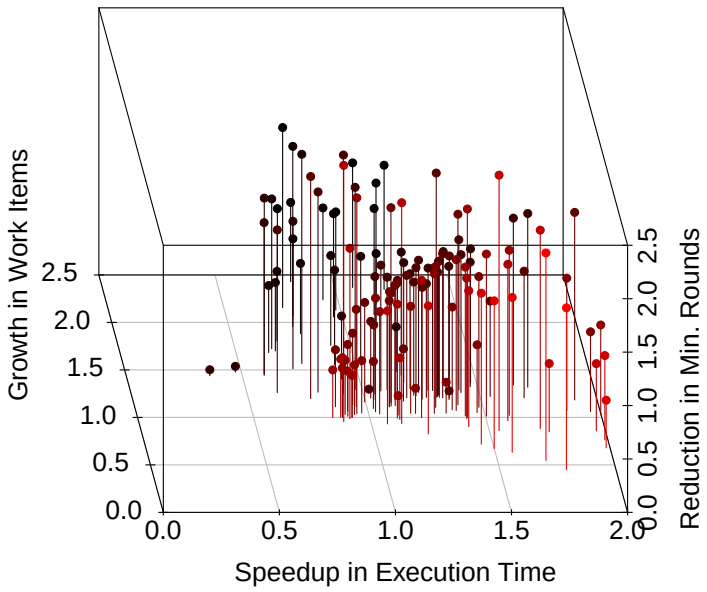


Figure 5.12: BASP over BSP: correlation between speedup, growth in maximum # local work items, and reduction in minimum # local rounds for all benchmarks, inputs, and devices (CPUs or GPUs). Red color indicates lower growth in work items.

work done in the execution. Figure 5.12 presents the correlation between the speedup in execution time, the increase or growth in the work done (maximum local work items or edges processed), and the reduction in the critical path (minimum local rounds). Each point is a value for a distinct configuration of benchmark, input, and number of devices (CPU or GPU); we have omitted outliers. Red (closer) points have lower growth in the work done and higher points (taller lines) have more reduction in the critical path. If BASP reduces both the work done (growth < 1) and the critical path (reduction > 1), then it would obviously be faster. As shown in the figure, BASP is faster than BSP (speedup > 1) when work done is reduced. More importantly, BASP does more work than BSP in many cases, but it is faster due to a reduction in the critical path. When BASP is slower than BSP (speedup < 1), it is due to a high growth in work done without sufficient reduction in critical path. Although the minimum number of local rounds in BASP may be smaller than that of BSP, the maximum number of local rounds in BASP may be higher because the faster hosts need not wait and may execute more local rounds. Instead of waiting after every round in BSP, faster hosts in BASP may execute more rounds. Consequently, faster hosts could make more progress and send updated values to the stragglers or slower hosts. The straggler hosts receive these updated values before they move to the next round, saving them from doing redundant work using stale values. Thus, straggler hosts doing fewer local rounds leads to faster convergence in BASP.

Table 5.5: Fastest execution time (sec) of Gluon-Sync and Gluon-Async using the best-performing number of hosts (# of hosts in parenthesis; "-" indicates out of memory).

Bench-mark	Input	CPUs (Stampede)		GPUs (Bridges)	
		Gluon-Sync	Gluon-Async	Gluon-Sync	Gluon-Async
bfs	gsh15	1.3 (128)	**0.8 (128)**	0.9 (64)	**0.7 (64)**
	clueweb12	4.5 (128)	**3.0 (128)**	4.2 (64)	**2.4 (64)**
	uk14	13.0 (128)	**8.8 (128)**	8.8 (64)	**7.4 (64)**
	wdc14	9.3 (128)	**6.5 (128)**	7.6 (64)	**4.6 (64)**
	wdc12	110.3 (128)	**48.9 (128)**	-	-
cc	gsh15	**1.0 (128)**	**1.0 (128)**	1.2 (64)	**1.1 (64)**
	clueweb12	5.8 (128)	**2.3 (128)**	10.7 (64)	**6.6 (64)**
	uk14	2.2 (64)	**1.3 (128)**	10.4 (64)	**6.1 (64)**
	wdc14	7.3 (128)	**5.2 (128)**	6.7 (64)	**4.8 (64)**
	wdc12	29.8 (128)	**21.0 (128)**	-	-
kcore	gsh15	9.8 (128)	**6.9 (128)**	3.0 (64)	**2.3 (64)**
	clueweb12	64.3 (128)	**36.2 (128)**	7.8 (64)	**3.7 (64)**
	uk14	11.8 (128)	**6.4 (128)**	2.2 (64)	**1.7 (64)**
	wdc14	18.4 (128)	**9.4 (128)**	-	-
	wdc12	62.4 (128)	**29.9 (128)**	-	-
pr	gsh15	47.3 (128)	**46.2 (64)**	14.0 (64)	**10.5 (64)**
	clueweb12	130.5 (16)	**91.6 (128)**	32.3 (64)	**24.9 (32)**
	uk14	11.7 (128)	**10.1 (128)**	6.3 (64)	**5.6 (64)**
	wdc14	**24.7 (128)**	25.3 (128)	13.4 (64)	**11.9 (64)**
	wdc12	120.2 (128)	**102.2 (128)**	-	-
sssp	gsh15	2.9 (128)	**1.9 (128)**	2.8 (64)	**2.1 (64)**
	clueweb12	8.1 (128)	**4.6 (128)**	5.1 (32)	**4.0 (32)**
	uk14	16.3 (128)	**13.0 (128)**	12.4 (64)	**9.0 (64)**
	wdc14	10.9 (128)	**7.7 (128)**	10.1 (64)	**5.9 (64)**
	wdc12	168.3 (128)	**78.9 (128)**	-	-

5.4.5 Summary and Discussion

Table 5.5 compares the performance of Gluon-Sync and Gluon-Async using the best-performing number of CPUs and GPUs. Both Gluon-Sync and Gluon-Async mostly scale well, so their best performance is usually on the maximum number of CPUs or GPUs we evaluated. For low-diameter graphs, Gluon-Async and Gluon-Sync are comparable. For high-diameter graphs, Gluon-Async is on average $\sim$ 1.5$\times$ faster than Gluon-Sync. The speedup varies depending on the benchmark, the input, and the scale (number of devices). The speedup is typically best for high-diameter graphs at scale. This is similar to what has been observed for asynchronous execution on CPUs [134] or GPUs [13]. Thus, Gluon-Async helps scaling out large real-world graph datasets.

5.5 Related Work

Asynchronous Distributed Graph Analytics Systems. The popularity of the bulk-synchronous parallel (BSP) model [159] of computation has led to work that improves its performance by improving the underlying asynchrony and reducing the wait time. GraphLab [118] and PowerSwitch [167] systems use their gather-apply-scatter model along with distributed locking for non-blocking, asynchronous execution. None of the other systems, including Gluon-Async, use locks. Systems like KLA [78], Aspire [162], GRACE [164], Giraph++ [156], and ASYMP [62], which are based on asynchronous parallel (AP) model, avoid delaying the processing of the already arrived messages.

GiraphUC [76] proposes the barrierless asynchronous parallel (BAP) model that uses local barriers to reduce the message "staleness" and overheads due to global synchronization. While GiraphUC is lock-free and asynchronous, it blocks during synchronization until it receives the first message (from any host). Most recently, Fan et al. [60] show that the Adaptive Asynchronous Parallel (AAP) model used in their GRAPE+ system can be used to dynamically adjust the relative progress of different worker threads and reduce the stragglers and stale computations. Similarly, Groute [13] proposes an asynchronous system, but it is limited to a single node system.

Most of these existing systems either perform fine-grained synchronization or do not support general partitioning policies. None of them can be extended for vertex-cuts without significantly increasing the communication cost; i.e., some of the communication optimizations [52] would need to be dropped for such an extension (to elaborate, GRAPE+ is the only one that can support vertex-cuts without using distributed locks, but they send an updated value from a proxy directly to all the other proxies instead of reducing updated values to a master proxy and broadcasting the result to mirror proxies, resulting in more communication volume and messages). Consequently, prior asynchronous systems do not perform as well as the state-of-the-art BSP-style distributed systems [52, 173]. Moreover, none of the prior asynchronous systems can be extended trivially to support execution on multi-host multi-GPUs.

In contrast, we propose a Bulk-Asynchronous Parallel (BASP) model for both distributed CPUs and GPUs in which the threads potentially never

wait and instead continue to do local work if available without explicitly waiting for the communication from other hosts. Our redesign of reduce and broadcast communication phases enables removing synchronization while exploiting bulk-communication.

Bulk-Synchronous Distributed Graph Analytics Systems. There have been many works that support graph analytics on distributed CPUs [38, 52, 74, 122, 173] or GPUs [52, 95] in the Bulk-Synchronous Parallel (BSP) model. Our proposed approach targets wait-time reduction in graph applications by exploiting the underlying asynchrony in codes written in BSP models, and it targets distributed CPU and GPU systems.

Models for Parallel Computation. In addition to the Bulk-Synchronous Parallel (BSP) model [159], many models [49, 63, 66] have been proposed for parallel computation. These models have been used in analyzing parallel algorithms as well as in developing parallel systems. Some models like LogP [49] and Queueing Shared Memory (QSM) [66, 143] provide abstractions for parallel computation on both shared-memory and distributed-memory machines. Our Bulk-Asynchronous Parallel (BASP) model is an extension to BSP that is useful for distributed-memory computation. The BASP model is only applicable for algorithms that are robust to stale reads. We observe that many graph analytical algorithms satisfy this property and exploit that transparently in Gluon-Async using the BASP model.

Chapter 6

Conclusions and Future Work

Compiler and runtime systems are essential in enabling programmers to easily develop applications that can run on distributed and heterogeneous architectures. This thesis introduces novel ways to build compiler and runtime systems for privacy-preserving computing and sparse computing using Fully Homomorphic Encryption (FHE) and graph processing applications. It advances techniques and optimizations in compiler and runtime systems by exploiting domain-specific knowledge. This thesis also lays the foundation for new avenues of future work in FHE and graph processing systems. Section 6.1 and Section 6.2 discuss the conclusions and future work for FHE and graph processing respectively.

6.1 Fully Homomorphic Encryption (FHE)

I believe that with continued cryptographic innovations, the task of developing practical and efficient FHE applications is a "systems and compiler" problem. This thesis validates this hypothesis.

This thesis introduces a novel way to automatically support computations on encrypted data by building a language and compiler, called EVA,

that transforms the required computation to run on data encrypted using FHE. EVA is also designed for easy targeting of domain specific languages. For Deep Neural Network (DNN) inference, this thesis introduces an optimizing compiler, called CHET, that takes tensor programs (similar to that in TensorFlow or ONNX) as input and generates EVA programs that perform the required inference on encrypted data (e.g., image). While both CHET and EVA target the RNS-CKKS [41] FHE scheme, they make it easy to port the same programs to different FHE schemes — a task that will become necessary as new and improved FHE schemes are discovered. Our evaluation uses the SEAL [149] library implementation of the RNS-CKKS scheme, which is currently implemented to run on a single CPU core. In the future, FHE schemes like RNS-CKKS can be implemented to run on heterogeneous architectures like GPUs and FPGAs, and CHET and EVA can run the same programs on different architectures transparently. CHET was the first compiler for DNN inference using FHE and EVA is currently the only compiler for general-purpose computation using the RNS-CKKS FHE scheme.

In this thesis, I formulate compiling programs for FHE as constrained optimization problems and solve them by introducing novel compiler optimizations that exploit cryptographic expertise. CHET optimizes the mapping of tensor operations to the FHE-supported vector instructions by systematically exploring many more optimization choices than manually feasible. EVA automatically inserts FHE-specific instructions optimally. CHET and EVA exploit the semantics of FHE instructions to analyze the program and determine the

encryption parameters for the program to be correct, secure, and performant. CHET and EVA can optimize and target more difficult to use (or compile) but more efficient FHE schemes like RNS-CKKS [41], whereas even FHE experts implement non-trivial applications like DNN inference only for more easier to use but less efficient FHE schemes like CKKS [42] or BFV [59]. Therefore, the compiler-generated code for DNN inference outperforms highly tuned expert-written implementations by more than an order of magnitude. Furthermore, CHET and EVA compilers enable inference of deeper DNNs than was viable with experts programming before.

CHET and EVA provide a solid foundation for a richer variety of FHE applications as well as domain-specific compilers and auto-vectorizing compilers for computing on encrypted data. There are still many open problems in compiling programs for FHE. For example, most FHE schemes support only multiplication and addition instructions, so FHE programmers currently need to approximate operations like ReLU or square root by using polynomial functions. Moreover, FHE schemes like CKKS and RNS-CKKS perform approximate multiplication and addition. Compilers that automatically approximate computation and define approximation semantics can help ease the burden on programmers. Another example is that the CHET runtime currently only supports two data layouts because the tensor kernels are manually written. Automatically generating kernels for more data layouts can help improve the performance of CHET further. Solutions to these problems can be built on top of CHET and EVA.

6.2 Graph Processing

This thesis demonstrates that graph analytics applications scale well on distributed and heterogeneous architectures with very little programming effort. The key to this is Gluon(-Async), a communication-optimizing substrate for distributed graph analytics that supports heterogeneity in programming models, partitioning policies, and processor types. This thesis introduces a novel way to build distributed and heterogeneous graph analytics systems out of plug-and-play components. Gluon(-Async) can be plugged into any existing shared-memory CPU or GPU graph analytics system to scale that system on distributed clusters of CPUs or GPUs respectively. Gluon(-Async) enabled building one of the first distributed GPU graph analytics systems. This Gluon(-Async)-based system is currently the only graph analytics system that supports asynchronous execution on distributed GPUs.

In this thesis, I present novel communication optimizations that exploit invariants in the graph partitioning policy at runtime. Gluon supports arbitrary graph partitioning policies while exploiting the structural invariants of a given partitioning policy at runtime to optimize communication. Gluon also optimizes communication by exploiting the temporal invariant that the partitioning of the graph does not change during the iterative computation in analytics. In addition, this thesis presents a novel programming model called *bulk*-asynchronous parallel (BASP) that takes bulk-communication from BSP models and continuous compute from asynchronous models to improve overall runtime of programs. Gluon-Async enables BASP-style execution for

graph analytics applications that are amenable for asynchronous execution. All these optimizations enable modular graph analytics systems built using Gluon(-Async) to outperform integrated distributed CPU-only and GPU-only graph analytics systems by more than an order of magnitude on distributed clusters of 128 CPUs and 64 GPUs respectively.

This thesis forms the foundation for a plethora of work on distributed and heterogeneous graph analytics. In subsequent work, a compiler called Abelian [68] automatically translates graph analytics code for a shared-memory CPU to equivalent code for a single GPU and generates the required Gluon code to communicate between CPUs and GPUs. Thus, Abelian and Gluon enable portability and performance for graph analytics on distributed and heterogeneous architectures. Phoenix [54] extends Gluon to make it resilient to fail-stop faults by exploiting properties of graph analytics algorithms. Subsequent work builds a fast graph partitioner called CuSP [85] and studies massive graph analytics using Gluon-based systems on distributed CPUs [70], distributed GPUs [94], and Intel Optane DC Persistent Memory [69]. MRBC [86] implements a new algorithm for betweenness centrality using Gluon and scales well on distributed CPUs. DistTC [84] extends Gluon to support edge proxies along with vertex proxies to implement a distributed triangle counting algorithm that scales well on distributed GPUs. Recent work [71] implements semi-supervised training of word embeddings (using the Skip-Gram model) as a graph analytics application in a Gluon-based system to scale the training on distributed CPUs without losing accuracy.

Gluon(-Async) lays the groundwork for compiler and runtime systems that support more complex sparse computing applications on distributed and heterogeneous architectures. Sparse computing applications such as graph convolutional networks, graph embeddings, sparse matrix applications, and numerical simulations on unstructured grids involve iterative computation of vertex or edge labels similar to that in graph analytics applications, but they may also involve iterative computation of other shared data. Compiler and runtime systems for such applications can be built on top of Gluon(-Async). Gluon(-Async) can also be extended or used in building compiler and runtime systems for graph mining [39] on distributed and heterogeneous architectures.

Bibliography

[1] Graph 500 benchmarks. http://www.graph500.org, 2017.

[2] The Galois system. http://iss.ices.utexas.edu/?p=projects/galois, 2018.

[3] The Lonestar benchmark suite. http://iss.ices.utexas.edu/?p=projects/galois/lonestar, 2018.

[4] Pittsburgh Supercomputing Center (PSC). https://www.psc.edu/, 2018.

[5] Texas Advanced Computing Center (TACC), The University of Texas at Austin. https://www.tacc.utexas.edu/, 2018.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[7] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *CoRR*, abs/1704.03578, 2017.

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

[10] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, pages 57–68, New York, NY, USA, 2019. ACM.

[11] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *arXiv preprint arXiv:1508.06574*, 2015.

[12] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 423–442. Springer, 2017.

[13] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 235–248, New York, NY, USA, 2017. ACM.

[14] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. Optimization of bootstrapping in circuits. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2423–2433. SIAM, 2017.

[15] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA., 2007.

[16] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[17] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019.

[18] P. Boldi and S. Vigna. The WebGraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[19] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 227–228, New York, NY, USA, 2014. ACM.

[20] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 587–596, New York, NY, USA, 2011. ACM.

[21] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

[22] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme.

In Martijn Stam, editor, *Cryptography and Coding*, pages 45–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[23] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465, New York, NY, USA, 2014. ACM.

[24] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 483–512, Cham, 2018. Springer International Publishing.

[25] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[26] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[27] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In Kamalika Chaudhuri and Ruslan Salakhut-

dinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2019.

[28] Aydin Buluc and John R Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[29] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151, 2012.

[30] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. A multi-start heuristic for multiplicative depth minimization of boolean circuits. In Ljiljana Brankovic, Joe Ryan, and William F. Smyth, editors, *Combinatorial Algorithms*, pages 275–286. Springer International Publishing, 2018.

[31] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, SCC '15, pages 13–19, New York, NY, USA, 2015. ACM.

[32] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, February 2010.

[33] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance

Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, page 35, 2017.

[34] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, pages 442–446.

[35] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *IEEE European Symposium on Security and Privacy, EuroS&P*, 2019.

[36] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. http://homomorphicencryption.org/white_papers/security_homomorphic_encryption_white_paper.pdf.

[37] Hao Chen. Optimizing relinearization in circuits for homomorphic encryption. *CoRR*, abs/1711.06319, 2017. https://arxiv.org/abs/1711.06319.

[38] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[39] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. volume 13 of *PVLDB*, 2020.

[40] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 360–384, Cham, 2018. Springer International Publishing.

[41] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr:, editors, *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Calgary, AB, Canada, August 15–17, 2019. Springer, Heidelberg, Germany.

[42] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[43] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Fal-

con: A graph manipulation language for heterogeneous systems. *TACO*, 12(4):54, 2016.

[44] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018. https://eprint.iacr.org/2018/421.

[45] Cingulata. https://github.com/CEA-LIST/Cingulata, 2018.

[46] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[47] David Corvoysier. Squeezenet for CIFAR-10. https://github.com/kaizouman/tensorsandbox/tree/master/cifar10/models/squeeze, 2017.

[48] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1020–1037, New York, NY, USA, 2018. Association for Computing Machinery.

[49] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[50] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[51] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. A Lightweight Communication Runtime for Distributed Graph Analytics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[52] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 752–768, New York, NY, USA, 2018. ACM.

[53] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Vishwesh Jatala, V. Krishna Nandivada, Marc Snir, and Keshav Pingali. Gluon-Async: A Bulk-Asynchronous System for Distributed and Hetero-

geneous Graph Analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, PACT '19. IEEE, 2019.

[54] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 615–630, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, New York, NY, USA, 2020. ACM.

[56] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 142–156, New York, NY, USA, 2019. ACM.

[57] Yao Dong, Ana Milanova, and Julian Dolby. Jcrypt: Towards computation over encrypted data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 8:1–8:12, 2016.

[58] Erich Elsen and Vishal Vaidyanathan. Vertexapi2 – a vertex-program api for large graph computations on the gpu. 2014.

[59] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[60] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. Adaptive Asynchronous Parallelization of Graph Algorithms. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1141–1156, New York, NY, USA, 2018. ACM.

[61] J. S. Firoz, M. Zalewski, A. Lumsdaine, and M. Barnas. Runtime Scheduling Policies for Distributed Graph Algorithms. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 640–649, May 2018.

[62] Eduardo Fleury, Silvio Lattanzi, Vahab S. Mirrokni, and Bryan Perozzi. ASYMP: fault-tolerant mining of massive graphs. *CoRR*, abs/1712.09731, 2017.

[63] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, NY, USA, 1978. Association for Computing Machinery.

[64] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[65] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354. ACM, 2012.

[66] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, page 72–83, New York, NY, USA, 1997. Association for Computing Machinery.

[67] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 201–210, 2016.

[68] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 249–264, Cham, 2018. Springer International Publishing.

[69] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. volume 13 of *PVLDB*, 2020.

[70] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms. volume 12 of *PVLDB*, 2018.

[71] Gurbinder Gill, Roshan Dathathri, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Distributed Training of Embeddings using Graph Analytics, 2019.

[72] Apache Giraph. http://giraph.apache.org/, 2013.

[73] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[74] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[75] Protocol buffer. https://developers.google.com/protocol-buffers. Google Inc.

[76] Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[77] W. Han, D. Mawhirter, B. Wu, and M. Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245, Sep. 2017.

[78] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 27–38, 2014.

[79] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on*

Principles and practice of parallel programming, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

[80] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[81] Helib. https://github.com/homenc/HElib, 2020.

[82] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[83] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *CoRR*, abs/1711.05189, 2017.

[84] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali. DistTC: High Performance Distributed Triangle Counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.

[85] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2019, 2019.

[86] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. A Round-Efficient Distributed Betweenness Centrality Algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 272–286, New York, NY, USA, 2019. ACM.

[87] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan. Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40, Sep. 2017.

[88] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 58:1–58:12, New York, NY, USA, 2015. ACM.

[89] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276. ACM, 2011.

[90] Imranul Hoque and Indranil Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 9:1–9:17, New York, NY, USA, 2013. ACM.

[91] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. https://arxiv.org/abs/1602.07360.

[92] Cryptography Lab in Seoul National University. Homomorphic encryption for arithmetic of approximate numbers (heaan). https://github.com/snucrypto/HEAAN.

[93] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. Graphbuilder: Scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[94] Vishwesh Jatala, Roshan Dathathri, Gurbinder Gill, Loc Hoang, V. Krishna Nandivada, and Keshav Pingali. A Study of Graph Analytics for Massive Datasets on Distributed GPUs. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2020, 2020.

[95] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11(3):297–310, November 2017.

[96] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural net-

works. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1209–1222, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[97] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, 2010.

[98] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 1651–1669, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

[99] Thejaka Amila Kanewala, Marcin Zalewski, and Andrew Lumsdaine. Families of Graph Algorithms: SSSP Case Study. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 428–441, Cham, 2017. Springer International Publishing.

[100] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[101] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automa-*

tion Conference, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM.

[102] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[103] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[104] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 447–461. ACM, 2016.

[105] Alex Krizhevsky. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

[106] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed computing: Principles, algorithms, and systems*. Cambridge University Press, 2008.

[107] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[108] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[109] Kim Laine. Simple encrypted arithmetic library (seal) manual. https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf, 2017.

[110] Monica S. Lam, Stephen Guo, and Jiwon Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 278–289, Washington, DC, USA, 2013. IEEE Computer Society.

[111] Kristin Lauter. Postquantum opportunities: Lattices, homomorphic encryption, and supersingular isogeny graphs. *IEEE Security Privacy*, 15(4):22–27, 2017.

[112] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. Data partitioning strategies for graph workloads on heterogeneous

clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 56:1–56:12, New York, NY, USA, 2015. ACM.

[113] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[114] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Commun. ACM*, 59(5):78–87, April 2016.

[115] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[116] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[117] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 619–631, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[118] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Frame-

work for Machine Learning and Data Mining in the Cloud. *Proceedings VLDB Endow.*, 5(8):716–727, April 2012.

[119] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[120] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 195–207, Santa Clara, CA, 2017. USENIX Association.

[121] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[122] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings ACM SIGMOD Intl Conf. on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[123] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[124] Mario Mendez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, October 2010.

[125] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012.

[126] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph/, 2012.

[127] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web — revisited: A trick of the heavy tail. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 427–432, New York, NY, USA, 2014. ACM.

[128] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael

Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[129] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[130] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 96–107, New York, NY, USA, 2013. ACM.

[131] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, London, UK, 2013. Springer-Verlag.

[132] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '13, New York, NY, USA, 2013. ACM.

[133] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed

shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[134] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[135] Nicholas A. Nystrom, Michael J. Levine, Ralph Z. Roskies, and J. Ray Scott. Bridges: A uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 30:1–30:8, New York, NY, USA, 2015. ACM.

[136] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[137] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[138] Palisade homomorphic encryption software library. https://palisade-crypto.org/, 2020.

[139] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-gpu graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 479–490, May 2017.

[140] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA, 2015. ACM.

[141] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[142] The Lemur Project. The ClueWeb12 Dataset. http://lemurproject.org/clueweb12/, 2013.

[143] Vijaya Ramachandran, Brian Grayson, and Michael Dahlin. Emulations between qsm, bsp, and logp: A framework for general-purpose parallel algorithm design. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, page 957–958, USA, 1999. Society for Industrial and Applied Mathematics.

[144] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, pages 707–721, Incheon, Republic of Korea, April 2–6, 2018. ACM Press.

[145] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 2:1–2:6, New York, NY, USA, 2018. ACM.

[146] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[147] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[148] Microsoft SEAL 3.1.0. https://github.com/Microsoft/SEAL, February 2019. Microsoft Research, Redmond, WA.

[149] Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL, June 2019. Microsoft Research, Redmond, WA.

[150] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[151] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 646–655, May 2017.

[152] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[153] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 15:1–15:8, New York, NY, USA, 2017. ACM.

[154] LeNet-5-like convolutional MNIST model example. https://github.com/tensorflow/models/blob/v1.9.0/tutorials/image/mnist/convolutional.py, 2016.

[155] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd D. Millstein. Mrcrypt: Static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 271–286, 2013.

[156] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[157] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. XSEDE: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, 2014.

[158] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.

[159] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[160] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan.

Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[161] Stijn Marinus Van Dongen. *Graph clustering by flow simulation.* PhD thesis, 2000.

[162] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, New York, NY, USA, 2014. ACM.

[163] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, July 2019.

[164] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[165] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, 2015.

[166] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux2: Distributed graph computation for machine learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 669–682, Boston, MA, 2017. USENIX Association.

[167] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 194–204, New York, NY, USA, 2015. ACM.

[168] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[169] Fan Yang, Ming Wu, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15. ACM – Association for Computing Machinery, August 2015.

[170] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Sci-*

ence, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[171] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[172] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6), 2014.

[173] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association.

[174] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.