UNIVERSITY OF CALIFORNIA
RIVERSIDE

Architectural-Aware Performance Optimization: From the Foundational Math Library to Cutting-Edge Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yujia Zhai

June 2023

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Rajiv Gupta
Dr. Daniel Wong
Dr. Zhjia Zhao

The Dissertation of Yujia Zhai is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The past five years have concluded a winding path toward my doctoral degree. It has taken me through organic and computational chemistry, and finally to computer science. I would first express my appreciation for all the ups and downs, the joys and sorrows in life, which together form a simple yet fascinating life. Riverside was a lovely college. It has been a memorable experience for me to study and live there for the past five years. Winston Chung Hall, Highlander Union Building, Two Trees Trail, and the Botanic Gardens — these are all the places that have become an integral part of my life and have given me memories that I will always cherish.

A massive thanks must go to my advisor, Professor Zizhong Chen. I would never be able to delve into the complex and fascinating field of computational science without his generous support and patient guidance. I would also like to express my gratitude to my dissertation committee members: Professors Rajiv Gupta, Daniel Wong, and Zhijia Zhao for their constant scientific guidance and valuable feedback.

It has been an exceptional honor to collaborate with my outstanding colleagues at Riverside. Whether we worked together in-person or remotely, our team shared positive professional and personal relationships. The atmosphere was always welcoming, and I am grateful for that. I would like to express my gratitude to Jieyang and Sihuan, who served as role models for me as a doctoral student. I am thankful to Kaiming, Kai, Jinyang, and Quan for their valuable feedback on my research projects. I would like to extend my heartfelt thanks to Elisa, whose tireless assistance improved the quality of my manuscripts. Additionally, I would like to thank Ziyang, Zizhe, Shixun, and Jiajun for the learning and

discussions in our group meetings. Together, they all contributed to the vibrant life in the SuperLab. Thank you all.

I would also like to express my sincere gratitude to my managers, mentors, and colleagues during my internships. I would like to thank Alex and Alexey at Intel for their invaluable advice, guidance, and support during and after my initial industrial internship. I would also like to extend my appreciation to Xin, Leyuan, and Yibo at ByteDance for their expertise, encouragement, and collaboration throughout the project. Lastly, I am grateful to Haicheng, Pradeep, and Alan at NVIDIA for their mentorship, insights, and unwavering support, which have been instrumental in my professional growth and success.

As a doctoral student in engineering, my focus on research and development often consumed most of my time. However, I cannot take full credit for my achievements in the field without acknowledging the unwavering support of my fiancee and parents. Your unconditional love has always been the deepmost driving force that has consistently inspired and motivated me throughout this journey.

Chapter 2, in full, has been submitted to IEEE Transactions on Parallel and Distributed Systems for publication. The preprint of this manuscript is posted on ResearchGate. A short version of the journal paper has been published in the proceedings at the 2021 International Conference on Supercomputing (ICS '21). **Yujia Zhai**, Elisabeth Giem, Quan Fan, Kai Zhao, Jinyang Liu, and Zizhong Chen. "FT-BLAS: a High-Performance BLAS Implementation with Online Fault Tolerance." In Proceedings of the ACM International Conference on Supercomputing, pp. 127-138. 2021. The dissertation author was the primary author of both the journal article [207] and the conference paper [206].

Chapter 3, in full, has been published in the proceedings at the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS '22). **Yujia Zhai**, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. "Accelerating Encrypted Computing on Intel GPUs." In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 705-716. IEEE, 2022. The dissertation author was the primary author of the conference paper [208].

Chapter 4, in full, has been accepted for publication in the proceedings at the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS '23). The preprint of this manuscript is posted on arXiv. **Yujia Zhai**, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. "ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs." arXiv preprint arXiv:2210.03052 (2022). The dissertation author was a co-primary author of this conference paper [209].

To my family for all the support.

ABSTRACT OF THE DISSERTATION

Architectural-Aware Performance Optimization: From the Foundational Math Library to Cutting-Edge Applications

by

Yujia Zhai

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2023
Dr. Zizhong Chen, Chairperson

Efficient performance is essential for deploying a system in the real world. This thesis presents techniques for optimizing performance with an awareness of architecture for applications ranging from foundational math libraries, such as Basic Linear Algebra Subprograms (BLAS), to cutting-edge applications like homomorphic encryption (HE) and deep learning (DL) inference for the transformer model.

First, we introduce FT-BLAS, a new implementation of BLAS that offers high reliability and superior performance compared to other libraries, including Intel MKL, OpenBLAS, and BLIS. FT-BLAS is capable of tolerating soft errors on-the-fly, making it more robust than other libraries. The experimental results of FT-BLAS on Intel Skylake, Intel Cascade Lake, and AMD Zen2 processors demonstrate its high performance, being up to 3.50%, 22.14%, and 21.70% faster than Intel MKL, OpenBLAS, and BLIS, respectively.

We then present XeHE, a HE library accelerated for Intel GPUs. Our staged optimizations, including low-level optimizations and kernel fusion, accelerate the Number Theoretic Transform (NTT), a fundamental algorithm for HE, by up to 9.93X compared to

the naive GPU baseline. Our optimized NTT reaches 79.8% and 85.7% of the peak performance on two GPU devices, and our systematic optimizations improve the performance of encrypted element-wise polynomial matrix multiplication applications by up to 3.11X.

Finally, we present ByteTransformer, an industrial transformer framework optimized for variable-length inputs. ByteTransformer has been deployed to serve TikTok and Douyin applications of ByteDance, and part of our proposed optimizations has been integrated into the production code base of NVIDIA. Experimental results on an NVIDIA A100 GPU with variable-length sequence inputs validate that our fused MHA outperforms the standard PyTorch MHA by 6.13x. ByteTransformer's end-to-end performance for a standard BERT Transformer model surpasses state-of-the-art transformer frameworks, such as PyTorch JIT, TensorFlow XLA, Tencent TurboTransformer, Microsoft DeepSpeed-Inference, and NVIDIA FasterTransformer, by 87%, 131%, 138%, 74%, and 55%, respectively. We also demonstrate the general applicability of our optimization methods to other BERT-like models, including ALBERT, DistilBERT, and DeBERTa.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents a series of practices in performance optimizations for software systems on modern computing platforms, such as Intel and AMD x86 CPUs, and Intel and NVIDIA GPUs. The target applications span from a foundational math library, Basic Linear Algebra Subprograms (BLAS), to cutting-edge applications including homomorphic encryption (HE) and machine learning systems.

As Moore's law comes to an end and hardware technology is approaching its limits, software-level performance optimizations become increasingly important. With hardware improvements slowing down, software-level optimizations can help to achieve better performance, energy efficiency, and cost savings. In addition, modern software systems are often complex and require significant computing power, which makes efficient utilization of hardware resources even more critical. The power wall is another issue that motivates software-level optimizations as energy consumption becomes a significant bottleneck in system performance. By optimizing software at the code level, developers can improve

application performance, reduce energy consumption, and extend the lifespan of existing hardware, providing a sustainable solution for computing systems. Overall, with the limitations of hardware technology, software-level optimizations have become an essential means to achieve better performance and efficiency.

Ever since first defined in the 1990s, the BLAS library serves as a core linear algebra library fundamental to a broad range of applications, including weather forecasting [164], deep learning [2, 144], and molecular dynamics simulation [147]. Because of this pervasive usage, academic institutions and hardware vendors provide a variety of BLAS libraries such as Intel MKL [1], AMD ACML, IBM ESSL, ATLAS [190], BLIS [181], and OpenBLAS [185] to pursue extreme performance on a variety of hardware platforms. A minor improvement in a BLAS routine can significantly impact its dependent upper-level applications.

Another application that attests to our attention is homomorphic encryption, an emerging cryptographic encryption scheme that allows computations to be performed directly on encrypted messages without the need for decryption. This encryption scheme, thus, protects private data from both internal malicious actors and external intruders, while assuming honest computations. This attractive feature, however, adds significant computations to ordinary encryption schemes. To address the memory and runtime overhead of HE — a major obstacle to immediate real-world deployments, HE libraries support efficient implementations of multiple HE schemes, including Microsoft SEAL [108] (BFV/CKKS), HElib [80] (BFV/BGV/CKKS), and PALISADE [148] (BGV/BFV/CKKS/TFHE). In [18], Intel published HEXL, accelerating HE integer arithmetic on finite fields by featuring Intel Advanced Vector Extensions 512® (Intel AVX512) instructions. Since GPUs deliver higher

memory bandwidth and computing throughput with lower normalized power consumption, researchers presented libraries such as cuHE [53], TFHE [47] and NuFHE [129] to accelerate HE using CUDA-enabled GPUs.

We also focus on accelerating machine learning systems. The last decade has witnessed rapid developments in natural language processing (NLP) pre-training models based on the transformer model, such as Seq2seq [182], GPT-2 [150], XLNET [201] and ChatGPT [141]. BERT-like models consume increasingly larger parameter space and correspondingly more computational resources. When BERT was discovered, a large model required 340 million parameters [205], but currently, a full GPT-3 model requires 170 billion parameters [22]. The base BERT model requires 6.9 billion floating-point operations to infer a 40-word sentence, and this number increases to 20 billion when translating a 20-word sentence using a base Seq2Seq model [63]. The size of the parameter space and the computational demands increase the cost of the training and inference for BERT-like models, which requires the attention of the DL community in order to accelerate these models.

In this thesis, we leverage architectural-aware performance optimizations to accelerate software systems in the three aspects above. To be more specific, our contributions include:

- We present FT-BLAS, a new implementation of BLAS routines that not only tolerates soft errors on the fly but also provides comparable performance to modern state-of-the-art BLAS libraries on widely-used processors such as Intel Skylake and Cascade Lake. To accommodate the features of BLAS, which contains both memory-bound and computing-bound routines, we propose a hybrid strategy to incorporate fault tolerance

into our brand-new BLAS implementation: duplicating computing instructions for memory-bound Level-1 and Level-2 BLAS routines and incorporating an Algorithm-Based Fault Tolerance mechanism for computing-bound Level-3 BLAS routines. Our high performance and low overhead are obtained from delicate assembly-level optimization and a kernel-fusion approach to the computing kernels. Experimental results demonstrate that FT-BLAS offers high reliability and high performance – faster than Intel MKL, OpenBLAS, and BLIS by up to 3.50%, 22.14%, and 21.70%, respectively, for routines spanning all three levels of BLAS we benchmarked, even under hundreds of errors injected per minute.

- We present XeHE, a software framework that accelerates privacy-preserved computations on Intel GPUs. XeHE provides the first-ever GPU backend for the Microsoft SEAL library. We perform optimizations from the instruction level, algorithmic level, and application level to accelerate our HE library based on the Cheon, Kim, Kim and Song (CKKS) scheme on Intel GPUs. The performance is validated on two latest Intel GPUs. Experimental results show that our staged optimizations, including low-level optimizations and kernel fusion, accelerate the Number Theoretic Transform (NTT), a key algorithm for HE, by up to 9.93X compared with the naive GPU baseline. The roofline analysis confirms that our optimized NTT reaches 79.8% and 85.7% of the peak performance on two GPU devices. Through the highly optimized NTT and the assembly-level optimization, we obtain 2.32X - 3.05X acceleration for HE evaluation routines. In addition, our all-together systematic optimizations improve the performance of encrypted element-wise polynomial matrix.

- We present ByteTransformer, a high-performance transformer boosted for variable-length inputs. We propose a padding-free algorithm that liberates the entire transformer from redundant computations on zero-padded tokens. In addition to algorithmic-level optimization, we provide architecture-aware optimizations for transformer functional modules, especially the performance-critical algorithm Multi-Head Attention (MHA). Experimental results on an NVIDIA A100 GPU with variable-length sequence inputs validate that our fused MHA outperforms PyTorch by 6.13x. The end-to-end performance of ByteTransformer for a forward BERT transformer surpasses state-of-the-art transformer frameworks, such as PyTorch JIT, TensorFlow XLA, Tencent TurboTransformer, Microsoft DeepSpeed-Inference, and NVIDIA FasterTransformer, by 87%, 131%, 138%, 74%, and 55%, respectively. We also demonstrate the general applicability of our optimization methods to other BERT-like models, including ALBERT, DistilBERT, and DeBERTa.

Chapter 2

FT-BLAS: A Fault-Tolerant High-Performance BLAS Implementation on x86 CPUs

2.1 Introduction

Due to common performance-enhancing technologies such as shrinking transistor width, higher circuit density, and lower near-threshold voltage operations, processor chips are more susceptible to transient faults than ever before [110, 122, 127]. Transient faults can alter a signal transfer or corrupt the bits within stored values instead of causing permanent physical damage [71, 112]. As a consequence, reliability has been identified by the U.S. Department of Energy as one of the major challenges for exascale computing [120].

The academic and industry communities have observed a significant effects of transient faults, since the first transient error and resulting soft data corruption was observed by Intel Corporation in 1978 [124]. Sun Microsystems reported in 2000 that server crashes caused by cosmic ray strikes on unprotected caches were responsible for the outages of random customer sites including America Online, eBay, and others [14]. In 2003, Virginia Tech demolished the newly-built Big Mac cluster of 1100 Apple Power Mac G5 computers into individual components and sold them online because the cluster was not protected by error correcting code (ECC) and fell prey to cosmic ray-induced partial strikes, causing repeated crashes and rendering it unusable [65]. Transient faults can still threaten system reliability even if a cluster is protected by ECC: Oliveira et al. simulated an exascale machine with 190,000 cutting-edge Xeon Phi processors, which could still experience daily transient errors under ECC protection [140].

Transient faults can be grouped into two categories according to the outcome. If an affected application crashes when a transient fault occurs, it is a fail-stop error. If the affected application continues but produces incorrect results, it is called a fail-continue error. Fail-stop errors can often be protected by checkpoint/restart mechanisms (C/R) [2,144,147, 177] and algorithmic approaches [36, 39, 79]. Fail-continue errors are often more dangerous because they can corrupt application states without any warning from the system and lead to incorrect computing results [24, 45, 58, 125, 170], which can be catastrophic under safety-critical scenarios [113]. In this chapter, we restrict our scope to fail-continue errors (*soft errors*) from computing logic units (e.g., 1+1=3), assuming fail-stop errors are protected by checkpoint/restart and memory errors are protected by ECC.

Dual modular redundancy (DMR) is an approach to handle soft errors. Typically assisted by compilers, DMR duplicates computing instructions and inserts check instructions into the original programs [35,137,138,156,203]. DMR is very general and can be applied to any application, but it introduces a high overhead especially for computing-bound applications because it duplicates all computations. To reduce fault tolerance overhead, algorithm-based fault tolerance (ABFT) schemes have been developed for many applications in recent years. Huang and Abraham proposed the first ABFT scheme for matrix-matrix multiplication [90]. Sloan et al. proposed an algorithmic scheme to protect conjugate gradient algorithms for sparse linear systems [165]. Sao and Vuduc explored a self-stabilizing FT scheme for iterative methods [161]. Di and Cappello proposed an adaptive impact-driven FT approach to correct errors for a series of real-world HPC applications [56]. Chien at al. proposed the Global View Resilience system, a library that enables applications to add resilience efficiently [46]. Many other FT schemes have been developed for widely-used algorithms such as sorting [114], fast Fourier transforms (FFT) [10,115,179], iterative solvers [32,37,178], and convolutional neural networks [210]. Recently, the interplay among resilience, power, and performance has been studied [174, 175, 204], revealing the strong correlation among these key factors in HPC.

Although numerous efforts have been made to protect scientific applications from soft errors, most routines in the Basic Linear Algebra Subprograms (BLAS) library remain unprotected. The BLAS library is a core linear algebra library fundamental to a broad range of applications, including weather forecasting [164], deep learning [2, 144], and molecular dynamics simulations [147]. Because of this pervasive usage, academic institutions and

hardware vendors provide a variety of BLAS libraries such as Intel MKL [1], AMD ACML, IBM ESSL, ATLAS [190], BLIS [181], and OpenBLAS [185] to pursue extreme performance on a variety of hardware platforms. BLAS routines are organized into three levels: Level-1 (vector/vector), Level-2 (matrix/vector), and Level-3 (matrix/matrix). [191]. Except for the general matrix-matrix multiplication (GEMM) routine, which has been extensively studied [38,78,90,169,194], minimal research has concentrated on protecting the rest of the BLAS routines.

For the general matrix-matrix multiplication routine, several fault tolerance schemes have been proposed to tolerate soft errors with low overhead [78,90,169,194]. The schemes in [90] and [78] are much more efficient than DMR. However, these two schemes are offline schemes which cannot correct errors in the middle of the computation in a timely manner. In [194], Wu et al. implemented a fault tolerant GEMM that corrects soft errors online. However, built on third-party BLAS libraries, this ABFT scheme becomes less efficient when using AVX-512-enabled processors because the current gap between computation and memory transfer speed becomes so large that the added memory-bound ABFT checksum computation is no longer negligible relative to the original computing-bound GEMM routine. In [169], Smith et al. proposed a fused ABFT scheme for BLIS GEMM at the assembly level to reduce the overhead for checksum calculations. An in-memory checkpoint/rollback scheme is used to correct multiple simultaneous errors online. Although this scheme provides wider error coverage, it presents a moderate overhead "in the range of 10%" [169].

When projecting a BLAS implementation to real-world deployment, enabling support for parallel multi-core systems, as well as for a variety of mainstream micro-architectures,

such as AVX-512 and AVX2 extensions, can both be crucial. Compared with AVX-512-enabled processors, an AVX2-enabled-only processor exposes halved vectorized registers to a user, and, consequently, significantly higher register pressure when designing performance-oriented fault-tolerant algorithms. In addition to providing delicate assembly-level optimizations on computing kernels, one should propose a cache-friendly design for parallel Level-3 BLAS routines [168].

In this chapter, we develop FT-BLAS—the first BLAS implementation that not only corrects soft errors online, but also provides at least comparable performance to modern state-of-the-art BLAS libraries such as Intel MKL, OpenBLAS, and BLIS. Our FT-BLAS provides superior performance and maintains negligible overhead on both AVX-512 and AVX-2-enabled x86 processors with multi-thread support. FT-BLAS not only protects the general matrix-matrix multiplication routine GEMM, but also protects other Level-1, Level-2, and Level-3 routines. BLAS routines are widely-used in many applications from an extensive range of fields; therefore, improvements to the BLAS library will benefit not only a large number of people but also a broad cross-section of research areas. The main contributions of this chapter include:

- We develop a brand-new implementation of BLAS using AVX-512 assembly instructions that achieves comparable or better performance than the latest versions of OpenBLAS, BLIS, and MKL on AVX-512-enabled processors such as Intel Skylake and Cascade Lake.
- We benchmark our hand-tuned BLAS implementation on an Intel Skylake processor and find that it is faster than the open-source libraries OpenBLAS and BLIS by 3.85%-22.19% for DSCAL, DNRM2, DGEMV, DTRSV, and DTRSM, and comparable performance

(±1.0%) for the remaining selected routines. Compared to the closed-source Intel MKL, our implementation is faster by 3.33%-8.06% for DGEMM, DSYMM, DTRMM, DTRSM, and DTRSV, with comparable performance in the remaining benchmarks.

- We build FT-BLAS, the first fault-tolerant BLAS library, on our brand-new BLAS implementation by leveraging the hybrid features of BLAS: adopting a DMR strategy for memory-bound Level-1 and Level-2 BLAS routines and ABFT for computing-bound Level-3 BLAS routines. Our fault-tolerant mechanism is capable of not only detecting but also correcting soft errors online, during computation. Through a series of low-level optimizations, we manage to achieve a negligible (0.35%-3.10%) overhead.

- We provide multi-thread AVX-512-enabled implementations for BLAS routines (DDOT, DNRM2, DGEMV, DGEMM) and benchmark their parallel performance on an Intel Cascade Lake processor. Experimental results validate that our fault-tolerant designs maintain a negligible overhead (0.16%-3.53%), and the performance with the FT capability remains comparable to or faster than reference libraries.

- We extend FT-BLAS with AVX2-instruction support. We benchmark four representative routines (DNRM2, DGEMV, DTRSV, and DGEMM) on an AMD R7 3700X processor. Experimental results validate that FT-BLAS maintains its high performance and low overhead on this AVX2-enabled AMD processor.

- We evaluate the performance of FT-BLAS under error injection on Intel Skylake, Intel Cascade Lake, and AMD Zen2 processors. Experimental results demonstrate FT-BLAS maintains a negligible performance overhead under hundreds of errors injected per minute

while outperforming state-of-the-art BLAS implementations OpenBLAS, BLIS, and Intel MKL by up to 22.14%, 21.70%, and 3.50% respectively—all of which cannot tolerate any errors.

The rest of the chapter is organized as follows: We introduce background and related works in Section II and then detail how we achieve higher performance than the state-of-the-art BLAS libraries in Section III. Section IV and Section V present the design and optimization of our fault-tolerant schemes. Evaluation results are given in Section VI. We present our conclusions and future work in Section VII.

2.2 Related Work and Background

Algorithmic research [40,199,200], as well as architectural research, are two prominent integrals of computer science studies. In this chapter, we focus on the architectural perspective with a focus on non-communicating algorithms, though communication optimization plays a vital role in the community [88].

2.2.1 Algorithm-Based Fault Tolerance

Algorithmic approaches to soft error protection for computing-intensive or iterative applications have achieved great success [31, 37, 38, 115, 169, 193, 195, 196], ever since the first algorithmic fault tolerance scheme for matrix/matrix multiplication in 1984 [90]. The basic idea is that for a matrix-matrix multiplication $C = A \cdot B$, we first encode matrices into checksum forms. Denoting e=$[1, 1, \ldots, 1]^T$, we have $A \xrightarrow{encode} A^c := \begin{bmatrix} A \\ e^T A \end{bmatrix}$ and

$B \xrightarrow{encode} B^r := \begin{bmatrix} B & Be \end{bmatrix}$. With A^c and B^r encoded, we automatically have:

$$C^f = A^c \cdot B^r = \begin{bmatrix} C & Ce \\ e^T C & \end{bmatrix} = \begin{bmatrix} C & C^r \\ C^c & \end{bmatrix}$$

The correctness of the multiplication can be verified by checking the matrix C against C^r and C^c. Any disagreements, that is, if the difference exceeds the round-off threshold, indicate errors occurred during the computation. The cost of checksum encoding and verification is $O(n^2)$, negligible compared to the $O(n^3)$ of matrix multiplication algorithms and thus ensures lightweight soft error detection for matrix multiplication. For any arbitrary matrix multiplication algorithm, correctness can be verified at the end of the computation (offline) via the checksum relationship.

The previous ABFT scheme can be extended to outer-product matrix-matrix multiplication and the checksum relationship can be maintained during the middle of the computation:

$$C^f = \sum_s A^c(:, s) \cdot B^r(s, :) = \sum_s \begin{bmatrix} C_s & C_s e \\ e^T C_s & \end{bmatrix}$$

where s is the step size of the outer-product update on matrix C, and C_s represents the result of each step of the outer-product multiplication $A^c(:, s) \cdot B^r(s, :)$. Noting this outer-product extension, Chen et al. proposed correcting errors for GEMM online with a double-checksum scheme [38]. The offline version of the double-checksum scheme can only correct a single error in a full execution, while the online version, which corrects a single error for *each* step of the outer-product update, is able to handle multiple errors for the whole program. A checkpoint-rollback technique can also be added to overcome a many-error scenario. In [169], once errors, regardless how many, are detected via the checksum relationship, the

program restores from a recent checkpoint to correct the error. In this chapter, we target a more light-weight error model and correct one error in each verification interval using online ABFT without checkpoint/rollback for the sake of performance. This kernel fusion optimization has been widely adopted to accelerate a series of applications, such as deep learning [102], scientific computing [29], and the fault-tolerant feature [194, 197].

2.2.2 Duplication-Based Fault Tolerance

Known as dual modular redundancy (DMR), duplication-based fault tolerance is rooted in compiler-assisted approaches and has been widely studied [35, 137, 138, 156, 203]. Classified by the Sphere of Replication (SoR), that is, the logical domain of redundant execution [155], previous duplication-based fault-tolerant work can be grouped into one of three cases:

- Thread Level Duplication (TLD). This approach duplicates the entire processor and memory system: Everything is loaded twice, computed twice, and two copies are stored [137, 138].
- TLD with ECC assumption (TLD+ECC). In this approach, operands are loaded twice, but from the same memory address. All other instructions are still duplicated. [156].
- DMR only for computing errors. Only the computing instructions are duplicated and verified to prevent a faulty result from being written back to memory [35, 203].

Different SoRs target different protection purposes and error models. TLD and TLD+ECC lead to the worst performance and memory overheads, but provide the best fault coverage

without requiring any other fault-tolerance support such as checkpoint/restarting. Duplicating only the computing instructions shrinks the SoR to soft errors but almost halves the performance loss compared with TLD. We adopt the third SoR, duplication and verification of computing instructions only, in this work.

Since compiler front ends never intrude into the assembly kernels of performance-oriented BLAS libraries, in the few cases that can be found in the compiler literature relating to soft error resilience in BLAS routines [35], the performance is *never* compared against OpenBLAS or Intel MKL, but only to LAPACK [8], a reference implementation of BLAS with much slower performance on modern processors. In this work, we manually insert FT instructions into self-implemented assembly computing kernels for Level-1 and Level-2 BLAS, and then hand-tune them for highest performance.

2.3 Optimizing Level-1, Level-2, and Level-3 BLAS Routines

Before adding FT capabilities to BLAS, we first create a brand new library that provides *comparable or better* performance to modern state-of-the-art BLAS libraries. We introduce the target instruction set of our work, as well as a sketch of the overall software organization. We then dive into our detailed optimization strategies for the assembly kernel to illustrate how we push our performance from the current state-of-the-art closer to the limits of hardware.

2.3.1 Optimizing Level-1 BLAS

Level-1 BLAS contains a collection of memory-bound vector/vector dense linear algebra operations, such as vector dot products and vector scaling.

Opportunities to Optimize Level-1 BLAS

Software strategies to optimize serial Level-1 BLAS vector routines are typically no more than exploiting data-level parallelism using vectorized instructions: processing multiple packed data via a single instruction, loop unrolling to benefit pipelining and exploit instruction-level parallelism, and inserting prefetching instructions. In contrast to computing-bound Level-3 BLAS routines, where performance can reach about 90% of the theoretical limit, sequential memory-bound routines usually reach 60%-80% saturation because throughput is not high enough to hide memory latency. This fluctuating saturation range makes experimental determination of underperforming routines difficult. We therefore survey open-source BLAS library Level-1 routines source code with regard to three key optimization aspects: single-instruction multiple-data (SIMD) instruction set support, loop unrolling, and software prefetching. We include double-precision routines in Table 2.1 for analytical reference.

As seen in Table 2.1, all Level-1 OpenBLAS routines have been implemented with support for loop unrolling. We also observe the interesting fact that software prefetching, an optimization strategy as powerful as increasing SIMD width for Level-1 routines, is only adopted in legacy implementations of x86 kernels in OpenBLAS. Based on the results of this optimization survey, we optimize two representative routines: we upgrade DNRM2 with AVX-512 support and enable prefetching for DSCAL. In the evaluation section, we show that the performance of our AVX-512-enabled DNRM2 with software prefetching surpasses OpenBLAS DNRM2 (SSE+prefetching) by 17.89%, while our DSCAL with data prefetch obtains a 3.85% performance improvement over OpenBLAS DSCAL with no prefetch.

Table 2.1. Survey of Selected OpenBLAS Level-1 Routines

AVX-512/AVX2	DDOT, DSCAL, DAXPY, DROT
AVX or earlier	DNRM2, DCOPY, DROTM, IDAMAX, DSWAP
Loop Unrolling	all routines
Prefetching	DNRM2, DCOPY, DROTM, IDAMAX, DSWAP

2.3.2 Optimizing Level-2 BLAS

Level-2 BLAS performs various types of memory-bound matrix/vector operations. In contrast to Level-1 BLAS, which never re-uses data, register-level data re-use emerges in Level-2 BLAS. We choose the two most typical routines, DGEMV and DTRSV, as examples to explain the theoretical underpinnings of our Level-2 BLAS optimization strategies.

Optimizing DGEMV

DGEMV, double-precision matrix/vector multiplication, computes $y = \alpha op(A)x + \beta y$, where A is an $m \times n$ matrix and $op(A)$ can be A, A^H or A^T. The cost of vector scaling βy and $\alpha \cdot (Ax)$ is negligible compared with $A \cdot x$, therefore it suffices for us to consider $\beta = 1$, and $\alpha = 1$, and restrict our discussion to the case $y = Ax + y$, where A is an $n \times n$ square matrix. The naive implementation can be summarized as $y_i = \sum_j^n A_{ij}x_j + y_i$. Since DGEMV is a memory-bound application, the most efficient optimization strategy is to reduce unnecessary memory transfers. It is clear that the previous naive implementation requires n^2 loads for A and x and n^2 loads + stores for y. No memory transfer operations can be eliminated on matrix A because each element must be accessed at least one time.

We must focus on register-level re-use for vectors x and y to optimize DGEMV. We notice that index variable i in $A(i, j)$ is partially independent of the index j of the j-loop, and we can unroll the i-loop R_i times to exploit loading x_j into registers for re-use. Now each load of x_j is reused R_i times within a single register, so the total load operations for x improves from n^2 to n^2/R_i. In practice, R_i is typically between 2-6, because accessing too many discontinuous memory addresses increases the likelihood of translation lookaside buffer (TLB) and row buffer thrashing. We adopt R_i=4 because the longest SIMD ALU instruction (`VFMA`) latency in this loop is 4 cycles [4].

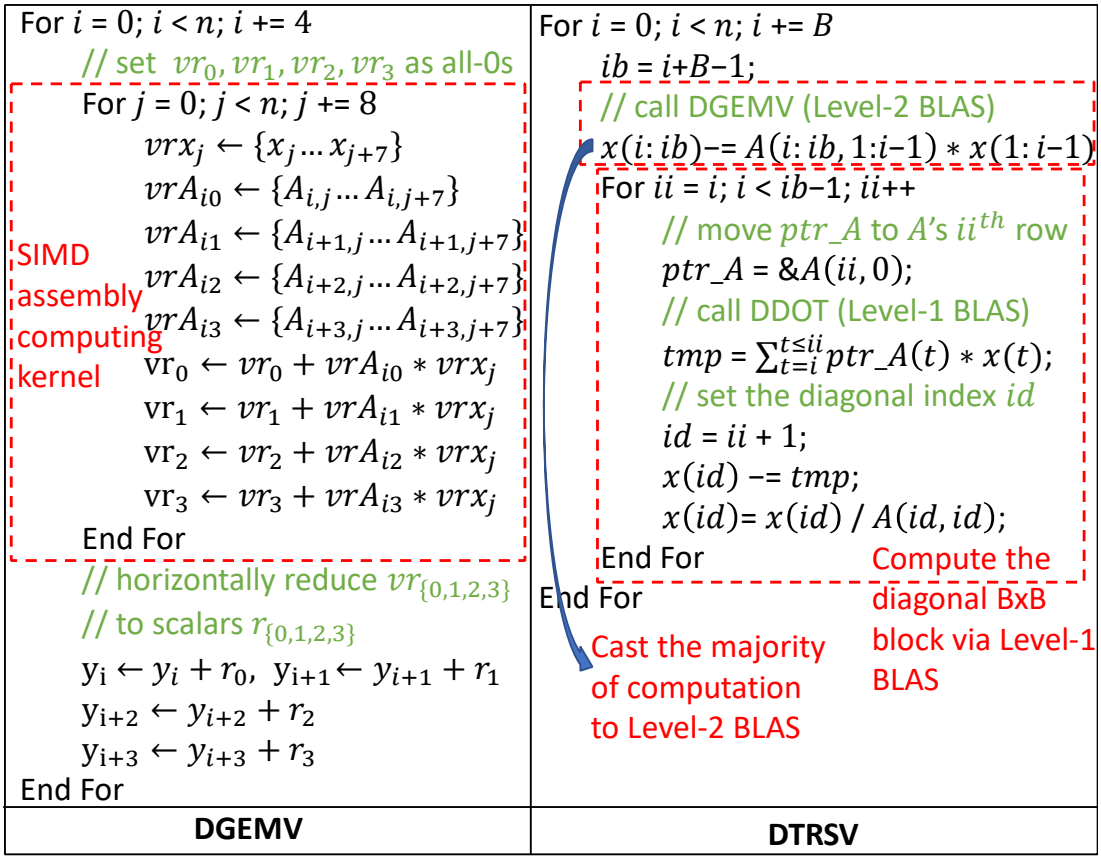


Figure 2.1: Optimization schemes of DGEMV and DTRSV.

Unrolling the inner loop (j-loop) improves nothing in terms of load/store numbers, but will benefit a SIMD implementation (vectorization). Because both an AVX-512 SIMD register and a cache line of the Skylake microarchitecture accommodate 8 doubles, we unroll the j-loop 8 times. Before entering the j-loop, four SIMD registers $vr_{\{0,1,2,3\}}$ are initialized to zero. Within the innermost loop body, each x element is still reused R_i times (shown as 4 in Figure 2.1). We load 8 consecutive x elements into a single SIMD AVX-512 register vrx_j, load the corresponding A elements into SIMD registers vrA_{i*}, and conduct vectorized fused multiplication/addition operations to update vr_*. After exiting the j-loop, vectorized registers vr_* holding temporary results are reduced horizontally to scalar registers, added onto the corresponding y_i, and stored back to memory. Some previous literature [185,191] suggests blocking for cache level re-use of vector elements. However, this may break the continuous access of the matrix elements, which is the main workload of the DGEMV computation. Hence, we do not adopt a cache blocking strategy in our DGEMV implementation: experimental results validating our DGEMV obtain a 7.13% performance improvement over OpenBLAS.

Optimizing DTRSV

Double-precision triangular matrix/vector solver (DTRSV) solves $x = op(A)^{-1}x$, where A is an $n \times n$ matrix, $op(A)$ can be A, A^H or A^T, and either the lower or upper triangular part of the matrix is used for computation due to symmetry. We restrict our discussion to $x = A^{-1}x$ using the lower triangular part of A. Since Level-2 BLAS routines are more computationally intensive than Level-1 BLAS routines, we introduce a paneling strategy for DTRSV to cast the majority of the computations — $(n^2 - nB)/2$ elements —

to the more computationally-intensive Level-2 BLAS routine DGEMV. The minor $B \times B$ diagonal section is handled with the less computationally-intensive Level-1 BLAS routine DDOT. Given that DGEMV is more efficient, adopting a smaller block size B is preferable since it allows more computations to be handled by DGEMV. Considering the practical implementation of DGEMV, where we unroll the j-loop 4 times for register re-use (shown in Figure 2.1), the minimal, and also the optimal, block size B should then be 4. In fact, OpenBLAS adopts block size B=64 for DTRSV [142], resulting in more computations handled by the less efficient diagonal routine; this is the major reason our performance exceeds that of OpenBLAS by 11.17%.

2.3.3 Optimizing Level-3 BLAS

Overview of Level-3 BLAS

Level-3 BLAS routines are matrix/matrix operations, such as dense matrix/matrix multiplication and triangular matrix solvers, where extreme cache and register level data re-use can be exploited to push the performance to the peak computation capability. We choose two representative routines, DGEMM and DTRSM to illustrate our implementation and optimization strategies for Level-3 BLAS.

Implementation of DGEMM

We adopt packing and cache blocking frames similar to OpenBLAS and BLIS. The outermost three layers of the `for` loop are partitioned to allow submatrices of A and B to reside in specific cache layers. The step sizes of these three `for` loops, M_C, N_C, and K_C, define the size and shape of the macro kernel, which are determined by the size of each

layer of the cache. A macro kernel updates an $M_C \times N_C$ submatrix of C by iterating over A ($M_R \times K_C$) multiplying B ($K_C \times N_R$) in micro kernels. Since our implementation contains no major update on the latest version of OpenBLAS other than selecting different micro kernel parameters M_R and N_R, nor on the performance ($< \pm 0.5\%$), we do not present a detailed discussion of the DGEMM implementation here but instead refer readers to [181] for more details.

Optimizing DTRSM

DTRSM, a double-precision triangular matrix/matrix solver, solves $B = \alpha \cdot op(A)^{-1}B$ or $B = \alpha B \cdot op(A)^{-1}$, where α is a double-precision scalar, A is an $n \times n$ matrix, $op(A)$ can be A, A^H, or A^T, and either the lower or upper triangular part of the matrix is used for computation due to symmetry. We restrict our discussion to $B = A^{-1}B$ in the presentation of our optimization strategy. We adopt the same cache blocking and packing scheme as DGEMM, but with the packing routine for A and the macro kernel slightly modified. For DTRSM, the packing routine for matrix A not only packs the matrix panels into continuous memory buffers to reduce TLB misses, but also stores the reciprocal of the diagonal elements during the packing to avoid expensive division operations in the performance-sensitive computing kernels. When the A_{block} to feed into the macro kernel is on the diagonal, `macro_kernel_trsm` is called to solve $B_{block} := \tilde{A}^{-1} \cdot \tilde{B}$, where $\tilde{A}$ and $\tilde{B}$ are packed matrices. Otherwise, the corresponding B_{block} is updated by calculating B_{block} -= $\tilde{A} \cdot \tilde{B}$, using the highly-optimized GEMM macro kernel. We see that the performance of the overall routine is affected by both macro kernels, and to ensure overall high performance, we must ensure the TRSM kernel is near-optimal as well.

Inside `macro_kernel_trsm`, the B_{block} is calculated by updating a small $M_R \times N_R$ B_{sub} block each time. The B_{sub} block is calculated by B_{sub} -= $\tilde{A_{curr}} \cdot \tilde{B_{block}}$ until A_{curr} reaches the diagonal block. Temporary computing results are held in registers instead of being saved to memory during computation. When A_{curr} is on the diagonal, we solve B_{sub} := $\tilde{A_{curr}}^{-1} \cdot \tilde{B_{block}}$ using an AVX-512-enabled assembly kernel. It should be noted that the packed buffer $\tilde{B}$ needs to be updated during the solve because DTRSM is an in-place update and the corresponding elements of the buffer should be updated during computation. Our highly-optimized TRSM macro kernel grants us a 22.19% overall performance gain on DTRSM over OpenBLAS, where the TRSM macro kernel is an under-optimized prototype.

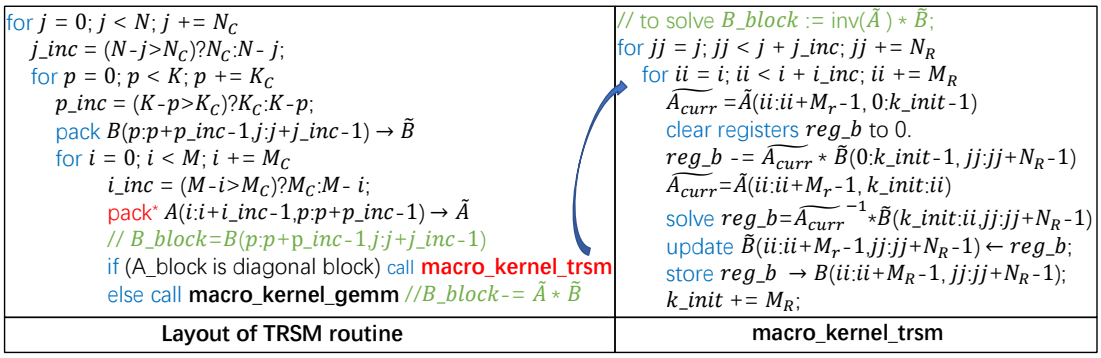


Figure 2.2: DTRSM optimization layout.

2.4 Optimizing Fault Tolerant Level-1 and Level-2 BLAS

We first outline our assembly code syntax and duplication scheme. We then show our step-wise assembly optimization of DMR decreases fault tolerance overhead from 50.8% in the scalar version to our 0.35% overhead. After the optimization, the performance of both our FT and non-FT versions surpasses both current state-of-the-art BLAS implementations.

2.4.1 Assembly Syntax and Duplication Scheme

In this chapter, all assembly examples follow AT&T syntax; that is, the destination register is in the right-most position. We adopt the most common duplication scheme, DMR [138, 156, 203]. Our chosen sphere of reduction dictates that we duplicate and verify computing instructions instead of memory instructions. More specifically, in our case, most ALU operations are floating point operations. Integer addition/subtraction are used to check whether the loop terminates. We only use two integer registers (%0, %1) throughout our assembly kernels.

2.4.2 Scalar DMR Versus Vectorized DMR

We use DSCAL, one of the most important routines in Level-1 BLAS, to show how even though DMR is labeled slow, it can actually be fast. DSCAL computes $x := \alpha \cdot x$, where x is a vector containing n DPs. DP represents a double-precision data type, so α is also a DP.

Scalar Scheme

The scalar implementation of DSCAL performs a load (`movsd`), multiplication (`mulsd`), and then a store (`movsd`) operation on scalar elements. The scalar α is invariant within the loop body, so we load it before entering the loop. The array index (stored in register %0) to access array elements is incremented by $1 before starting the next iteration. Meanwhile, register %1 (initialized by the array length n) is decremented by one to test whether the loop terminates. Once register %1 reaches zero, the EFLAG ZF is set to 1,

Table 2.2. DSCAL assembly kernel: scalar and vectorized fault tolerance schemes.

Original Scalar Instructions	Scalar FT Instructions	Original Vectorized Instructions	Vectorized FT Instructions
movsd (%2), xmm0	movsd (%2), xmm0	vbroadcastsd (%2), zmm0	vbroadcastsd (%2), zmm0
Loop:	Loop:		kxnorw, k1, k1, k1
	movsd (%3, %0, 8), xmm1	Loop:	Loop:
	movsd xmm1, xmm2	vmovupd (%3, %0, 8), zmm1	vmovupd (%3, %0, 8), zmm1
mulsd xmm0, xmm1	**mulsd xmm0, xmm1**	**vmulpd zmm0, zmm1, zmm2**	**vmulpd zmm0, zmm1, zmm2**
	mulsd xmm0, xmm2		vmulpd zmm0, zmm1, zmm3
	ucomisd xmm1, xmm2		vpcmpeqd zmm2, zmm3, k0
	jne ERROR_HANDLER		kortestw k0, k1
movsd xmm1, (%3, %0, 8)	movsd xmm1, (%3, %0, 8)		jnc ERROR_HANDLER
add $1, %0; sub $1, %1	add $1, %0; sub $1, %1	vmovupd zmm2, (%3, %0, 8)	vmovupd zmm2, (%3, %0, 8)
jnz Loop	jnz Loop	add $8, %0; sub $8, %1	add $8, %0; sub $8, %1
		jnz Loop	jnz Loop

branch instruction `jnz` will not be taken, and the loop terminates. Because scalar multiplication `mulsd` only supports a two-operand syntax—that is, `mulsd, src, dest` multiplies values from two operands and stores the result in the `dest` register—the value in the `dest` register will be overwritten when the computation finishes. Therefore, we should back up a copy of the loaded value of $x[i]$ into an unused register for use in our duplication to avoid an extra load from memory. After both the original and duplicated computations finish, we check for correctness and set the EFLAGs via `ucomsid`. If two computing results (`xmm1` and `xmm2`) are different, the EFLAG is set as ZF=1 and the branch `jne ERROR_HANDLER` will redirect the control flow to activate a resolving procedure, a self-implemented error handling assembly code. When the correctness of computing is confirmed or an erroneous result is recovered by the error handler, the result $\alpha \cdot x[i]$ is stored into memory.

AVX-512 Vectorized Scheme

Our AVX-512 vectorized duplication scheme differs from the scalar version in two ways. First, vectorized multiplication supports a three operand syntax, so source operand

registers are still live after computing and an in-register backup is no longer needed. Second, comparison between SIMD registers cannot set EFLAGs directly. Therefore, we set EFLAGs indirectly: The comparison result is first stored in an opmask register `k0`, and then `k0` is tested against another pre-initialized opmask register `k1` to set EFLAGs. If two 512-bit SIMD registers with 8 packed DPs are confirmed equal, opmask register `k0`, updated by `vpcmpegd`, will be eight consecutive '1's corresponding to the eight DPs in the comparison. If one (or more) DP(s) from two source operands in comparison are different, the corresponding bit(s) of the opmask register is set to `0`, indicating the erroneous position. We test the comparison result opmask, `k0`, with another opmask, `k1`, pre-initialized to `00000000` via `kortestw`. EFLAG is set to CF=0 first, and updated to CF=1 only if the results of OR-ing both source registers (`k0, k1`) are all '`1`'s. Any detected errors will leave CF=0, and the control flow is branched to the error handler by `jnc`.

Performance Gain Through Vectorization

Our vectorized FT enlarges the verification interval compared to the scalar implementation: The scalar scheme gives a computing/comparison+branch ratio of 1:1, while the vectorized scheme expands this ratio to 8:1, which significantly ameliorates the data hazards introduced by duplication and verification. Experimental results confirm that vectorization improves the overhead from 50.8% in the scalar scheme to 5.2% in the vectorized version.

2.4.3 Adding More Standard Optimizations

The peak single-core performance of an Intel processor that supports AVX-512 instructions is 30-120 GFLOPS, while the performance of DSCAL is less than 2 GFLOPS.

Since CPU utilization is severely bounded by memory throughput, the inserted FT instructions, which do not introduce extra memory queries, should ideally bring a near-zero overhead if computations and memory transfers are perfectly overlapped. This underutilization of CPU performance motivated us to explore optimization strategies to further bring the current 5% overhead to a negligible level.

Step 1: Loop Unrolling

Loop unrolling is a basic optimization strategy for loop-based programs. However, it can only reduce a few branch and add/sub integer instructions in practice because CPUs automatically predict branches and unroll loops via speculative execution. Possible data hazards caused by speculative execution can be ameliorated by out-of-order execution mechanisms in hardware. Experiments show that the performance of both our FT and non-FT versions only slightly improves after unrolling the loop 4 times: The overhead decreases from 5.2% to 4.9%.

Step 2: Adding Comparison Reduction

Inspired by the previous ten-fold improvement on overhead due to the enlargement of the verification interval, this optimization is naturally focused on the reduction of branch instructions for comparison and diverging to the error handler by leveraging features of the AVX-512 instruction set. Intermediate comparison results are stored in opmask registers and a correct comparison result is stored as "`11111111`" in an opmask register. Therefore, we can propagate the comparison results via `kandw k1, k2, k3`, AND-ing the two intermediate comparison results (`k1,k2`), and storing into the third opmask register `k3`. The AND

operation ensures that any detected incorrectness marked by "0" in source opmask registers will pollute bit(s) in the destination register during *reduction* and will be kept. Instead of inserting a branch to the error handler for each comparison, only one branch instruction is needed for every 4 comparisons in a loop iteration. This enlargement of the verification interval further decreases the overhead from 4.9% to 2.7%.

2.4.4 Optimizations Underrepresented in Main Libraries

At this point, we still have not reached optimality. We review possible performance concerns left from the previous step:

- Data hazards. A read-after-write hazard is a true data dependency, and severely impacts this version of the code.
- Structural hazards. Four consecutive store instructions all demand specific AVX-512 units, but there are only two in SkylakeX processors; the instructions stall until hardware becomes available.

Although out-of-order execution performed by a CPU can avoid unnecessary stalls in the pipeline stage, it consumes hardware resources and those resources are not unlimited. Therefore, we optimize instruction scheduling manually, assuming no hardware optimizations.

Heuristic Software Pipelining

We perform software pipelining to reschedule the instructions *across* the boundary of basic blocks to reduce structural and data hazards. Unfortunately, finding an optimal software pipelining scheme is NP-complete [87]. To simplify the issue, we design the software

pipelining heuristically by not considering the actual latency of each type of instruction. To scale eight consecutive elements that can be packed and processed in a AVX-512 SIMD register, we should first load them from memory (L), multiply with the scalar (M1), duplicate multiplication for verification (M2), compare between the original and duplicated results (C), and store back to memory (S) if correct. Stacking these five stages within the loop body causes a severe dependency chain because they all work on the same data stream. To deal with this issue, we first write down the required five stages for a single iteration (L, M1, M2, C, S) vertically and issue horizontally with a one-cycle latency for two adjacent instruction streams.

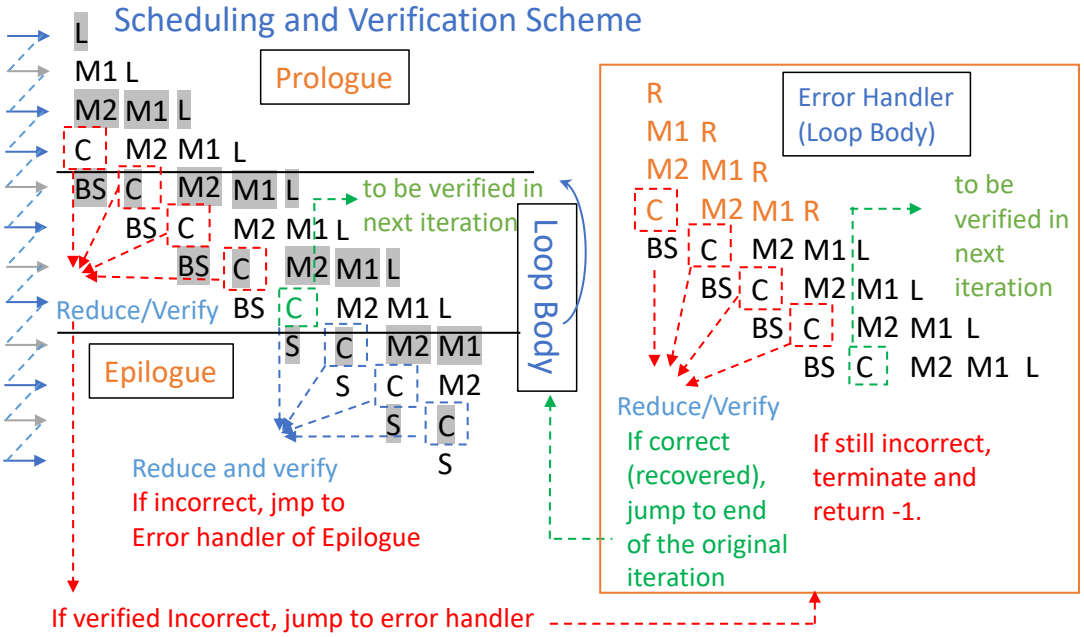


Figure 2.3: Software pipelining design.

Verification Reduction and In-Register Check-Pointing

Since the loop is still unrolled four times, comparison results can be reduced via `kandw` between opmask registers. The next loop iteration will start to execute only if the loop does not terminate and the correctness of the current iteration is verified. With cross-boundary scheduling, we compute for iterations 2, 3, 4, 5 but verify iterations 1, 2, 3, 4. The comparison result of the fifth iteration is only stored and then verified in the next iteration or in the epilogue. Because the memory is updated before the computing results are verified, we checkpoint original elements loaded from memory in an unused register. This operation coalesces the "in-register checkpoint" (B) followed by a store (S), and is denoted by BS when designing our software pipelining. Once an error is detected in the loop body and the recovery procedure is activated, the error handler restarts the computation from a couple of prologue-like instructions where the load is substituted with recovery from the backup registers. The corruption is recovered by a third calculation with duplication so the results must be verified again. If the disagreement still exists, the program is terminated and signals that it is unable to recover. If the recovered computing results reach consensus, the control flow returns back to the end of the corrupted loop iteration and continues as normal.

Effectiveness of Scheduling

Experimental benchmarks report the latencies of `vmulpd`, `vcmpeqpd`, and `vmovpd` (both store and load) are 4, 3, and 3 cycles (under a cache hit), respectively [4]. After scheduling, operands are consumed after 3 instructions; before our scheduling, these operands were consumed immediately by the following instruction. For structural hazards,

according to the Intel official development manual [50], two adjacent vectorized multiplications (M2, M1) can be executed by Port 0 and Port 1, and Port 5 accommodates the following comparison (C) simultaneously. Therefore, three consecutive ALU operations $C, M2$, and $M1$ within the loop body produce no structural hazard concerns. Additionally, Skylake processors can execute two memory operations at the same time so the structural hazard concerns on load and store are also eliminated. Therefore, we confirm that our heuristic scheduling strategy on DSCAL effectively ameliorates the hazards introduced by fault tolerance. We optimize the non-FT version using the same method and compare with our FT version. Our experimental result demonstrates that software pipelining improves FT overhead from 2.7% to 0.67%.

Adding Software Prefetching

Prefetching data into the cache before it is needed can lower cache miss rates and hide memory latency. Dense linear algebra operations demonstrate high regularity on their memory access patterns, enabling performance improvement via accurate cache prefetching. We can send a prefetch signal *before* data is needed by a *proper prefetch distance*. When the data is actually needed, it has been prefetched into cache instead of waiting the approximately 100 ns required to load it from DRAM. Accurate prefetching distance is important. If data is prefetched too early or too late, the cache is polluted and performance can degenerate. Here we select the prefetch distance to be 1024 bits: We prefetch 128 elements in advance into the L1 cache using the instruction `prefetcht0`. Instead of prefetching for all load operations, we only prefetch half of them in the loop body to avoid conflicts with hardware prefetching. Prefetching improves the performance of both

our non-FT and FT versions by ˜4%, and the overhead further decreases from 0.67% to 0.36%.

2.4.5 Enabling Parallel Support Using OpenMP

As has been discussed in Section 2.4.4, the redundant computation and verification instructions for the FT functionality lead to extra structural and data dependencies, which is the major reason of the fault-tolerant overhead for memory-bound routines. Therefore, we propose delicate assembly-level optimizations to alleviate the overhead. On a multi-core system, the massive parallelism introduced by enabling multithreading naturally reduces the cost of FT.

Most Level-1 and Level-2 BLAS routines require little communication or synchronization among threads, so one can promptly enable the parallel support for these routines by partitioning input vectors and/or input matrices when mapping workloads to physical cores. According to our evaluation, our DMR-based fault-tolerant BLAS implementations maintain negligible overhead after being threaded on Intel Cascade Lake processors.

2.4.6 Extending to AVX2-Enabled CPUs

In an AVX2-enabled processor, there are 16 256-bit SIMD registers (`ymm0` - `ymm15`), which can store 4 packed DPs, namely 4 lanes. Compared with AVX-512-enabled Intel processors, an AVX2-enabled microarchitecture possesses a shorter SIMD width and fewer SIMD registers. Since the AVX2 instruction set does not support opmask registers, we substitute the `ymm` registers for the opmask registers in AVX512 in select cases.

Table 2.3. Code snippet of the AVX2 fault-tolerant code.

Original Computation	**AVX2 Protected Computation**
	mov $15, r14d
vfmadd231pd ymm0, ymm1, ymm2	**vfmadd231pd ymm0, ymm1, ymm2**
	vfmadd231pd ymm0, ymm1, ymm3
	vpcmpeqd ymm2, ymm3, ymm4
	vmovskpd ymm4, r10d
	cmp r14d, r10d
	jne ERROR_HANDLER

Table 2.3 shows the code snippet of computation, duplication, and comparison using AVX2 instructions. We duplicate the original SIMD computation instruction and perform a lane-by-lane comparison between the duplicated and the original computational results using `vpcmpeqd`. The comparison result is stored in the SIMD register `ymm4`, which is further extracted and stored in the 32-bit general-purpose register `r10d`. If all the four DPs (lanes) of the two 256-bit SIMD registers are confirmed equal, the `r10d` register will be 4 consecutive '1's. Otherwise, a mismatch at any specific lane will set its corresponding bit in `r10d` to '0'. We test `r10d` by comparing against `r14d`, which is pre-initialized to '1111'. Any detected errors will redirect the the control flow to the error handler by `jne`. When the loop body is unrolled and there are multiple comparisons in an iteration, we store intermediate comparison results in different general purpose registers such as `r11d`, `r12d`, and `r13d`. These intermediate results, similar to when adopting opmask registers

in AVX-512, can be reduced by `and` to reduce the verification interval as well as the FT overhead.

2.5 Optimizing Fault Tolerant Level-3 BLAS

Since Level-3 BLAS routines are computing-bound routines, adopting the same DMR strategy as Level-1 and Level-2 BLAS, which doubles the computing instructions, will consequently double the performance overhead. Considering the limited registers in a single core, DMR will also increase the register pressure in the computing kernels, which will further hinder the performance. Therefore, we adopt the classic checksum-based ABFT scheme for our fault-tolerant functionality, introducing $O(n^2)$ computational overhead over the original $O(n^3)$ computation.

2.5.1 First Trial: Building Online ABFT on a Third-Party Library

Building ABFT on a third-party library is not a new topic [194]. As shown in the left side of Figure 2.4, we first encode checksums for matrices A, B, and C before starting matrix multiplication. The checksums C^c and C^r are updated asynchronously using a rank-k outer-product update of matrix C with a step size $k{=}K_c$. In every completed rank-k update, we verify the checksum relationship by first computing the reference row checksum C^r_{ref} according to the current matrix C and comparing it against C^f. If an error is detected, we continue to compute the reference column checksum C^c_{ref} and compare it against C^c to locate the erroneous row index i_{err} of C. If there is no error detected when comparing the row checksum vectors, we do not need to verify the column checksum vectors.

The total cost of the ABFT overhead consists of the initial checksum encoding, online checksum updating, and reference checksum computing–all of which are matrix-vector multiplications (DGEMV). T_{enc} includes the costs of encoding for four checksums (C^c,C^r,A^r,B^c). T_{update} includes the costs of updating on two checksums (C^c,C^r). Denoting the time of an $n \times n$ DGEMV as t_{mv}, the total cost of ABFT T_{ovhd} is

$$T_{ovhd} = T_{enc} + T_{update} + \frac{K}{Kc} \cdot (T_{C^r_{ref}} + T_{C^c_{ref}}) = (6 + \frac{2K}{Kc})t_{mv}.$$

We further denote the performance of DGEMV and DGEMM as P_{mv} and P_{mm}, both in the unit of GFLOPS. Then the total execution times of $n \times n$ DGEMM and DGEMV are T_{GEMM}=$2e^{-9}n^3/P_m$ and t_{mv}=$2e^{-9}n^2/P_{mv}$. Therefore, we have

$$\frac{T_{ovhd}}{T_{GEMM}} = \frac{(6 + \frac{2K}{Kc})t_{mv}}{2e^{-9}n^3/P_{mm}} = \frac{(6 + \frac{2K}{Kc})P_{mm}}{n \cdot P_{mv}}.$$

As shown in the above derivation, the real influence of ABFT is not simply a $O(1/n)$ computationally negligible to the baseline, but is dependent on the relative performance between the memory-speed-determined P_{mv} and the computing-capability-determined P_{mm} as well. On non-AVX-512-enabled CPUs, P_{mm}/P_{mv} ranges from 5 to 20, while on AVX-512-enabled CPUs, this ratio can be as large as 35, exaggerating the overhead up to 7-fold over old processors. The ABFT overhead reported for an older CPU [194] is around 2%, while the overhead on an AVX-512-enabled processor, measured by our benchmark in Section VI, is 15.27% — much larger than on old processors.

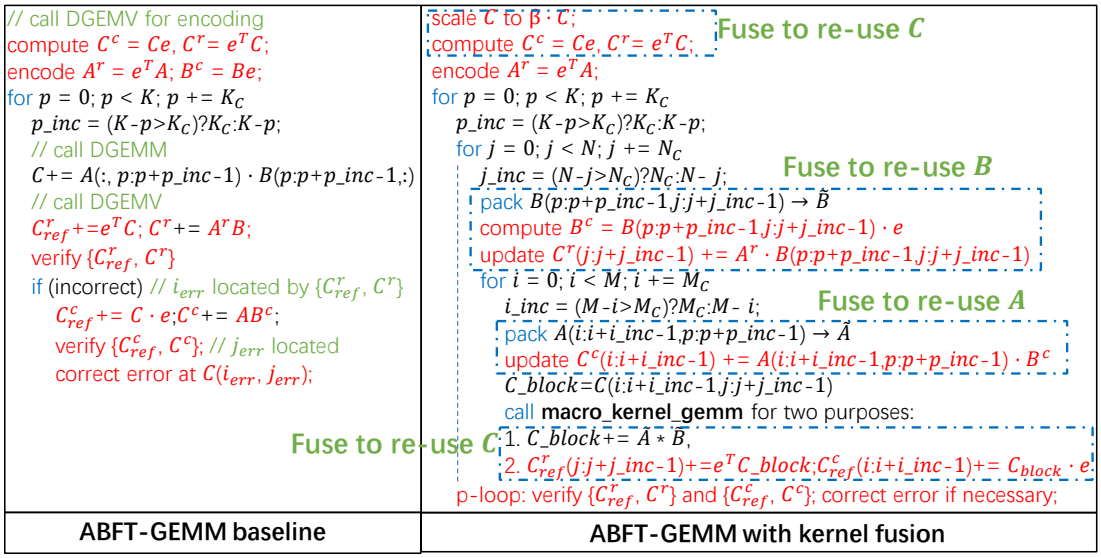


Figure 2.4: Outer-product online ABFT DGEMM optimization layout.

2.5.2 Reducing the Memory Footprint: Fusing ABFT Into DGEMM

As discussed in the previous section, the huge gap between memory transfer and floating-point computation is the reason the $O(n^2)$ checksum-related operations can no longer be amortized by $O(n^3)$ GEMM. We therefore design a fused ABFT scheme to minimize the memory footprint of checksum operations. To be more specific, the encoding of C^c and C^r is fused with the matrix scaling routine $C{=}\beta C$. When we load B to pack it to the continuous memory buffer $\tilde{B}$, checksum B^c is computed and checksum C^r is computed simultaneously by reusing B. In this fused packing routine, each B element is reused three times for each load. Similarly, each element of A loaded for packing is reused to update the column checksum C^c. In the macro kernel, which computes $C_{block}{+=}\tilde{A} \cdot \tilde{B}$, we reuse the computed C elements at register level to update the reference checksums C^r_{ref} and C^c_{ref} in order to verify the correctness of the computation. By fusing the ABFT memory footprint into DGEMM, the FT overhead decreases from 15% to 2.94%.

2.5.3 Enabling Parallel Support for ABFT Using OpenMP

In addition to providing highly efficient serial implementations, we further enable the multithreading support for DGEMM with and without fault tolerance. As discussed in Section 3.3.2, our DGEMM starts from three `for` loops allowing submatrices of A ($M_C \times K_C$) and B ($K_C \times N_C$) to reside in L2 cache and L3 cache, respectively. The cache blocking parameters M_C, K_C, and N_C are tuned to fit with the physical cache size. In practice, we set $M_C = 192, K_C = 384$, and $N_C = 9216$ for AVX-512-enabled DGEMM. Before starting the computation, both submatrices A and B are packed into continuous memory buffers, namely $\tilde{A}$ and $\tilde{B}$, to minimize TLB misses in performance-sensitive computing kernels.

On Intel Skylake and Cascade Lake server CPUs, physical cores share a large unified L3 cache while each physical core holds a smaller private L2 cache. To map this cache hierarchy in a threaded implementation, we allocate a memory buffer shared among all the threads for $\tilde{B}$, and each thread requests a private memory buffer for $\tilde{A}$. The computation workload on the C matrix is partitioned along the M-dimension. Since memory buffers $\tilde{A}$ are thread-private, each thread packs data from matrix A into their own $\tilde{A}$ buffers. When packing matrix C into the shared memory buffer $\tilde{B}$, the memory access workloads are partitioned along the N-dimension and each thread is responsible for packing a chunk of $\tilde{B}$. Just like the serial ABFT GEMM implementation, we conduct checksum encoding for the row checksum vector of A (A^r) and full checksum vectors of C (C^c, C^r). To compute the C checksums, we partition the C matrix along the M-dimension such that each thread computes a slice of the column checksum C^c while maintaining a local copy of its own row checksum vector C^r. Similarly, we partition the A matrix along the M-dimension to

compute its row checksums A^r in parallel. The checksum encoding of B^c is fused with the parallel packing operation for B to $\tilde{B}$ and at the same time, we update the reference row checksum of C. Therefore, each B element loaded from the main memory is re-used three times. Since the parallel copy operation partitions B from the N-dimension, an extra stage of reduction operation among threads is required to compute the final column checksum B^c.

2.6 Experimental Evaluation

To validate the effectiveness of our optimizations, we compare the performance of FT-BLAS with three state-of-the-art BLAS libraries: Intel oneMKL (`2020.2`, abbreviated as MKL in this Section), OpenBLAS (`0.3.13`), and BLIS (`0.8.0`), on a machine with an Intel Gold 5122 Skylake processor at 3.60 GHz, equipped with 96 GB DDR4-2666 RAM. We also compare the performance of FT-BLAS under error injection with references on an Intel Xeon W-2255 Cascade processor. This Cascade Lake machine has a 3.70 GHz base frequency and 32 GB DDR4-2933 RAM. Hardware prefetchers on both machines are enabled according to the Intel BIOS default [91]. In addition to Intel processors, we validate our AVX2 implementations on an AMD Ryzen7 3700X processor. This AMD processor has a 3.60 GHz base frequency and 32 GB DDR4-2933 RAM. We repeat each measurement twenty times and then report the average performance. For Level-1 BLAS routines, the performance is averaged from array lengths ranging from 5×10^6 to 7×10^6. For Level-2 and Level-3 BLAS routines, the performance is averaged for matrices ranging from 2048^2 to 10240^2. For the multi-threading parallel benchmark, we test the array lengths ranging

from 2×10^8 to 3×10^8 and matrices ranging from 512^2 to 20480^2. We compile the code with `icc 19.0` and the optimization flag `-O3`.

2.6.1 Performance of FT-BLAS Without FT Capability

We provide a brand-new BLAS implementation, comparable or faster than the modern state-of-the-art, before embedding FT capability. We abbreviate this BLAS implementation *FT-BLAS: Ori* in the figures.

Optimizing Level-1 BLAS

For memory-bound Level-1 BLAS, the optimization strategies employed are: 1) exploiting data-level parallelism using the latest SIMD instructions, 2) assisting pipelining by unrolling the loop, and 3) prefetching. As seen in Table 2.1, OpenBLAS has under-optimized routines, such as DSCAL and DNRM2, with respect to prefetching and AVX-512 support. We add prefetching for DSCAL, obtaining 3.85% and 5.61% speed-up over OpenBLAS and BLIS. DNRM2 is only supported with SSE2 by OpenBLAS, so our AVX-512 implementation provides a 17.89% improvement over OpenBLAS, while reaching 2.25-fold speedup on BLIS. Our implementations for both routines reach comparable performance to closed-source MKL, as seen in Figure 2.6.

Optimizing Level-2 BLAS

Register-level data re-use enters the picture in the Level-2 BLAS routine optimization. Following the optimization schemes described in Section II, we see in Figure 2.6 that our DGEMV obtains a 7.13% speed-up over OpenBLAS. This is enabled by discard-

ing cache blocking on matrix A over concerns about the potential harm of discontinuous memory accesses regarding TLB thrashing and the corresponding performance of hardware prefetchers. Because BLIS adopts the same strategy as OpenBLAS on DGEMV, our DGEMV is 6.16% faster than BLIS, while achieving nearly indistinguishable performance with MKL. For DTRSV, our strategy of minimizing the blocking parameter to cast the maximized computations to the more efficient Level-2 BLAS DGEMV grants us higher performance than all baselines, surpassing MKL, OpenBLAS, and BLIS by 3.76%, 11.17%, and 6.98%, respectively.

Optimizing Level-3 BLAS

Adopting the traditional cache blocking and packing scheme, our DGEMM performs similarly to OpenBLAS DGEMM. As seen in Figure 2.7, both of these DGEMM implementations outperform MKL and BLIS by 7.29-11.75%. For the Level-3 BLAS routine DTRSM, we provide a highly-optimized macro kernel to solve for the diagonal block and cast the majority of the computation to the near-optimal DGEMM. Because OpenBLAS and BLIS simply provide an unoptimized scalar implementation for the diagonal solver, our DTRSM outperforms OpenBLAS and BLIS by 22.19% and 24.77%, and surpasses MKL by 3.33%.

2.6.2 Performance of FT-BLAS With Fault Tolerance Capability

Having achieved comparable or better performance than the current state-of-the-art BLAS libraries without fault tolerance, we now add on fault tolerance functionalities.

For memory-bound Level-1 and Level-2 BLAS routines, we propose a novel DMR verification scheme based on the AVX-512 instruction set and then further reduce the overhead of fault tolerance to a negligible level via assembly optimization. For compute-bound Level-3 BLAS, we fuse the checksum calculations into the packing routines and assembly kernels to reduce data transfer between registers and memory. The results in this section were obtained with fault tolerant DMR and ABFT operating, but not under active fault injection—see subsection C for injection experiments.

Reducing DMR Overhead for Memory-Bound Routines

Figure 2.8 presents the performance and overhead of DSCAL with step-wise assembly level optimization. In each step, the assembly optimization described in Sections III and IV are applied to the FT version and its baseline, our non-FT version evaluated above. The performance of the most naive baseline, a scalar implementation, is 1.15 GFLOPs. Duplicating computing instructions and verifying correctness for this baseline halves the performance to 0.56 GFLOPS, bringing a 50.83% overhead. A vectorized implementation based on AVX-512 instructions decreases overhead by 9.8-fold compared to the scalar duplication/verification scheme. A vectorized implementation with fault-tolerance capability increases performance to 1.36 GFLOPs, a 2.42-fold of the scalar FT version. After this vectorization, simply unrolling the loop gains 1.55% and 1.87% improvement on the non-FT (vec-unroll-ori) and FT (vec-unroll-naive) versions respectively, while the overhead is now 4.9%. It is at this point that our non-FT version reaches OpenBLAS performance. Our novel verification scheme involving opmask registers improves the overhead to 2.7%.

We then schedule instructions via heuristic software pipelining, improving the performance of the non-FT (sp-unroll-ori) and FT (sp-unroll-FT) implementations to 1.48 and 1.47 GFLOPs respectively. The overhead improves to 0.67% in this step. We add prefetch instructions as a final step, and the overhead settles at 0.36%.

Reducing ABFT Overhead for Compute-Bound Routines

Figure 2.9 (a) presents the performance of two methods of implementing ABFT for GEMM: building upon MKL (FT-MKL) and fusing into the GEMM routine (FT-BLAS: FT fused). FT-MKL under error injection leads to 15% overhead compared with baseline MKL. When there is no error injected, we no longer compute and verify the checksum C^r so the overhead decreases to 9%. In contrast, the fused implementation (2.9% overhead) of ABFT does not generate an extra cost when encountering errors because its reference checksum computation is fused into the assembly computing kernel and is computed regardless of whether an error is detected. As shown in Figure 2.9 (b), the overhead of building ABFT on a third-party library slightly varies when linking to different libraries but the trend is clear: reference checksum construction generates the majority of the ABFT overhead, which is eliminated by the fusing strategy. The overhead can be up to 5.35-fold that of fusing ABFT into DGEMM. Our overhead is also lower than Smith et al's work in 2015 [169], where checkpoint/rollback recovery is used to tolerate errors. Their checkpoint/rollback recovery has a wider error coverage, but the overhead is "in the range of 10%" [169].

Generalizing to Other Routines

Figure 2.10 compares the performance of FT-BLAS with FT capability (FT-BLAS: FT) against its baseline: our implementation without FT capability (FT-BLAS: Ori) and reference BLAS libraries on eight routines of all three levels of BLAS. The DMR-based FT implementations for the Level-1 and Level-2 BLAS routines (DSCAL, DNRM2, DGEMV, DTRSV) generate 0.34%-3.10% overhead over the baseline. For the Level-3 BLAS routines, DGEMM, DSYMM, DTRMM, and DTRSM, our strategy to fuse memory-bound ABFT operations with matrix computation generates overhead ranging from 1.62% to 2.94% on average. Our implementation strategy for DSYMM in both FT-BLAS: Ori and FT-BLAS: FT is similar to the DGEMM scheme, with moderate modification to the packing routines. For DTRMM, we use the same strategy with some additional modifications to the computing kernel, similar to the methods in [74]. With these negligible overheads added to an already high-performance baseline, our FT-BLAS with FT capability remains comparable to or faster than the reference libraries.

Benchmarking on an Intel Cascade Lake Processor

Figure 2.11 benchmarks the performance of FT-BLAS with and without FT capability by comparing against reference BLAS libraries on an Intel Cascade Lake Xeon W-2255 processor. Similar to the results on Skylake, our baseline BLAS implementations (FT-BLAS: Ori) present comparable or better performance compared with MKL, OpenBLAS, and BLIS. The DMR-based FT implementations for memory-bound Level-1 and Level-2 BLAS routines (DSCAL, DNRM2, DGEMV, and DTRSV) add 0.06%-2.79% over-

head to the non-FT baselines. Our fused fault tolerant strategy for compute-bound Level-3 BLAS routines (DGEMM, DSYMM, DTRMM, and DTRSM) generates 1.17%-3.58% overhead on average over the baseline. With these negligible overheads added to our highly efficient non-FT baselines, our FT-BLAS maintains its performance as comparable to or faster than all of the state-of-the-art BLAS libraries.

Enabling the Parallel Support

Figure 2.12 compares the parallel performance of FT-BLAS with FT capability (FT-BLAS: FT) against its baseline: our implementation without FT capability (FT-BLAS: Ori) and reference BLAS libraries on four routines of all three levels of BLAS on an Intel Cascade Lake Xeon W-2255 processor. After enabling parallel support, the memory-bound Level-1 and Level-2 BLAS routines DDOT, DNRM2, and DGEMV, which require mostly embarrassing parallelisms, maintain a negligible overhead (0.15% - 3.53%) similar to that of serial implementations. It is worth mentioning that BLIS supports parallel implementations only for Level-3 BLAS routines, and OpenBLAS also does not provide parallel support for DNRM2. Therefore, enabling multi-threading does not increase their performance. Regarding the compute-bound Level-3 BLAS DGEMM, with our parallel design introduced in Figure 2.5, we manage to scale the performance of ABFT-DGEMM on the shared-memory multi-core platform, obtaining a scalability similar to that of OpenBLAS. With the scalable parallel design and ABFT operations fused into packing routines and assembly kernels, FT-DGEMM presents a negligible overhead (1.79%). The performance of our DGEMM implementation with FT is 16.97% faster than BLIS and comparable to OpenBLAS.

Extending to AVX2-Enabled AMD Processors

Figure 2.13 compares the performance of four routines spanning all three levels of BLAS on an AVX2-enabled AMD R7 3700X processor. Since the number of SIMD registers on AVX2 ISA is halved compared with AVX-512, we suffer from register pressure for the in-register checkpointing strategy that we have presented. Therefore, we choose to only detect errors for memory bound routines rather than correcting them online. Experimental results show that both of our DMR- and ABFT-based fault tolerant optimization strategies remain valid for AVX2 routines. With negligible overhead added to our already highly efficient non-FT baselines, our BLAS implementation with FT capability maintains its performance comparable to or faster than the reference libraries. It is worth mentioning that our AMD Zen2 CPU adopts the Uniform Memory Access (UMA) mode, or namely distributed mode, by default, which enables a single-thread application to take advantage of the entire memory bandwidth delivered by all of the memory channels. Therefore, we observe the performance of single-thread memory-bound Level-1 and Level-2 BLAS routines to be significantly faster than that on Intel processors under the NUMA mode (or local mode) [51].

2.6.3 Error Injection Experiments

We validate the effectiveness of our fault-tolerance scheme by injecting multiple computing errors into each of our computing kernels and verifying our final computation results against MKL. External error injection tools often significantly slow down the native program [77, 121, 139]. Therefore, we inject errors at the source code level to minimize the performance impact on native programs.

We inject 20 errors into each routine. The length of the injection interval k is determined based on the number of errors to inject, that is, we inject one error every k iterations. For ABFT-protected Level-3 BLAS routines, the error injection is straightforward because we can directly operate in C code. An element of matrix C is randomly selected for modification when an injection point is reached. This injected error will lead to a difference in the checksum relationship, and the erroneous element and error magnitude will be computed accordingly. This detected error is then corrected by subtracting the error magnitude from the erroneous position. For DMR-protected Level-1 and Level-2 BLAS routines, the injection is more complicated since the loop body is implemented purely using assembly codes. Therefore, providing an assembly-level error injection mechanism becomes necessary. Once the program reaches an injection point, we redirect the control flow to a faulty loop body to generate an error. This generated error is then detected via comparison with the computed results of the duplicated instruction. After the error is detected, a recovery procedure is activated to recompute the corrupted iteration immediately. In all cases we validate the correctness our final computations by comparing with MKL to ensure all injected errors were truly corrected.

Figure 2.14 compares the performance of four routines under error injection. For both DMR-protected (DGEMV, DTRSV) and ABFT-protected (DGEMM, DTRSM) routines, we maintain negligible (2.47%-3.22%) overhead, and the overall performance under error injection remains comparable or faster than reference libraries. In particular, our DTRSM outperforms OpenBLAS, BLIS, and MKL by 21.70%, 22.14%, and 3.50% even under error injection. Experimental results confirm that our protection schemes do not require

significant extra overhead to correct errors. This is because our correction methods—either to recompute the corrupted iteration or to subtract an error magnitude from the incorrect position—generate only a few ALU computations instead of expensive memory accesses.

Figure 2.15 benchmarks FT-BLAS under error injection using another processor, the Intel Cascade Lake W-2255. According to experimental results, our protection scheme is as lightweight as it was on the Skylake processor, and is still able to surpass open-source OpenBLAS and BLIS by 22.89% and 21.56% and the closed-source MKL by 4.98% even while tolerating 20 injected errors. The execution time of DTRSM and DTRSV for 512^2 to 10240^2 matrices ranges from 2 ms to 20 seconds. Therefore, injecting 20 errors into these two routines is equivalent to injecting 1 to 10,000 errors per second. Hence, FT-BLAS is able to tolerate up to thousands of errors per second with comparable and sometimes faster performance than state-of-the-art BLAS libraries—and none of them can tolerate soft errors.

Figure 2.16 compares the parallel performance of DGEMV and DGEMM under error injection on an Intel Cascade Lake W-2255 processor. When being threaded, our FT-DGEMV under error injection remains 10.91% and 13.49% faster than OpenBLAS and MKL. Compared with the non-threaded BLIS, our FT-DGEMV is 3.72X faster even under error injection. Regarding the compute-bound DGEMM, our FT-BLAS presents a performance comparable to OpenBLAS and is 16.83% faster than BLIS. Figure 2.17 further benchmarks the serial performance of these two BLAS routines under error injection on an AMD Zen2 Ryzen 3700X processor, validating that the overall performance of FT-BLAS

remains comparable to the best of the-state-of-the-art reference libraries on AVX2-enabled AMD processors.

2.7 Conclusions

We present a fault-tolerant BLAS implementation that is not only capable of tolerating soft errors, but also achieves comparable or superior performance over the current state-of-the-art libraries, OpenBLAS, BLIS, and Intel MKL. Future work will focus on extending FT-BLAS to more architectures and eventually open-sourcing the code.

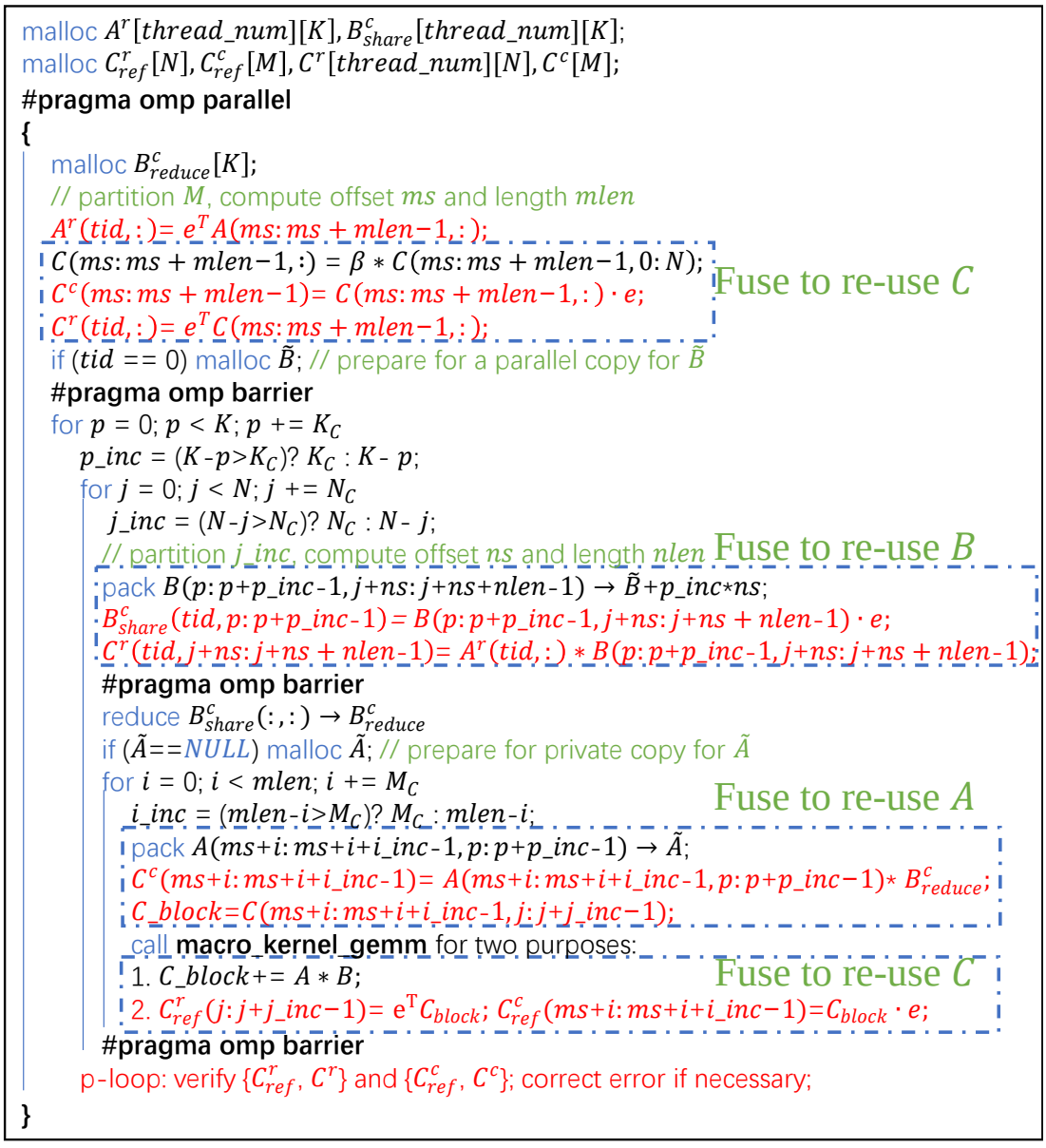
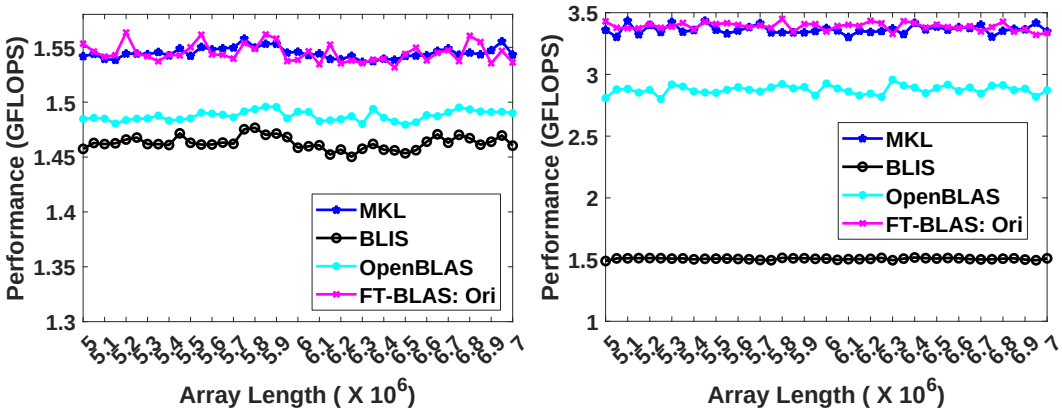
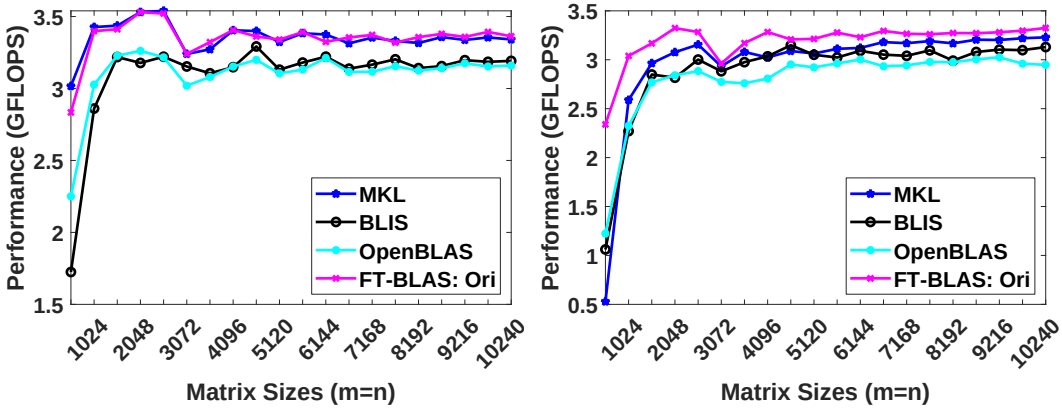


Figure 2.5: Parallel ABFT-GEMM with kernel fusion.

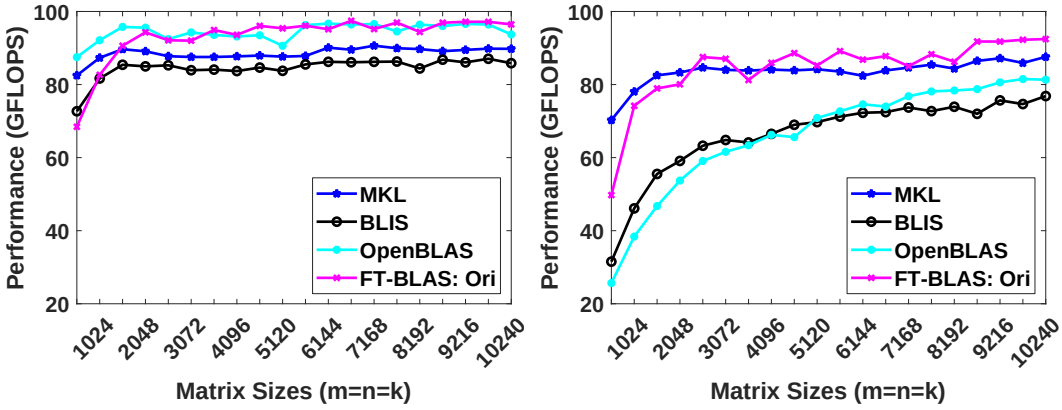


(a) DSCAL (b) DNRM2



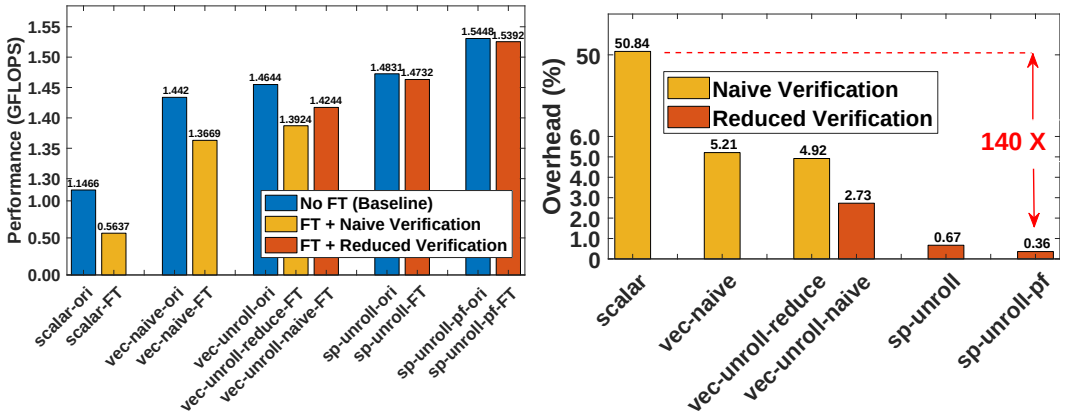
(c) DGEMV (d) DTRSV

Figure 2.6: Comparisons of selected Level-1/2 BLAS routines.



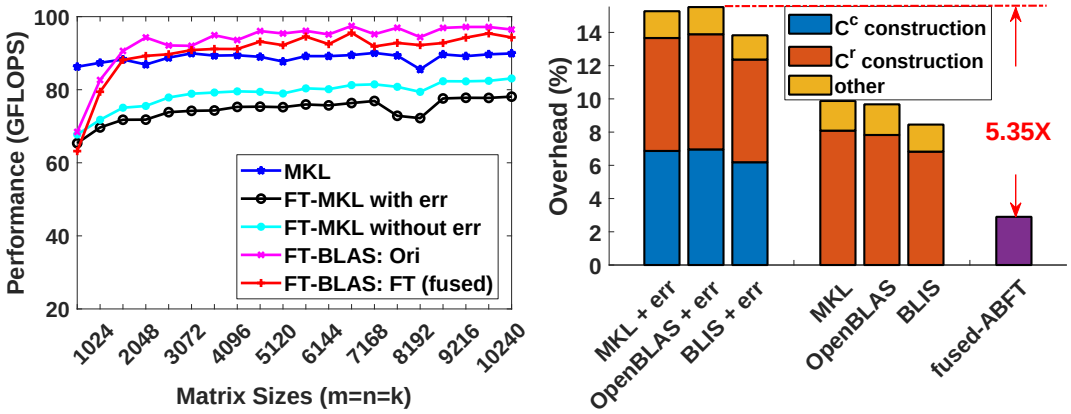
(a) DGEMM (b) DTRSM

Figure 2.7: Comparisons of selected Level-3 BLAS routines.



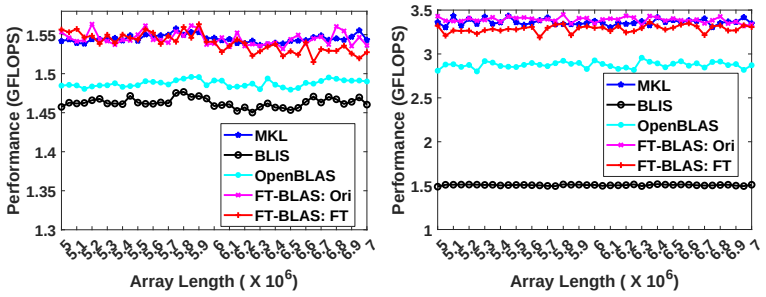
(a) Performance Optimization (b) Overhead Optimization

Figure 2.8: Optimizing DSCAL with/without FT.

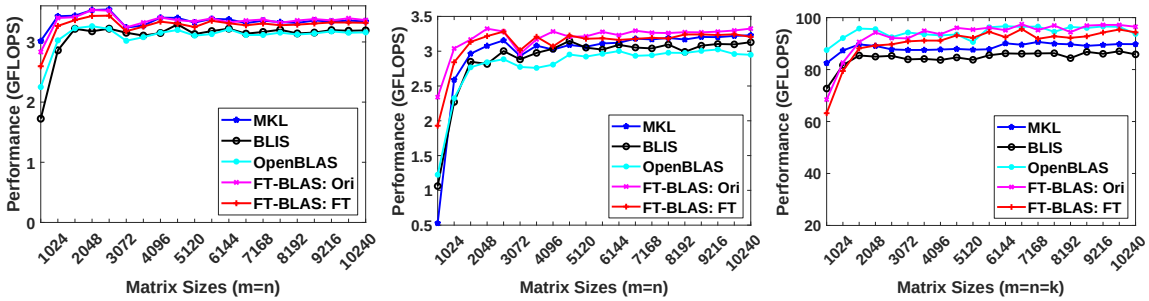


(a) Performance of FT-DGEMM (b) Linking to different libraries

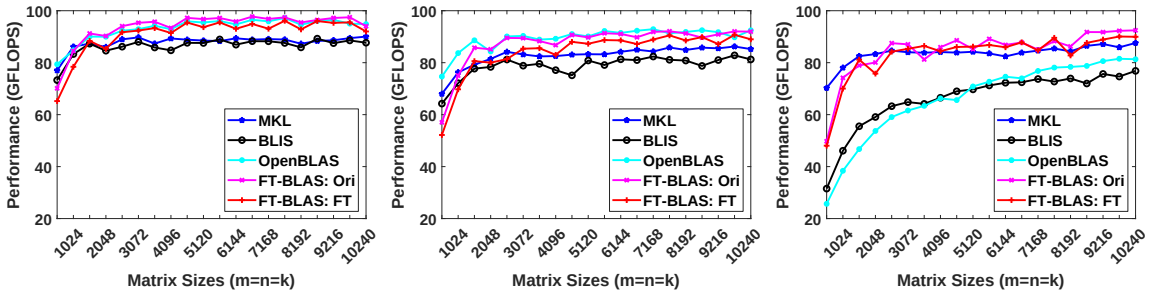
Figure 2.9: Optimizing DGEMM with FT.



(a) FT-DSCAL (b) FT-DNRM2

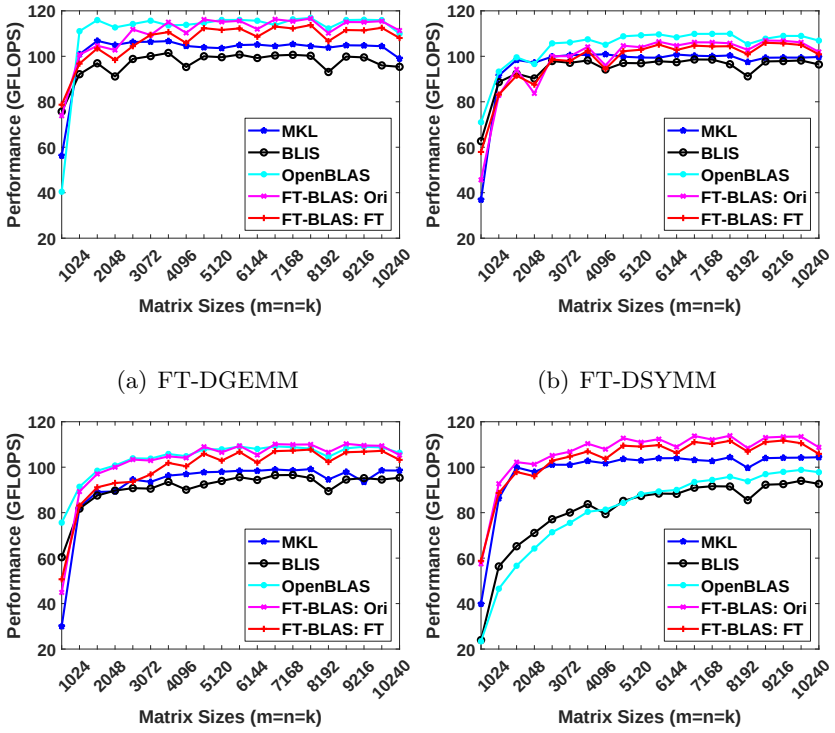


(c) FT-DGEMV (d) FT-DTRSV (e) FT-DGEMM



(f) FT-DSYMM (g) FT-DTRMM (h) FT-DTRSM

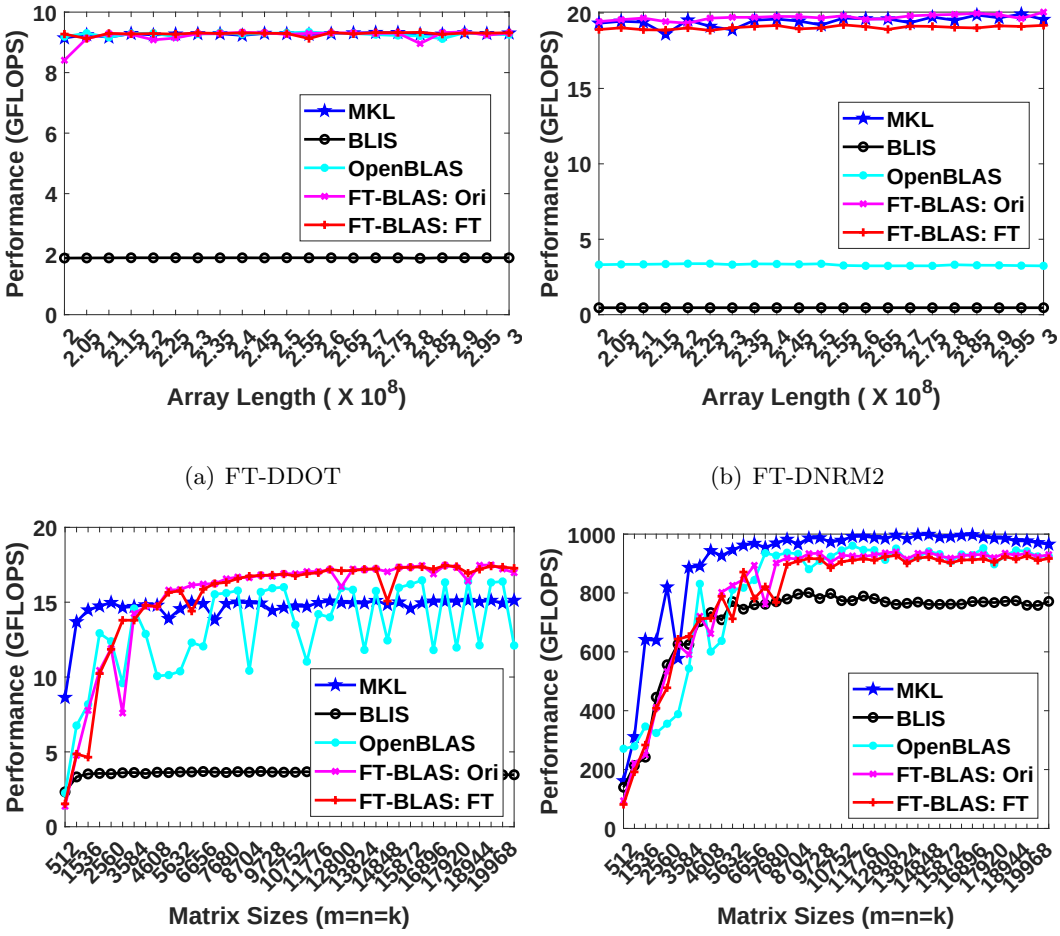
Figure 2.10: Comparisons of selected BLAS routines with FT on Skylake.



(a) FT-DGEMM (b) FT-DSYMM

(c) FT-DTRMM (d) FT-DTRSM

Figure 2.11: Comparisons of BLAS routines with FT on Cascade Lake.



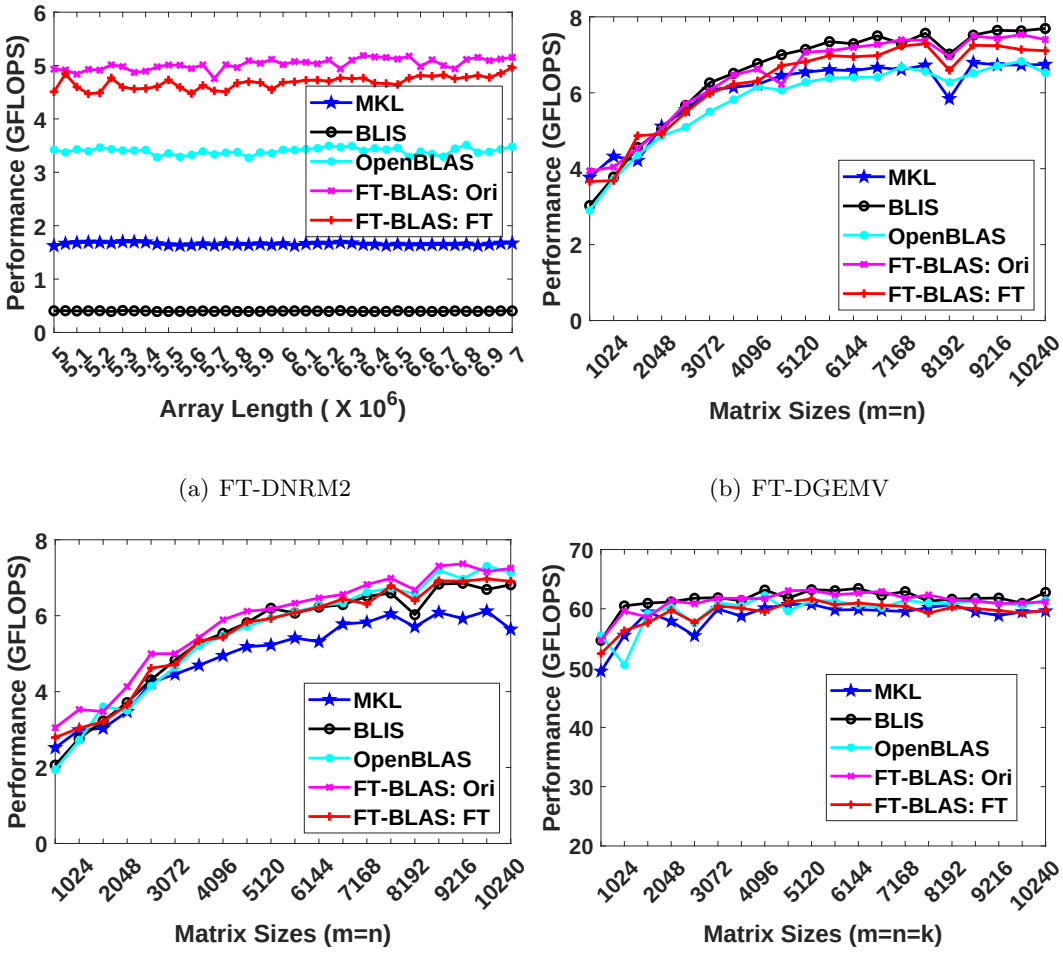
(a) FT-DDOT

(b) FT-DNRM2

(c) FT-DGEMV

(d) FT-DGEMM

Figure 2.12: Comparisons of parallel BLAS routines with FT on Cascade Lake.



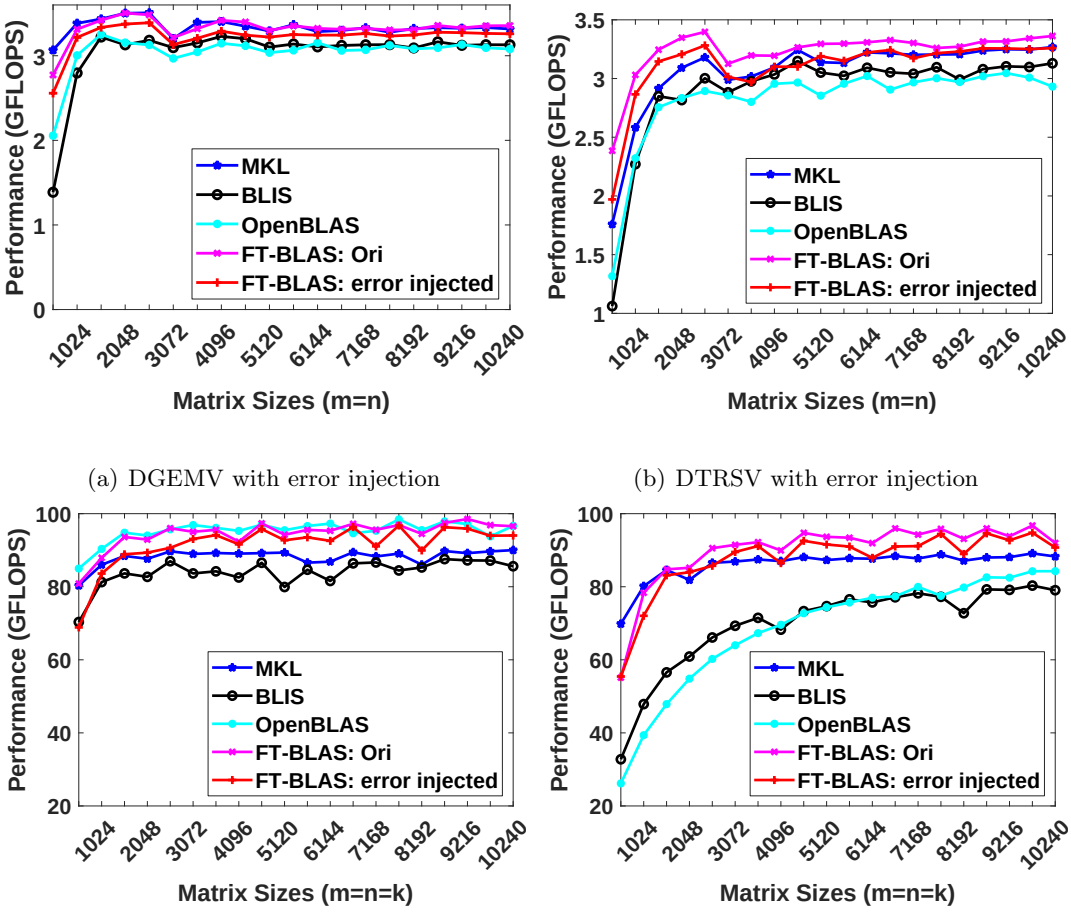
(a) FT-DNRM2

(b) FT-DGEMV

(c) FT-DTRSV

(d) FT-DGEMM

Figure 2.13: Comparisons of BLAS routines with FT on AMD Zen2.



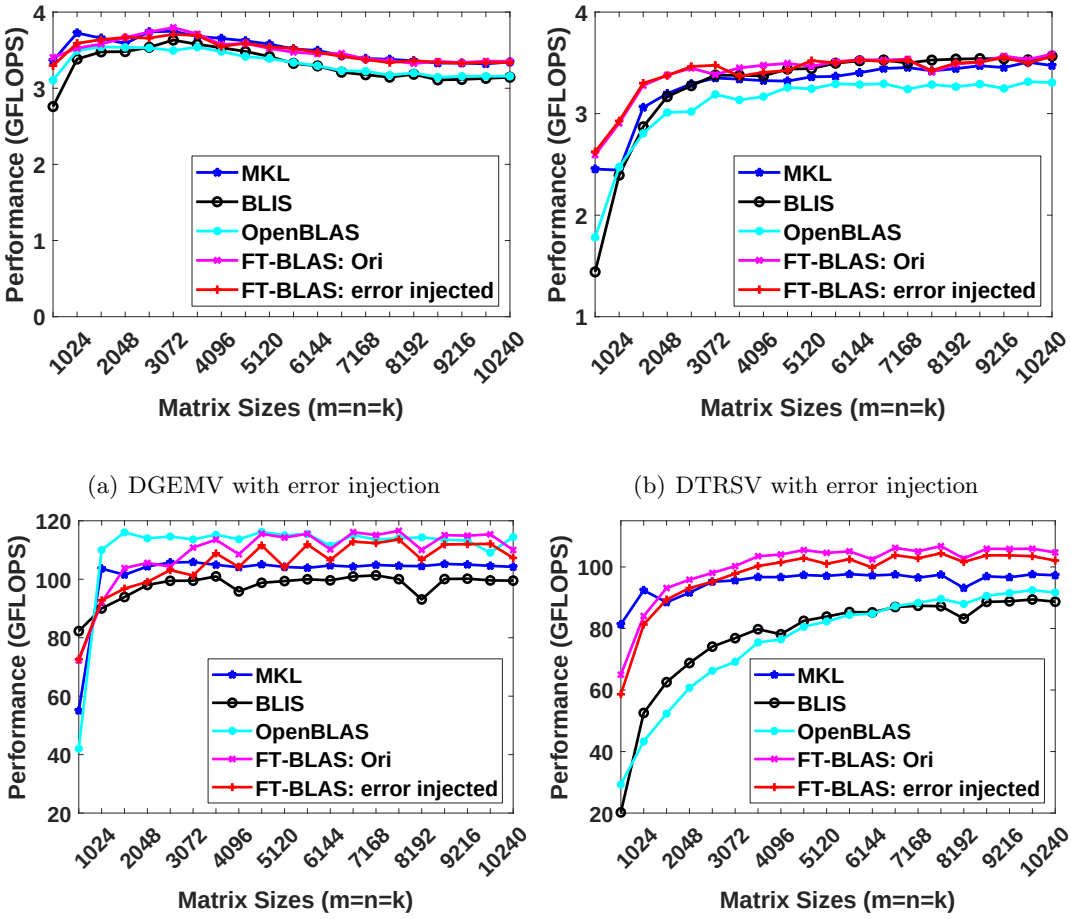
(a) DGEMV with error injection

(b) DTRSV with error injection

(c) DGEMM with error injection

(d) DTRSM with error injection

Figure 2.14: Performance under error injection on Skylake.



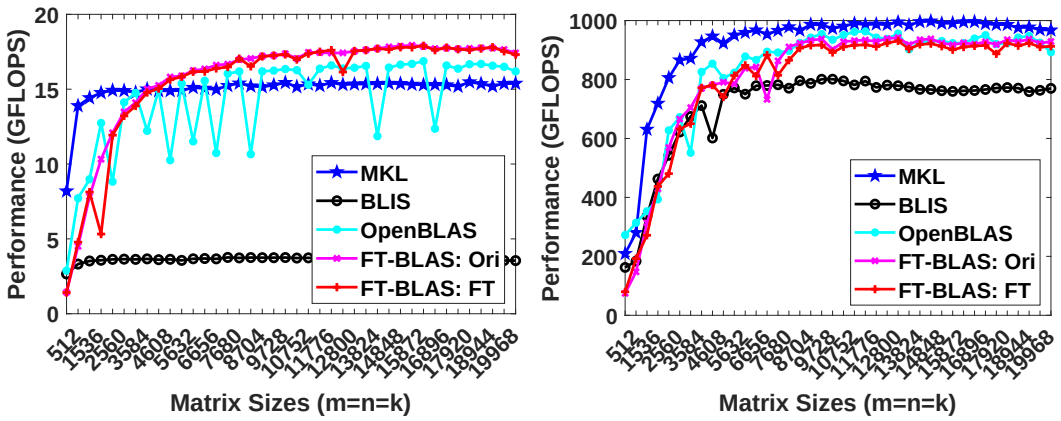
(a) DGEMV with error injection

(b) DTRSV with error injection

(c) DGEMM with error injection

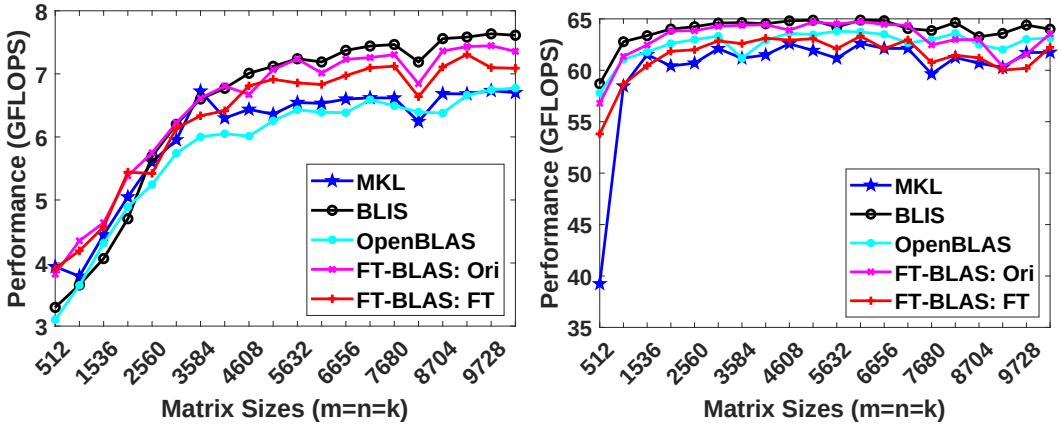
(d) DTRSM with error injection

Figure 2.15: Performance under error injection on Cascade Lake.



(a) FT-DGEMV (b) FT-DGEMM

Figure 2.16: Parallel performance under error injection on Cascade Lake.



(a) FT-DGEMV (b) FT-DGEMM

Figure 2.17: Performance under error injection on AMD Zen2.

Chapter 3

XeHE: A GPU-Accelerated Homomorphic Encryption Library

3.1 Introduction

The COVID-19 pandemic boosts the rapidly growing demand of enterprises on cloud computing. By 2021, 50% of enterprise workloads are deployed to public clouds, and this percentage is expected to reach 57% in the next 12 months [64]. Although outsourcing data processing to cloud resources enables enterprises to relieve the overhead of deployment and maintenance for their private servers, it raises security and privacy concerns of the potential sensitive data exposure.

Adopting traditional encryption schemes to address this privacy concern is less favorable because a traditional encryption scheme requires decrypting the data before the computation, which presents a vulnerability and may destroy the data privacy. In contrast,

Homomorphic Encryption (HE), an emerging cryptographic encryption scheme, is considered to be one of the most promising solutions to such issues. HE allows computations to be performed directly on encrypted messages without the need for decryption. This encryption scheme, thus, protects private data from both internal malicious actors and external intruders, while assuming honest computations.

In 1978, Rivest, Adleman, and Dertouzous [158], first introduced the idea of computing on encrypted data through the use of "privacy homomorphisms". Since then, several HE schemes have been invented, which can be categorized by the types of encrypted computation they support. *Partial* HE schemes enable only encrypted additions or multiplications. The famous RSA cryptosystem is, in fact, the first HE scheme, supporting encrypted modular multiplications. In contrast, the Paillier cryptosystem [143] is a partial HE scheme that supports only modular additions.

Levelled HE schemes, on the other hand, support both encrypted additions and multiplications, but only up to a certain circuit depth determined by the encryption parameters. The Brakerski/Fan-Vercauteren (BFV) [62] and Brakerski-Gentry-Vaikuntanathan (BGV) [21] schemes are two popular leveled HE schemes used today, which support exact integer computation. In [43], Cheon, Kim, Kim and Song presented the CKKS scheme, which treats the encryption noise as part of approximation errors that occur during computations within floating-point numerical representation. This imprecision requires a refined security model [111], but provides faster runtimes than BFV/BGV in practice.

Fully HE schemes enable an unlimited number of encrypted operations, typically by adding an expensive bootstrapping step to a levelled HE scheme, as first detailed by

Craig Gentry [66]. TFHE [47] improves the runtime of bootstrapping, but requires evaluating circuits on binary gates, which becomes expensive for standard 32-bit or 64-bit arithmetic. The improved capabilities and performance of these HE schemes have enabled a host of increasingly sophisticated real-world privacy-preserving applications. Early applications included basic statistics and logistic regression evaluation [126]. More recently, HE applications have expanded to a wide variety of applications, including privatized medical data analytics and privacy-preserving machine learning. [19, 20, 44, 76, 153].

To address the memory and runtime overhead of HE — a major obstacle to immediate real-world deployments, HE libraries support efficient implementations of multiple HE schemes, including Microsoft SEAL [108] (BFV/CKKS), HElib [80] (BFV/BGV/CKKS), and PALISADE [148] (BGV/BFV/CKKS/TFHE). In [18], Intel published HEXL, accelerating HE integer arithmetic on finite fields by featuring Intel Advanced Vector Extensions 512® (Intel AVX512) instructions. Since GPUs deliver higher memory bandwidth and computing throughput with lower normalized power consumption, researchers presented libraries such as cuHE [53], TFHE [47] and NuFHE [129] to accelerate HE using CUDA-enabled GPUs.

Although HE optimizations on CPUs and CUDA-enabled GPUs have been reported before, an architecture-aware HE library optimized for Intel GPUs has not been available. In addition, previous works that accelerate HE libraries majorly focus on optimizing Number Theoretic Transform (NTT) and inverse NTT (iNTT) computing kernels, since these two algorithms account for substantial execution time of HE routines (e.g., 72%-81% in its baseline variant on Intel GPUs according to our benchmarks in Fig. 3.5).

However, engineering an efficient HE library requires systematical optimizations for the whole HE pipeline beyond computing kernels. In this chapter, we present a HE library optimized for Intel GPUs based on the CKKS scheme. We not only provide a set of highly optimized computing kernels such as NTT and iNTT, but also optimize the whole HE evaluation pipeline at both the instruction level and application level. More specifically, our contributions include:

- To the best of our knowledge, we design and develop the first-ever SYCL-based GPU backend for Microsoft SEAL APIs, which is also the first HE library based on the CKKS scheme optimized for Intel GPUs.
- We provide a staged implementation of NTT leveraging shared local memory of Intel GPUs. We also optimize NTT by employing strategies including high-radix algorithm, kernel fusion, and explicit multiple-tile submission.
- From the instruction level, we enable low-level optimizations for 64-bit integer modular addition and modular multiplication using inline assembly. We also provide a fused modular multiplication-addition operation to reduce the number of costly modular operations.
- From the application level, we introduce the memory cache mechanism to recycle freed memory buffers on device to avoid the run-time memory allocation overhead. We also design fully asynchronous HE operators and asynchronous end-to-end HE evaluation pipelines.

- We benchmark our HE library on two latest Intel GPUs. Experimental results show that our NTT implementations reaches up to 79.8% and 85.7% of the theoretical peak performance on both experimental GPUs, faster than the naive GPU baseline by 9.93X and 7.02X, respectively.

- Our NTT and assembly-level optimizations accelerate five HE evaluation routines under the CKKS scheme by 2.32X - 3.05X. In addition, the polynomial element-wise matrix multiplication applications are accelerated by 2.68X - 3.11X by our all-together systematic optimizations.

The rest of the paper is organized as follows: we introduce background and related works in Section 3.2, and then detail the asynchronous design and systematic optimization approaches in Section 3.3. Evaluation results are given in Section 3.4. We conclude our paper and present future work in Section 3.5.

3.2 Background and Related Works

In this section, we briefly introduce the basics of the CKKS HE scheme. We then introduce the general architecture of Intel GPUs and summarize prior works of NTT optimizations on both CPUs and GPUs.

3.2.1 Basics of CKKS

The CKKS scheme was first introduced in [43], enabling approximation computation on complex numbers. This approximate computation is particularly suitable for real-world floating-point operations that are approximate by design. Further work improved

CKKS to support a full residue number system (RNS) [42] and bootstrapping [41]. In this chapter, we select CKKS as our FHE scheme, as implemented in Microsoft SEAL [108].

The CKKS scheme is composed of following basic primitives: *KeyGen*, *Encode*, *Decode*, *Encrypt*, *Decrypt*, *Add*, *Multiply* (*Mul*), *Relinearize* (*Relin*) and *Rescale* (*RS*). To be more specific, *KeyGen* first generates a set of keys for the CKKS scheme. An input message is encoded to a plaintext and then encrypted to a ciphertext. One can evaluate (compute) directly on the encrypted messages (ciphertexts). Noises are accumulated during the HE evaluation until one applies a *Relin* followed by a *RS* to the ciphertext. Once all the HE computations are completed, the result ciphertext is decrypted and decoded, providing the same result as ordinary non-HE computations. We provide only cursory descriptions here and refer interested readers to [43] for details.

3.2.2 Number Theoretic Transform and Residue Number System

As noted in [118], the NTT can be exploited to accelerate multiplications in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^N + 1)$. We represent polynomials using a coefficient embedding: $\mathbf{a} = (a_0, ..., a_{N-1}) \in \mathbb{Z}_q^N$ and $\mathbf{b} = (b_0, ..., b_{N-1}) \in \mathbb{Z}_q^N$. Let ω be a primitive N-th root of unity in $\mathbb{Z}_q$ such that $\omega^N \equiv 1(\mod q)$. In addition, let ψ be the $2N$-th root of unity in $\mathbb{Z}_q$ such that $\psi^2 = \omega$. Further defining $\tilde{\mathbf{a}} = (a_0, \psi a_1, ..., \psi^{N-1} a_{N-1})$ and $\tilde{\mathbf{b}} = (b_0, \psi b_1, ..., \psi^{N-1} b_{N-1})$, one can quickly verify that for $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} \in \mathbb{Z}_q^N$, there holds the relationship $\mathbf{c} = \mathbf{\Psi^{-1}} \odot \text{iNTT}(\text{NTT}(\tilde{\mathbf{a}}) \odot \text{NTT}(\tilde{\mathbf{b}}))$. Here $\odot$ denotes element-wise multiplication and $\mathbf{\Psi^{-1}}$ represents the vector $(1, \psi^{-1}, \psi^{-2}, ..., \psi^{-(N-1)})$. Therefore, the total computational complexity of ciphertext multiplication in $\mathcal{R}_q$ is reduced from $O(N^2)$ to $O(N \log N)$.

In practice, since polynomial coefficients in the ring space are big integers under modulus q, multiplying these coefficients becomes computationally expensive. The Chinese Remainder Theorem (CRT) is typically employed to reduce this cost by transforming large integers to the Residue Number System (RNS) representation. According to CRT, one can represent the large integer $x \mod q$ using its remainders ($x \mod p_1, x \mod p_2, \ldots, x \mod p_n$), where the moduli $(p_1, p_2, ..., p_n)$ are co-prime such that $\Pi p_i = q$. We note the CKKS scheme has been improved from the initial presentation in Section 3.2.1 to take full advantage of the RNS [42].

To summarize what we have discussed, to multiply polynomials $\mathbf{a}$ and $\mathbf{b}$ represented as vectors in $\mathbb{Z}_q^N$, one needs to first perform the NTT to transform the negative wrapped $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$ to the NTT domain. After finishing element-wise polynomial multiplication in the NTT domain, the iNTT is applied to convert the product to the coefficient embedding domain. When the polynomials are in RNS form, both the NTT and iNTT are decomposed to n concurrent subtasks. Finally, we compute the outer product result by merging the iNTT-converted polynomial with $\mathbf{\Psi^{-1}}$.

3.2.3 NTT optimizations

Due to the pervasive usage of NTT and iNTT in HE, prior researchers proposed optimized implementations for NTT on CPUs [18], CUDA-enabled GPUs [5,70,105,183] and FPGAs [106,157]. On the CPU end, SIMD instructions enable a wider data processing width to accelerate a broad range of applications [1,181,185,206,210]. Leveraging this architectural feature, Intel HEXL provides a CPU implementation of the radix-2 negacyclic NTT using Intel AVX512 instructions [18] and Harvey's lazy modular reduction approach [81]. GPU-

accelerated NTT implementations typically adopt the hierarchical algorithm first presented by Microsoft Research for the Discrete Fourier Transform (DFT) [75]. In [183], researchers implemented the hierarchical NTT with twiddle factors, which are multiplicative constants in the butterfly computation stage (i.e. W in Algorithm 1), cached in shared memory. Rather than caching twiddle factors, in [105], Kim et al. computed some twiddle factors on-the-fly to reduce the cost of modular multiplication and the memory access number of NTT. In [70], Goey et al. considered the built-in warp shuffling mechanism of CUDA-enabled GPUs to optimize NTT.

The hierarchical NTT implementation computes the NTT in three or four phases [70,75]. An N-point NTT sequence is first partitioned into two dimensions $N = N_\alpha \cdot N_\beta$ and then N_α NTT workloads are proceeded simultaneously, where each workload computes an N_β-point NTT. After this column-wise NTT phase is completed, all elements are multiplied by their corresponding twiddle factors and stored to the global memory. In the next phase, N_β simultaneous row-wise N_α-point NTTs are computed followed by a transpose before storing back to the global memory. N_α and N_β are selected to fit the size of shared memory on GPUs. Considering both the RNS representation of NTT and the batched processing opportunities in real-world applications can provide us with sufficient parallelisms, we adopt the staged NTT implementation rather than the hierarchical NTT implementation in this chapter.

3.2.4 An Overview of Intel GPUs

We use the Intel Gen11 GPU as an example [96] to elaborate the hierarchical architecture of Intel GPUs. An Intel GPU contains a set of execution units (EU), where

each EU supports up to seven simultaneous hardware threads, namely EU threads. In each EU, there is a pair of 128-bit SIMD ALUs, which support both floating-point and integer computations. Each of these simultaneous hardware threads has a 4KB general register file (GRF). So, an EU contains $7 \times 4\text{KB} = 28\text{KB}$ GRF. Meanwhile, GRF can be viewed as a continuous storage area holding a vector of 16-bit or 32-bit elements. For most Intel Gen11 GPUs, 8 EUs are aggregated into 1 Subslice. EUs in each Subslice can share data and communicate with each other through a 64KB highly banked data structure — shared local memory (SLM). SLM is accessible to all EUs in a Subslice but is private to EUs outside of this Subslice. Not only supporting a shared storage unit, Subslices also possess their own thread dispatchers and instruction caches. Eight Subslices further group into a Slice, while additional logic such as geometry and L3 cache are integrated accordingly.

3.3 Designs and Optimizations

In the CKKS scheme, an input message is first encoded and then encrypted to generate ciphertexts using the public key provided by the key generation primitive. We compute directly on the encrypted messages. Once the all computations are completed, the results can be decrypted and decoded by the private key's owner. Figure 3.1 describes the control flow of our asynchronous HE library. The client host (CPU) generates security parameters, submits GPU compute kernels and sends encrypted data to the GPU upon request. The CPU works asynchronously to the GPU until it receives result ciphertexts from the GPU. Regarding the data flow presented in the same figure, the CPU sends encrypted input data to GPU memory and receives output ciphertexts from the GPU for

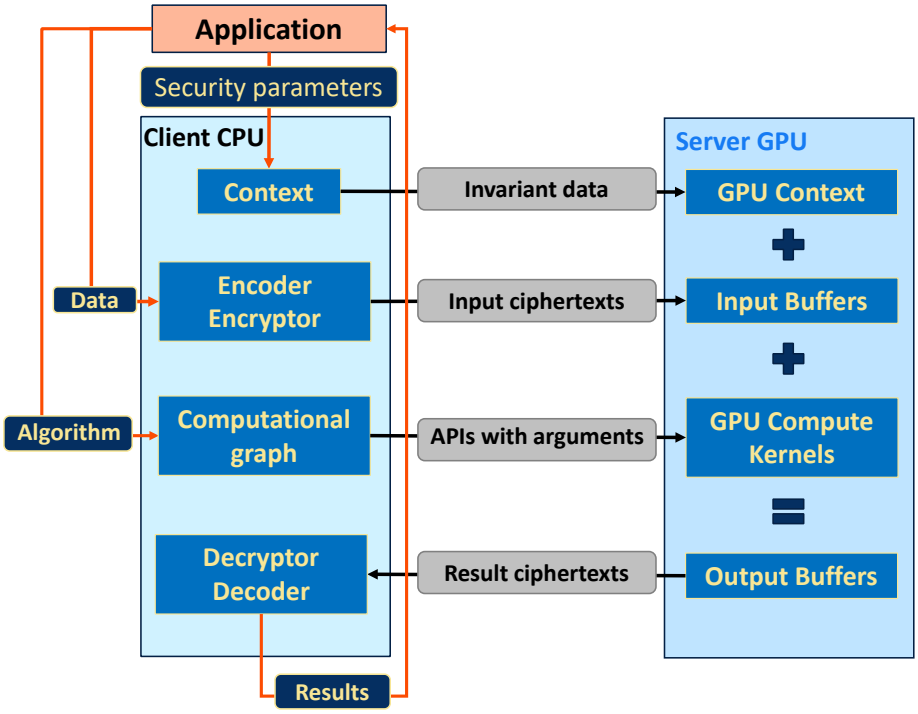


Figure 3.1: Client (CPU)/Server (GPU) control/data flow.

decryption. Our HE library accelerates the HE evaluation using Intel GPUs while leaving other phases such as key generation, encoding, encryption, decryption and decoding on the client CPU.

Once all the inputs and static data are sent to the GPU, the synchronization with the host becomes unnecessary. Since all host-device synchronizations take additional time, we developed a fully asynchronous execution pipeline to economize on synchronizations. As shown in Figure 3.2, the computation on the GPU starts as soon as the first kernel of the computational graph is submitted. Meanwhile, GPU buffers are allocated and managed at runtime. GPU synchronizes with the host only after the buffers with the results are transferred back to the system memory. In coordination with our memory allocation cache design (Fig. 3.11), all ciphertext objects are alive until the result ciphertext is sent to the

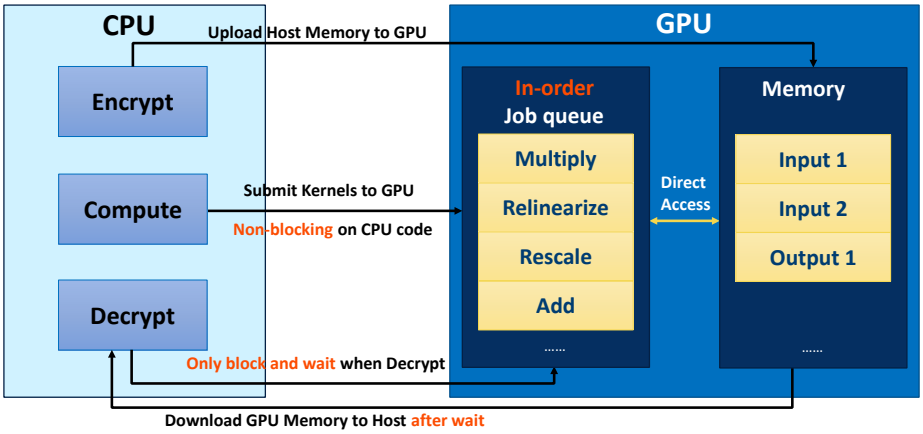


Figure 3.2: Asynchronous execution scheme

CPU for decryption. At that moment, all differed deletions for memory buffers are served. This design simplifies the lifetime management of ciphertext objects for SEAL API users. A more detailed description of our system design is presented in a book chapter at [123]. In the following contents of this section, we present optimizations of our library from three different angles: instruction, algorithm and application.

3.3.1 Instruction-Level Optimizations

Our HE library supports basic instructions such as addition, subtraction, multiplication and modular reduction – all are 64-bit integer (int64) operations. We explicitly select int64 because our goal has been to provide accelerated SEAL APIs on Intel GPUs transparently. This is the key reason why our current top-level software does not exactly fit to drive 32-bit integer (int32) calculations, although we envision to support both int32 and int64 eventually. Among these operations, the most expensive are modulus-related operations such as modular addition and modular multiplication. Although we can accelerate modular reduction using the Barrett reduction algorithm, which transforms the division operation

to the less expensive multiplication operation, modular computations remain costly since no modern GPUs support int64 multiplication natively. Such multiplications are emulated at software level with the compiler support.

Based on these observations, we propose instruction-level optimizations from two aspects: 1) fusing modular multiplication with modular addition to reduce the number of modulo operations and 2) optimizing modular addition/multiplication from assembly level to remedy the compiler deficiency.

Fused Modular Multiplication-Addition Operation (`mad_mod`)

Rather than eagerly applying modulo operation after both multiplication and addition, we propose to perform only one modulo operation after a *pair* of consecutive multiplication and addition operations, namely a `mad_mod` operation. We store the output of int64 multiplication in an 128-bit array. The potential overflow issue introduced by cancelling a modulus after addition is not a concern when both operands of addition are integers strictly less than 64 bits. This assumption holds because to assure a faster NTT transform, we adopt David Harvey's optimizations [81] following SEAL. Therefore, all of our ciphertexts are in the ring space under a integer modulus less than 60 bits.

Optimizing Modular Addition/Multiplication From Assembly Level

We review the assembly codes generated by the Intel DPC++ compiler and seek low-level optimization opportunities for the HE pipeline. We locate such opportunities in two of our core arithmetic operations: Unsigned Modular Addition, and Unsigned Integer Multiplication.

1: add dst, src1, src2 2: cmp.lt P1, dst, modulus 3: (P1) sel modulus, 0x0, modulus 4: add dst, dst, (-)modulus	1: add dst, src1, src2 2: cmp.lt P1, dst, modulus 3: (P1) add dst, dst, (-)modulus
(a) Compiler-generated assembly	**(b) Hand-crafted assembly**

Figure 3.3: Pseudo int64 addmod assembly

Unsigned Modular Addition (`add_mod`) Fig. 3.3(a) presents the compiler-generated sequence of `add_mod`. Two source operands (src1, src2), and the result is stored to the register (dst). If the summation exceeds the value of modulus, the result is added by the negative modulus; otherwise, no update is needed. The compiler suboptimally implements this logic by conditionally initializing the addend (modulus) and then updating the result. At line4 in Fig. 3.3(b), we directly perform a conditional addition by leveraging the optional guard predicate (P1) of `add` on Intel GPUs. Here we eliminate one instruction at the assembly level for this core HE arithmetic operation, which enables direct benefits to the whole HE pipeline.

1: mul temp, src2, src1 2: mulh temp1, src2, src1 3: mul temp2, src2, src1 4: add temp1, temp1, temp2 5: mul temp2, src2, src1 6: add temp1, temp1, temp2 7: mov dst_low, temp 8: mov dst_high, temp1	1: mul_low_high dst_low_high, src1, src2
(a) Compiler-generated assembly	**(b) Hand-crafted assembly**

Figure 3.4: Pseudo mul64 assembly

Unsigned Integer Multiplication (`mul64`) Another example where our hand-crafted assembly code outperforms the compiler-generated instruction sequence can be found in

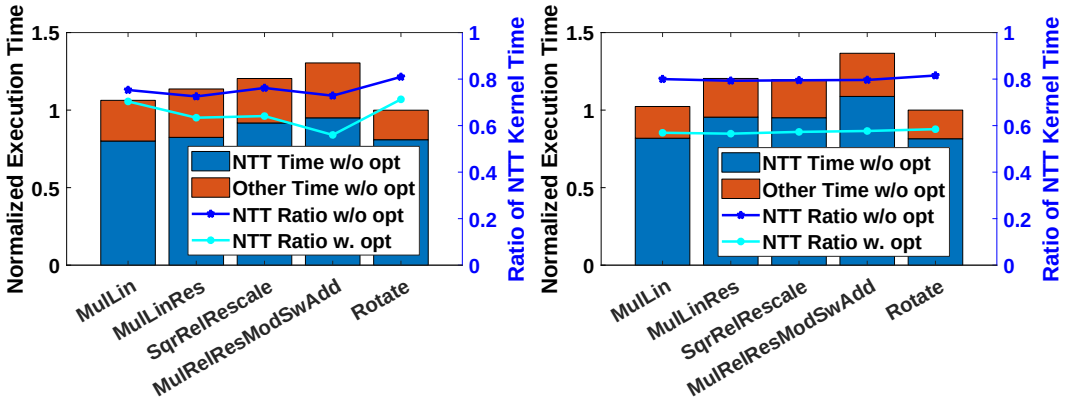
int64 multiplication. Fig. 3.4(a) shows the compiler-generated instruction sequence to multiply two 64-bit integers, producing an 128-bit result which is stored in two 64-bit registers (dst_high, dst_low). The instruction `mul` takes two 64bit operands to compute the lower 64 bits of the multiplication result, while `mulh` computes the higher 64 bits.

Although the compiler-generated code provides us with a correct result (the lower 64 bits of int64 multiplication), it also computes the higher 64 bits of in64 multiplication, which are redundant in our case. In order to address this issue, we adopt the built-in `mul_low_high` operator to explicitly compute the lower 64-bit multiplication result, as shown in Fig. 3.4(b). To elaborate, `mul_low_high` receives two int32 operands (cast from int64) and stores both the lower and higher 32 bits of the result in a 64-bit destination [93].

This presents an example of a compilation deficiency related to variables' type-casting. By default, the compiler minimizes the number of type-casting instructions, but it is overall detrimental in the above case. An integer multiplication, where both operands are int32, is more efficient than a longer emulated implementation whose both operands are of type int64. Our inline assembly bypasses this deficiency, yielding a significant reduction in instruction count from our original int64 multiplication implementation. As will be shown in Section 3.4, optimizations aimed at our core arithmetic operations greatly impact the performance of HE.

3.3.2 Algorithmic Level Optimizations (NTT)

An efficient NTT implementation is crucial for HE computations since it accounts for a substantial percentage of the total HE computation time [103, 105, 159]. Figure 3.5



(a) Profiling on Device1

(b) Profiling on Device2

Figure 3.5: Profiling for HE routines

presents the relative execution time of five HE evaluation routines and the percentage of NTT in each routine before and after optimizing NTT kernels on two latest Intel GPUs, Device1 and Device2. We observe that NTT accounts for 79.99% and 75.64% of the total execution time in average on these two platforms. After applying optimizations as shown in Fig. 3.16 and 3.18, these NTT kernel ratios remain greater than 56% on both devices.

Naive Radix-2 NTT

We start NTT optimizations from the most naive radix-2 implementation. This reference implementation of NTT, as shown in Figure 3.6, distributes rounds of radix-2 NTT butterfly operations among work-items, which are analogs to CUDA threads. In each round of the NTT computation, all the work-items compute their own butterfly operations and exchange data with other work-items using the global memory. More specifically, the k-th element will exchange with the $k + gap$ th element, while the exchanging gap sizes are halved after each round of NTT until it becomes equal to 1. Accordingly, an N-point

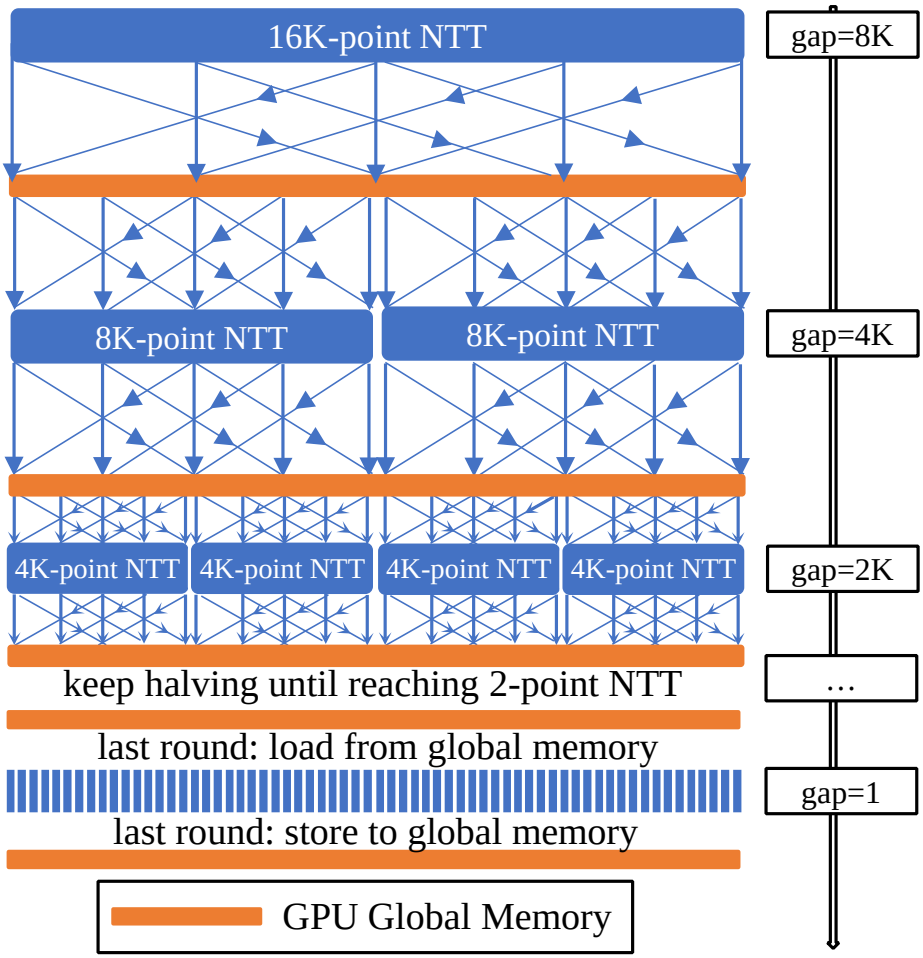


Figure 3.6: Naive implementation of 16K-point NTT

NTT is executed $\log(N)$ rounds throughout the computation. For each round of the NTT butterfly computation, one accesses the global memory $2N$ times. Here we multiply it by two because of both load and store operations. We ignore the twiddle factor memory access number in this semi-quantitative analysis.

At the lowest level, the NTT butterfly computation is accelerated using Algorithm 1 [81]. Since the output X', Y' of Algorithm 1 are both in $[0, 4p)$, to ensure all elements of the output NTT sequence falls inside of the interval $[0, p)$, a last round offsetting needs to be appended to the end of NTT computations. Therefore, the naive implementation of an N-

point NTT needs to access the global memory $2N\log(N)$ times for the NTT and $2N$ extra times for last round processing. This kernel reaches only 10.08% of the peak performance for a 32K-point, 1024-instance NTT as shown in Fig. 3.12(b). Here the number of instances refers to the number of polynomials in Fig. 3.10. More specifically, 1024-instance NTT denotes 1024 batched instances of N-point NTT computation.

Algorithm 1 ***Input:*** $0 \leq X, Y \leq 4p$, $p < \beta/4$, $0 < W < p, 0 < \lfloor W\beta/p \rfloor = W' < \beta$ ***Output:***
$X' = X + WY \mod p$ $Y' = X - WY \mod p$ $0 \leq X', Y' \leq 4p$ ***if*** $X \geqslant 2p$ ***then*** $X \leftarrow X - 2p$
$Q \leftarrow \lfloor W'Y/\beta \rfloor$ $T \leftarrow (WY - Qp) \mod \beta$ $X' \leftarrow X + T$ $Y' \leftarrow X - T + 2p$ ***return*** X', Y'

Staged Radix-2 NTT With Shared Local Memory

Since the naive radix-2 NTT exchanges data using the global memory, its performance is significantly bounded by the global memory bandwidth. To address this issue, we keep data close to computing units by leveraging shared local memory (SLM) in Intel GPUs, a memory region that is accessible to all the work-items belonging to the same work-group. Here the work-group is analogous to the CUDA thread block. Because the data exchanging gap size is halved after each round of NTT, at a certain round, the gap size becomes sufficiently small so that all data to exchange can be held in SLM. We call this threshold gap size `TER_SLM_GAP_SZ`, after which we retain the data in SLM for communication among work-items to avoid the expensive global memory latency. For example, in a 16K-point NTT, we first compute one round of NTT and exchange data using global memory and then the data exchanging gap size has decreased to 4K. We set the `TER_SLM_GAP_SZ` to 4K because the size of the SLM on most Intel GPUs is 64KB, which can hold 8K int64 elements. For the remaining rounds, the data are held in SLM until all computations are completed.

SIMD Shuffling

In addition to introducing shared local memory, when the exchanging distance becomes sufficiently small that all data to exchange are held by work-items in the same subgroup, we perform SIMD shuffling directly among all the work-items in the same subgroup after NTT butterfly computations. In Figure 3.7, we present the rationale of two SIMD shuffling operations among three stages. When the SIMD width equals to 8, there are 8 work-items in a subgroup. For the radix-2 NTT implementation, each work-item holds two elements of the NTT sequence in registers, namely one slot. We denote two local registers of each work-item as Register 0 and Register 1. At the end of Stage 1, where the gap size equals to 8, one needs to exchange data at positions "8, 9, 10, 11" with "4, 5, 6, 7". Such operations can be implemented using `shuffle` of the Intel extension of DPC++ [94]. More specifically, four lanes (lane ID: 0, 1, 2, 3) are exchanging data stored in their Register 1 with Register 0 of lane 4, 5, 6, 7. At the end of Stage 2, where the exchanging gap size is halved from 8 to 4, lanes 0, 1 will exchange data of their Register 1 with Register 0 of lanes 2, 3; similarly, lanes 4, 5 exchange their Register 0 with Register 1 of lanes 6, 7. For the remaining rounds, the data are held in registers and exchanged among work-items in the same subgroup by SIMD shuffling until the gap size becomes equal to 1.

Figure 3.8 summarizes our staged NTT implementation with both SLM and SIMD shuffling considered using 16K-point NTT as an example. Before the data exchanging gap size reaches the threshold to exchange data in SLM, work-items communicate through the global memory. The gap size is initially equal to 8K and is halved after each round of the NTT butterfly computation. After reaching the SLM threshold (8K-point NTT in this

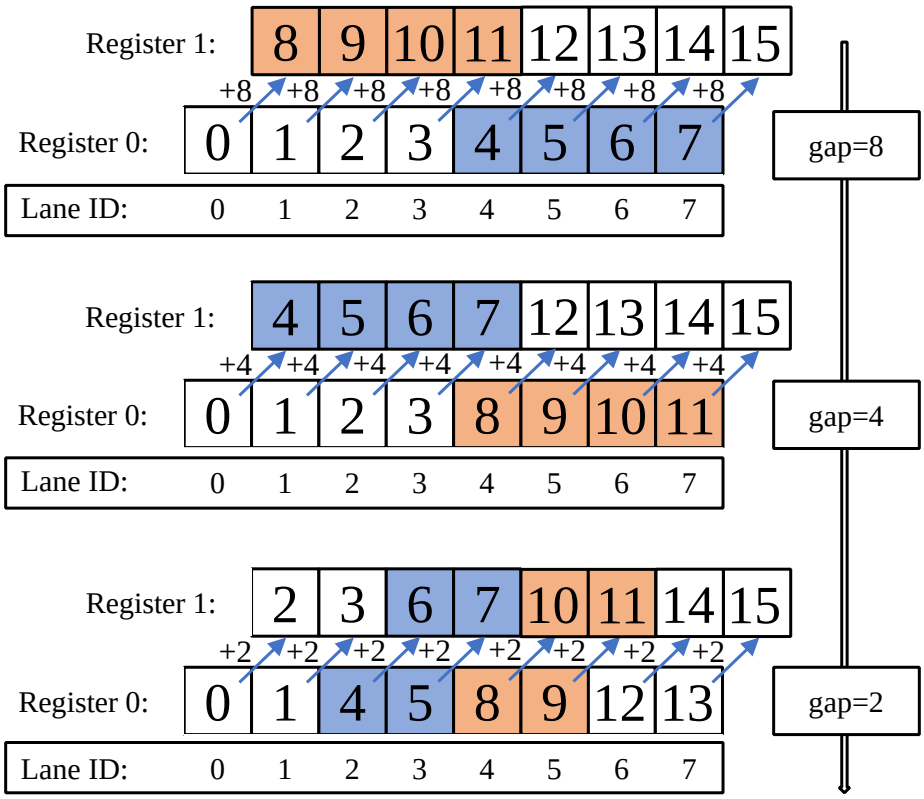


Figure 3.7: SIMD shuffling for data exchanging in NTT.

graphical example), one computes NTT butterfly operations and exchanges data through SLM until the gap size equals the threshold to exchange data using SIMD shuffling inside subgroups. It is worth mentioning that the SIMD kernel is fused with the aforementioned last round processing operation to reduce all NTT elements to $[0, p)$.

More Aggressive Register Blocking

Intel GPUs typically consist of 4KB GRF for each EU thread. When the SIMD width equals 8, that indicates 8 work-items are bounded executing as an EU thread in the SIMD manner. For the radix-2 NTT implementation, each work-item needs four registers, where two of them are used to hold NTT data elements and the other two are for twiddle

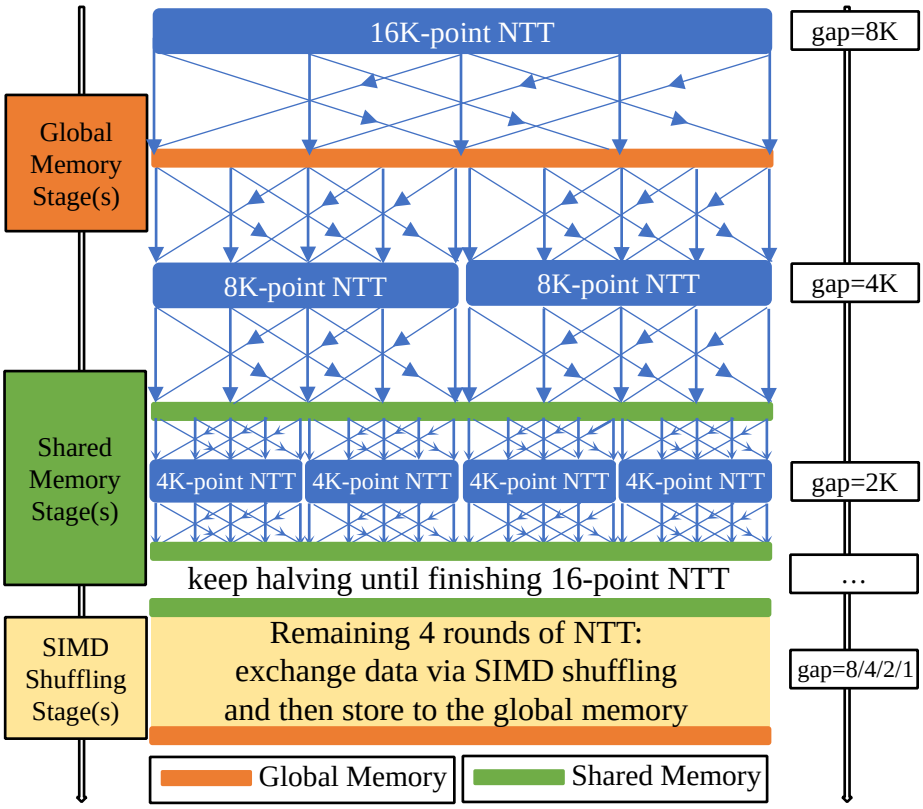


Figure 3.8: Staged implementation of 16K-point NTT

factors. Therefore, the NTT-related computation consumes $4 \cdot 8 \cdot 8\text{B} = 256\text{B}$ GRF for each EU thread — 6.25% of total GRF, indicating that the hardware is significantly underutilized at the register level.

Rather than initializing 1 slot of registers, one can assign more workloads (e.g. 2 register slots) to each work-item. For a subgroup of size 8, there are $8 \cdot 2 = 16$ NTT elements being held in registers in the SIMD kernel. We refer it as SIMD(16,8). Figure 3.9 shows a graphical example of the shuffling operation between two stages in SIMD(16,8). In this two-slot SIMD shuffling example, each work-item holds 4 elements in registers, namely 2 slots of registers, for the butterfly computation and data exchanging. Compared with the

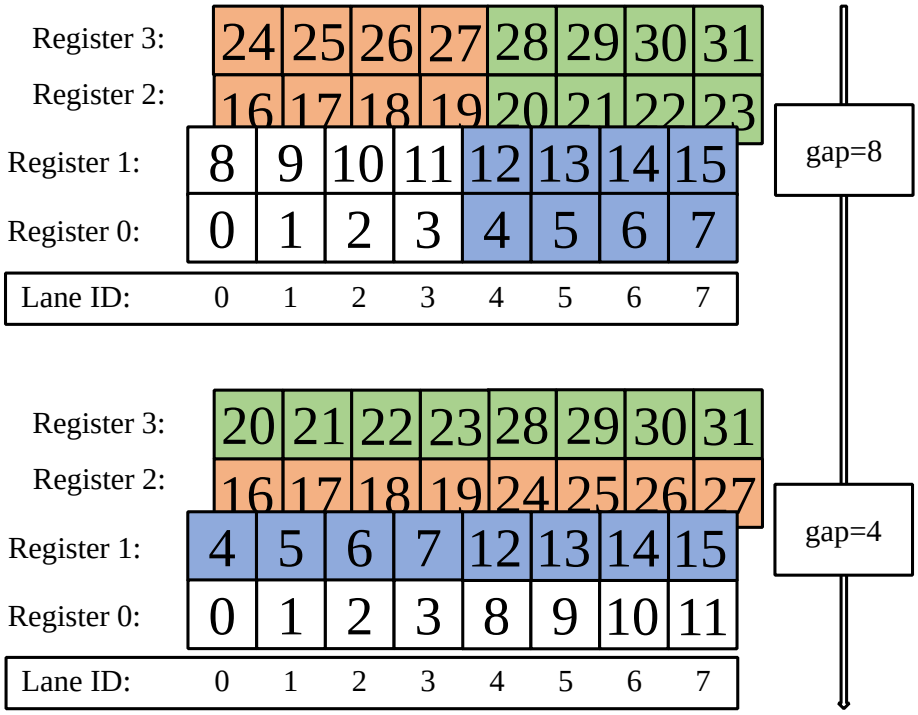


Figure 3.9: Multi-slot SIMD shuffling in NTT

single-slot implementation, the multi-slot SIMD implementation results in fewer accesses to the shared local memory, but suffers from higher register pressure and the in-register data exchange overhead. In practice, the efficiencies of both 2-slot SIMD(16,8) and 4-slot SIMD(32,8) implementations are worse than the 1-slot SIMD(8,8), suggesting that negative aspects dominate the performance.

High-Radix NTT

The staged NTT implementation with multi-slot SIMD shuffling increases the register-level data re-use without introducing the register spilling issue. However, this higher hardware utilization on registers comes at a cost of introducing extra efforts to compute target shuffling register indices and lane indices for SIMD shuffling. These integer operations

compete for the same ALU port with NTT butterfly computations, adding a non-negligible overhead to the overall performance as shown in Section 3.4. This performance loss, caused by data exchange overhead, motivates the high-radix NTT implementation, which requires no extra data exchange and communication among work-items compared with the naive radix-2 implementation.

We use radix-8 NTT as an example to demonstrate high-radix NTT algorithms. Each work-item allocates 8 registers to hold NTT elements and 8 more registers to hold root power and root power quotients as for the twiddle factors. For a specific round where the exchanging gap size is *gap*, one loads eight NTT elements from either global or shared local memory, indexing at $\{k, k+gap, k+2\cdot gap, ..., k+7\cdot gap\}$. There are three internal rounds of butterfly computations before a radix-8 NTT algorithm needs to exchange data among work-items. In the first internal round, four pairs of 2-point butterfly computations are performed among $\{x[k], x[k+4\cdot gap]\}$, $\{x[k+gap], x[k+5\cdot gap]\}$, $\{x[k+2\cdot gap], x[k+6\cdot gap]\}$ and $\{x[k+3\cdot gap], x[k+7\cdot gap]\}$. For the second internal round, these eight elements, still held in registers, are re-paired to $\{x[k], x[k+2\cdot gap]\}$, $\{x[k+gap], x[k+3\cdot gap]\}$, $\{x[k+4\cdot gap], x[k+6\cdot gap]\}$ and $\{x[k+5\cdot gap], x[k+7\cdot gap]\}$ so that Algorithm 1 can be leveraged. In the last internal round of the radix-8 kernel, each two consecutive elements are paired and fed into the 2-point butterfly algorithm. After all in-register computations are completed, we store results back to either global memory or shared local memory, depending on whether it is a global memory kernel or a shared local memory kernel. The exchanging gap size *gap* is divided by 8 as a new round of NTT is initiated. Same as the radix-2 NTT, the radix-8 NTT computations are completed when *gap* becomes equal to 1.

Compared with the radix-2 NTT, a radix-R NTT ($R = 4, 8, 16, ...$) decreases the total memory access number from $2N \log_2(N)$ to $2N \log_R(N)$. In coordination with adopting SLM to exchange data, the high-radix NTT maintains the data close to computing units and maximizes the overall efficiency. In Section 3.4 we show that the radix-8 NTT with SLM is up to 4.23X faster than the naive radix-2 NTT on Intel GPUs.

The Parallelism of Staged NTT for HE

Figure 3.10 presents the parallelism of NTT in the HE pipeline. Besides mapping one dimensional NTT operations to work-items as elaborated previously, both RNS and the batched number of polynomials can provide the staged NTT implementation with additional parallelisms. To be more specific, the RNS base can be up to several dozens [105] while a batch size can be up to tens of thousands in real-world deep learning tasks [104].

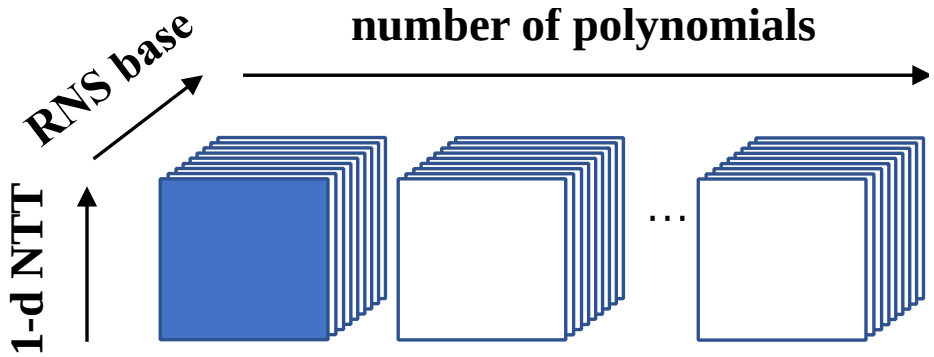


Figure 3.10: The parallelism of NTT for HE

3.3.3 Application-Level Optimizations

In addition to instruction-level and algorithm-level optimizations, we also optimize our GPU-accelerated HE library from the application level.

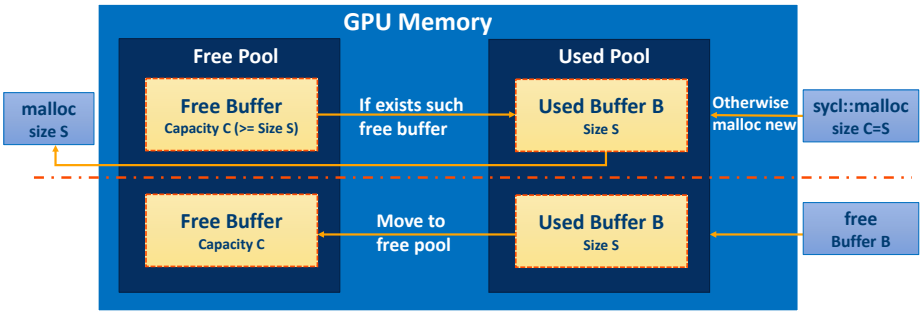


Figure 3.11: Memory cache design

Memory Cache

To reduce the overhead introduced by runtime memory allocation, we design a memory cache mechanism for our HE library, as shown in Figure 3.11. Similar to Microsoft SEAL, we introduce a memory pool to reuse allocated GPU memory buffers in the HE pipeline. A request for a new GPU memory buffer is routed through the memory cache for any existing free buffer with a capacity larger than the current request. If such buffer is found, this existing buffer is reused instead of allocating a new one. Upon freeing such a buffer, it is moved back to the free pool for potential reuse.

Inter-Device Scaling

Intel packages multiple computing tiles on a single board for scalable performance [17]. At this time, DPC++ does not implicitly support the multi-tile submission. As such, the workloads cannot automatically be distributed over all the computing units of a multi-tile Intel GPU. Therefore, we explicitly submit workloads to the multi-tile device through multiple queues. We refer a reader to [98] for more details of the DPC++ multi-queue submission.

3.4 Evaluation

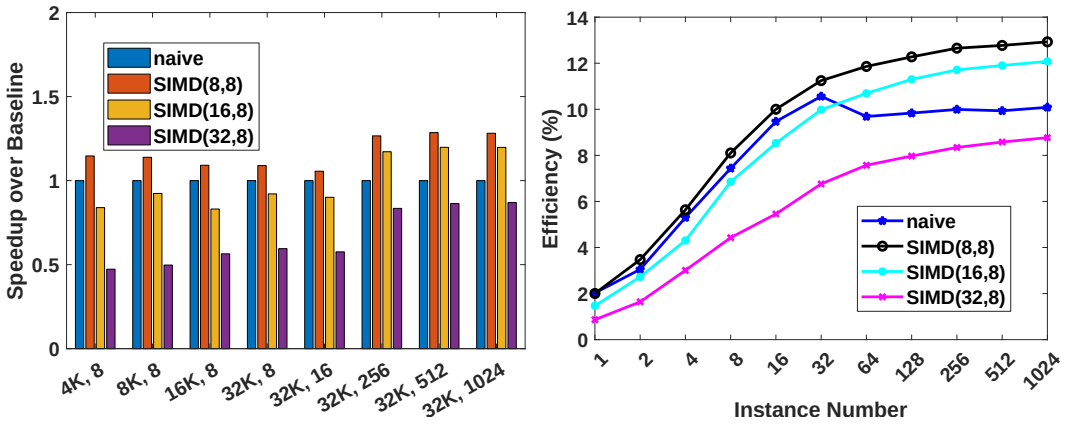
We evaluate our optimizations on two Intel GPUs with the latest microarchitecture. Both of them are pre-released models of Intel Xe GPUs. Due to confidentiality requirements, at this time, we do not disclose hardware specifications of these GPUs. For the same purpose, we present performance data by showing normalized execution time rather than the absolute elapsed time or showing performance in the unit of GOPS. The first Intel GPU, denoted as Device1 in following discussions, is a multi-tile GPU while the second Intel GPU, Device2, is a single-tile GPU. We utilize up to 2 tiles in the multi-tile Device1 for performance benchmarking and efficiency estimation. Both GPU devices are connected with 24-core Intel Icelake server CPUs, whose boost frequency is up to 4 GHz. The associated main memory systems are both 128GB at 3200 MHz. We compile programs using Intel Data Parallel C++ (DPC++) Compiler `2021.3.0` with the optimization flag `-O3`.

3.4.1 Optimizing NTT on Intel GPUs

Figure 3.5 shows that NTT accounts for significant ratios regarding the total execution time of HE evaluation routines. Therefore, we start optimizations from this decisive algorithm.

First Trial: Optimizing NTT Using SLM and SIMD

Figure 3.12 (a) compares the speedup of our first batch of NTT trials using the staged NTT implementation over the naive GPU implementation of NTT described in



(a) Speedup (b) Efficiency of 32K-point NTT

Figure 3.12: Radix-2 NTT with SLM and SIMD on Device1

Figure 3.6. We use SIMD(`TER_SIMD_GAP_SZ`, `SIMD_WIDTH`) to denote the different implementation variants. Here `TER_SIMD_GAP_SZ` refers to the switching threshold from SLM to SIMD shuffling for data exchanging among work-items. The number of register slots for each work-item can be computed by dividing `TER_SIMD_GAP_SZ` over `SIMD_WIDTH`. For example, each work-item holds a pair of NTT elements in registers for SIMD(8,8), and 4 pairs of NTT elements for SIMD(32,8). With the shared local memory as well as the SIMD shuffling for data exchanging among work-items included, we observe that SIMD(8,8) is faster than the baseline by up to 28%. Meanwhile, SIMD(16,8) is slightly slowed down compared with SIMD(8,8) but remains up to 19% faster than baseline. This indicates that the non-negligible cost of SIMD shuffling leads to the unfavorable performance. Accordingly, SIMD(32,8), which more aggressively performs SIMD shuffling and in-register data exchange than previous two variants, becomes even slower than the baseline.

Figure 3.12 (a) compares the efficiency of each NTT variant on Device1. The efficiency is computed by dividing the performance of each NTT implementation over the

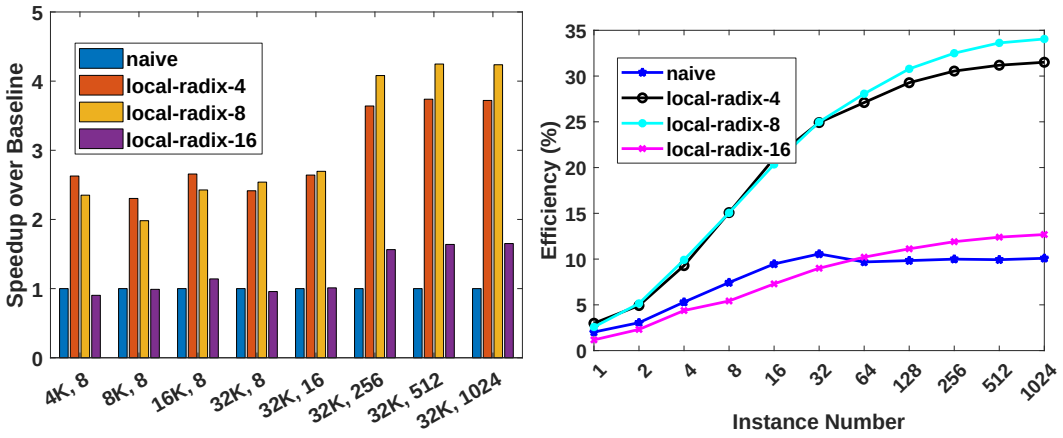
computed int64 peak performance, both in the unit of GFLOPS. The naive NTT reaches only 10.08% of the peak performance for a 32K-point NTT with 1024 instances executed simultaneously. The best one, SIMD(8,8) obtains an efficiency up to 12.93%. Since SIMD shuffling together with the SLM data communication fail to provide us with a high efficiency, we deduce that the cost of data communication is so high that radix-2 NTT cannot fully utilize the device.

Second Trial: Optimizing High-Radix NTT Using SLM

Figure 3.13 (a) compares the speedup of high-radix NTT implementations with shared local memory against the naive GPU baseline. High-radix NTT implementations re-use more data at the register-level, reducing the communication among work-items through either global memory or SLM. With the shared local memory also included, this time we obtain an up to 4.23X acceleration over the naive baseline. In Figure 3.13 (b), we see that the efficiency reaches its optimum, 34.1% of the peak performance, at radix-8 NTT with 1024 instances instantiated for 32K-point NTT computations. The radix-16 NTT, though it brings more aggressive register-level data re-use and requires less data exchange among work-items, leads to the register spilling issue so its performance becomes significantly slower than radix-8 NTT.

Assembly-Level Optimizations - `add_mod/mul64`

We further introduce assembly-level optimizations to improve the speed of the `add_mod` and `mul64` ops. As shown in Figure 3.14 (a), these low-level optimizations improve

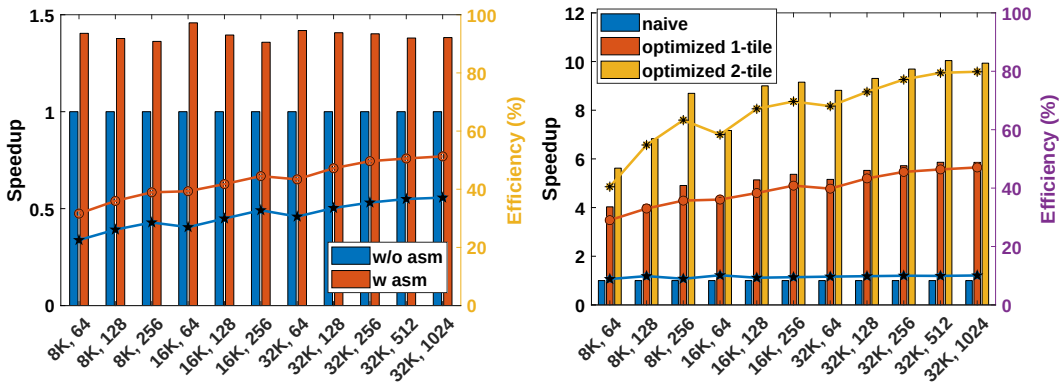


(a) Speedup (b) Efficiency

Figure 3.13: High-radix NTT with SLM on Device1

the NTT performance by 35.8% - 40.7%, increasing the efficiency of our radix-8 SLM NTT to 47.1%. The inline assembly low-level optimization provides a relatively stable acceleration percentage for different NTT sizes and instance numbers. This is because assembly-level optimization directly improves the clock cycle of the each int64 multiplication and modular addition operation, which is independent of the number of active EUs at runtime.

Explicit Dual-Tile Submission Through DPC++



(a) NTT with inline assembly

(b) NTT with explicit dual tile

Figure 3.14: NTT with inline-asm and multi-tile on Device1

With the low-level optimization, we observe that our NTT saturates only up to 47.1%, less than half of the peak performance. We observe this low efficiency because DPC++ runtime does not implicitly support multi-tile execution such that only half of the machine has been utilized. To address this issue, we explicitly submit workloads through multiple queues to enable a full utilization of our multi-tile GPU and manage to reach 79.8% of the peak performance. Meanwhile, our most optimized NTT is 9.93-fold faster than the naive baseline for the 32K-point, 1024-instance batched NTT.

3.4.2 Roofline Analysis for NTT

The most naive NTT needs to access the global memory for *each* round of the NTT computation. Therefore, its total memory access number can be computed as $2N\log_2(N)$. Here we multiply it by 2 because of both load and store operations at each round of NTT. We do not count the memory access of last round NTT processing to simplify the analysis.

Table 3.1. Number of 64-bit integer ALU operations of each work-item per round for NTT.

	64-bit int ops / round		
	other	butterfly	total
radix-2	20	28	48
radix-4	45	112	157
radix-8	120	336	456
radix-16	260	896	1156

Table 3.1 summarizes the number of ALU operations for each NTT variant. *Butterfly* refers to the ALU operations for the NTT butterfly computations while *other* denotes other necessary ALU operations such as index and address pointer computations. The radix-2 NTT performs 48 integer operations for each work-item in a single round of NTT, indicating that the naive NTT consumes $N/2 \cdot 48 \cdot \log_2(N)$ ALU operations throughout the whole computation process. Further dividing the total ALU number over the total memory access number, one can find that the operational density of naive NTT is equal to 1.5 for int64 NTT. This low operational density, as plotted in Figure 3.15, suggests that the naive NTT implementation is bounded by the global memory bandwidth and can never reach the int64 peak performance.

For the high-radix NTT, such as the radix-8 implementation for a 32K-point NTT, we first perform one round of radix-8 NTT to reduce the data exchanging gap size from 16K to 2K. After the first kernel stores the data to the global memory, another kernel is launched to compute the remaining rounds of NTT operations, where all the work-

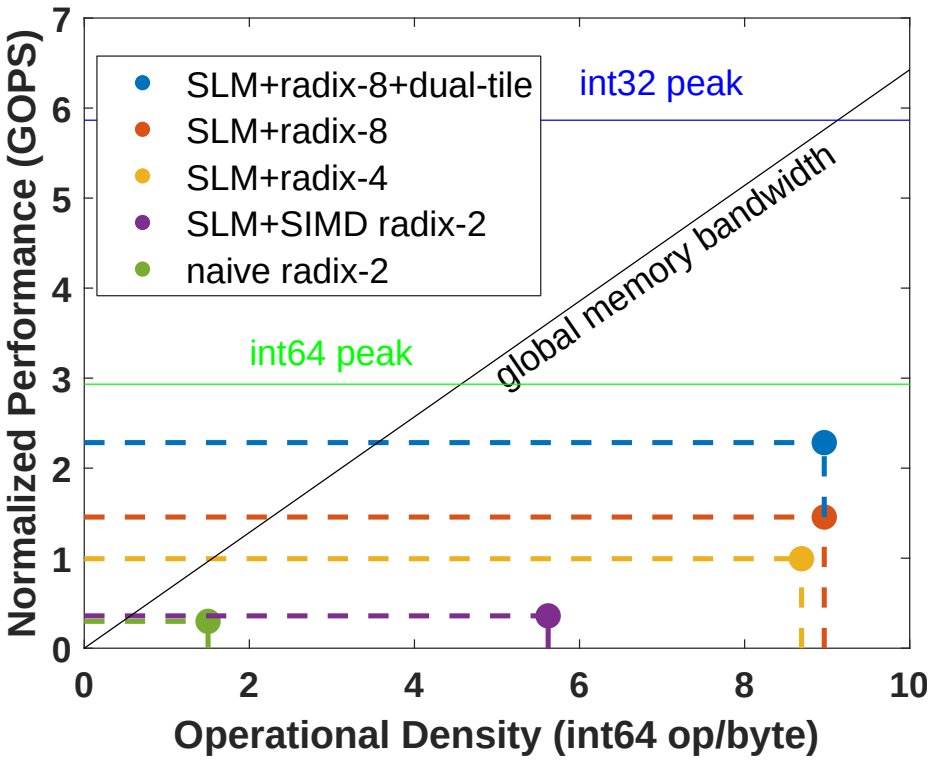


Figure 3.15: Roofline Analysis on Device1

groups hold 4K NTT sequence elements in the shared local memory for NTT operations. Therefore, we need only two rounds of global memory access for an instance of 32K-point NTT computation. Considering that its total ALU operation count equals to 456 ALU operations/round $\times \log_8(N)$ rounds — $456 \times \log_8(N)$, one can compute that the operational density of shared local memory radix-8 NTT equals 8.9, pushing the overall performance to the limits of int64 ALU throughput on Device1. The operational density of other NTT variants are computed similarly.

It is worth mentioning that a sound operational density with respect to the global memory access does not guarantee satisfactory overall performance. Although the staged radix-2 NTT with SLM and SIMD shuffling is no longer bounded by the global memory bandwidth, its practical efficiency remains far from the green line — int64 peak performance.

According to the Figure 3.15, we conclude that radix-8 shared local memory NTT with last round kernel fusion enables a sufficient operational density, which allows the performance to be shifted from memory bound to compute bound. Additionally, the shared local memory utilization together with the low-level optimization for int64 multiplication and DPC++ multi-tile submission, pushes the performance of radix-8 NTT to the ceiling of int64 ALU throughput on Device1.

3.4.3 Benchmarking for CKKS HE Evaluation Routines

Figure 3.16 benchmarks the performance of five basic HE evaluation routines under the CKKS scheme on Device1. Here *MulLin* denotes a multiplication followed by a relinearization; *MulLinRS* denotes a multiplication followed by relinearization and rescaling. Relinearization decreases the length of a ciphertext back to 2 after a multiplication. Rescaling is a necessary step for multiplication operation with the goal to keep the scale constant and reduce the noise present in the ciphertext. In addition, *SqrLinRS* refers to a ciphertext square computation with relinearization and rescaling followed. *MulLinRSModSwAdd* computes a ciphertext multiplication, then relinearizes and rescales it. After this, we switch the ciphertext modulus from $(q_1, q_2, ..., q_L)$ down to $(q_1, q_2, ..., q_{L-1})$ and scale down the message accordingly. Finally this scaled messaged is added with another ciphertext. The last benchmarked routine, *Rotate*, rotates a plaintext vector cyclically. We count the GPU kernel time exclusively for routine-level benchmarks. All evaluated ciphertexts are represented as tuples of vectors in $\mathbb{Z}_{q_L}^N$ where $N = 32\text{K}$ and the RNS size is $L = 8$.

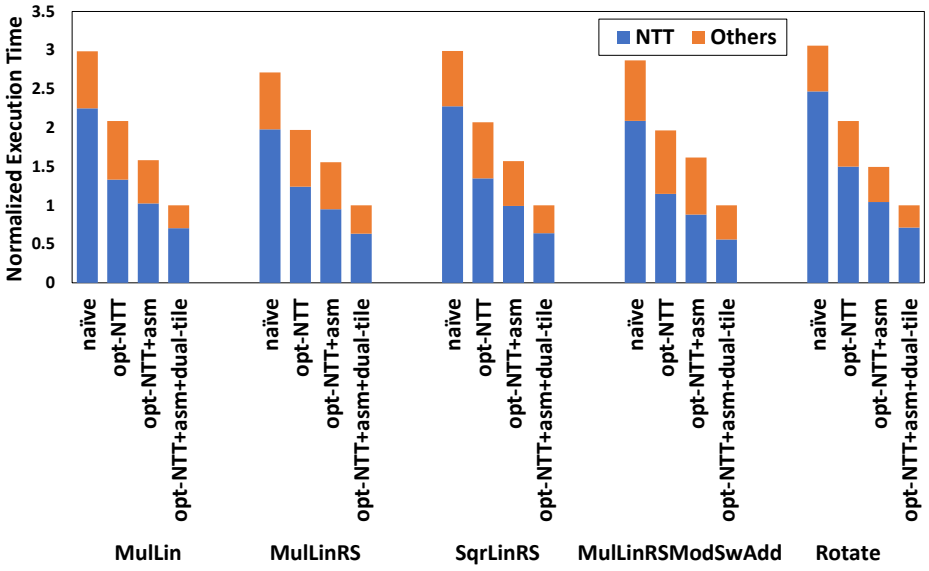


Figure 3.16: Benchmarking HE evaluation routines on Device1.

We present the impact of NTT optimizations to HE evaluation routines in four steps. We first substitute the naive NTT with our radix-8 NTT with SLM. We then employ assembly-level optimizations to accelerate the clock cycle of int64 *add_mod* and *mul_mod*. Finally we enable implicit dual-tile submission through the OpenCL backend of DPC++ to fully utilize our wide GPU. The baseline is the naive GPU implementation where no presented optimizations are adopted for the comparison purpose.

The radix-8 NTT with data communications through SLM improves the routine performance by 43.5% in average. It is worth mentioning that we do not benchmark batched routines and our wide GPU is not fully utilized such that the NTT acceleration is not as dramatic as the results in previous sections. The inline assembly optimization provides a further average 27.4% improvement in compared with the previous step. Meanwhile, the non-NTT computations show less sensitivity to the inline assembly optimization than the NTT because their computations are typically not as compute-intensive as NTT. We finally

submit the kernels through multiple queues to enable the full utilization of our multi-tile GPUs, further improving the performance by 49.5% to 78.2% from the previous step, up to 3.05X faster than the baseline.

3.4.4 Benchmarking on Device2

In addition to Device 1, a high-end multi-tile GPU, we benchmark our optimizations on another GPU, Device2, which is a single-tile GPU consisting of fewer EUs than Device1. Similar to the results obtained on Device1, the naive radix-2 NTT starts at a ∼15% efficiency of the peak performance, while the shared local memory SIMD implementation either fails to provide significant improvement, but reaches only 20.95%-24.21% efficiencies. After adopting the radix-8 shared local memory implementation, we manage to obtain up to 66.8% of the peak performance, where we are up to 5.47X faster than the baseline at this step. Since Device2 is a single-tile GPU, our final optimization here is to introduce inline assembly to optimize `add_mod` and `mul64`. Further improving the performance by 28.48% in average from the previous step, we reach an up to 85.75% of the peak performance, 7.02X faster than the baseline for 32K-point, 1024-instance NTT.

Figure 3.18 benchmarks the normalized execution time of HE evaluation routines on Device2. SIMD(8,8) denotes the radix-2 NTT with data exchanging through SLM and SIMD shuffling, where each work-item holds one slot of NTT elements in registers. opt-NTT refers to the optimal NTT variant, radix-8 NTT with data exchanging through SLM, shown in Figure 3.17. The last step is to further employ inline assembly to optimize modular addition and modular multiplication from instruction level. When substituting the naive

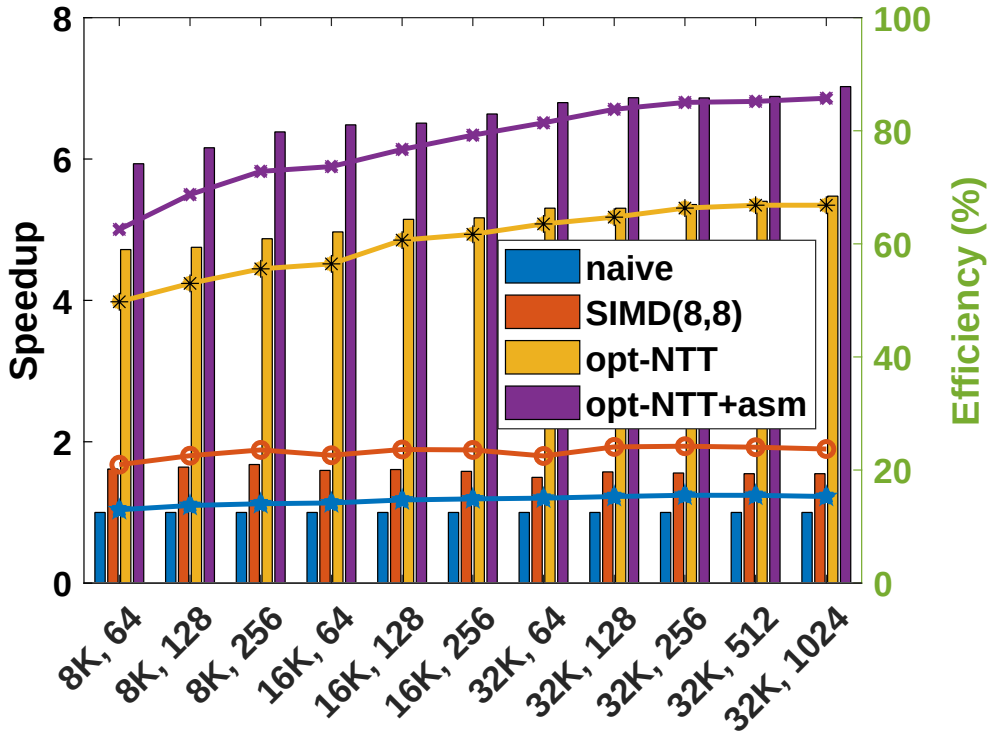


Figure 3.17: Benchmark for NTT on Device2.

NTT using SIMD(8,8) NTT, we observe the execution time of the NTT part is improved by 34% in average while the overall routines are accelerated by 29.6%. When switching to our optimal NTT variant, we observe the overall performance becomes faster than the baseline by 1.92X in average. Further enabling assembly-level optimizations, we manage to reach 2.32X - 2.41X acceleration for all five HE evaluation routines on this single-tile Intel GPU.

3.4.5 Benchmarks for Polynomial Matrix Multiplication

Besides the algorithmic level optimizations, we also demonstrate our instruction-level and application-level optimizations, which are modulus fusion, inline assembly for HE arithmetic operations and memory cache using a representative application of HE, encrypted element-wise polynomial matrix multiplication. In Figure 3.19, `matMul_mxnxk` denotes a

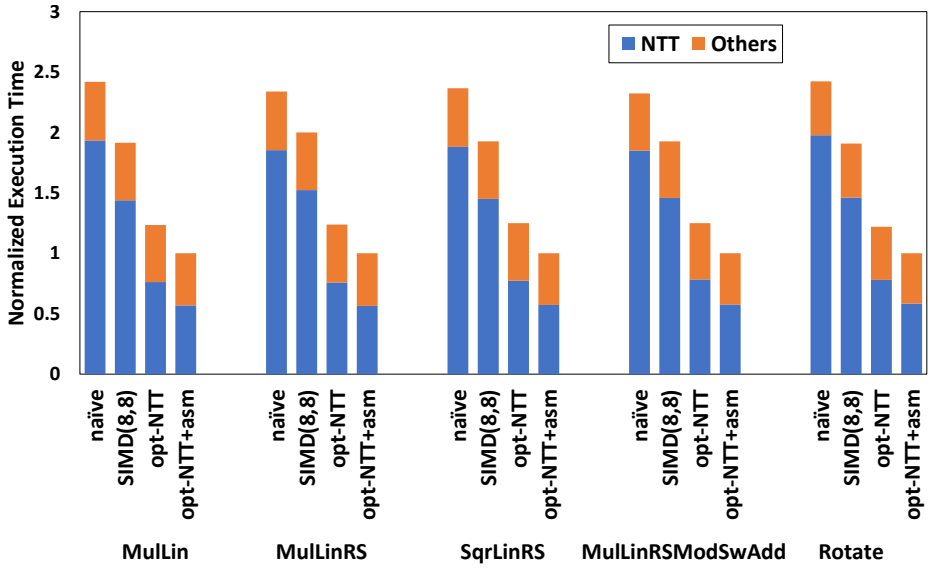
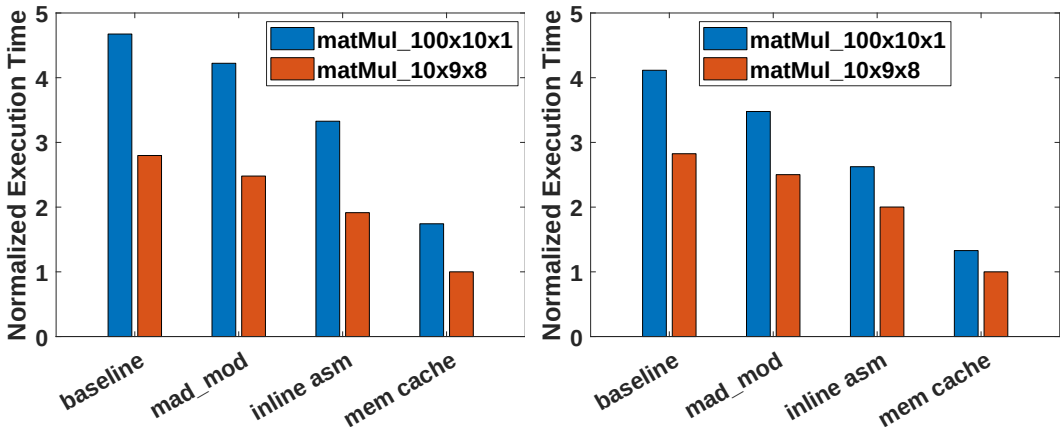


Figure 3.18: Benchmarking HE evaluation routines on Device2.

matrix multiplication $C+=A*B$, where C is m-by-n, A is m-by-k and B is k-by-n. Each matrix element is an 8K-element plaintext polynomial so each *element-wise* multiplication of `matMul` is a polynomial multiplication. Modulo operations are always applied at the end of each multiply or addition between polynomial elements. Before starting `matMul`, we need to allocate memory, initialize, encode and encrypt input sequences. Once `matMul` is completed, we decode and decrypt the computing results. We measure the elapsed time for this whole process.

Figure 3.19 compares our instruction-level and application-level optimizations for `matMul` on both Device1 and Device2. The fused `mad_mod` and inline assembly accelerate the both 100x100x1 and 10x9x8 polynomial matrix multiplications by 11.8% and 28.2%, respectively, in average on Device1. With the memory cache introduced, both `matMul` applications are further improved by ~90%. On Device1, our all-together systematic optimizations accelerate `matMul_100x10x1` and `matMul_10x9x8` by 2.68X and 2.79X, respectively. In regards



(a) Device 1 (b) Device 2

Figure 3.19: Element-wise polynomial multiplication.

to Device2, we observe a similar trend. These three optimizations together provide us with 3.11X and 2.82X acceleration for two `matMul` tests over the baseline on this smaller GPU.

3.5 Conclusions

In this chapter, we design and develop the first-ever SYCL-based GPU backend for Microsoft SEAL APIs. We accelerate our HE library for Intel GPUs spanning assembly level, algorithmic level and application level optimizations. Our optimized NTT is faster than the naive GPU implementation by 9.93X, reaching up to 85.1% of the peak performance. In addition, we obtain up to 3.11X accelerations for HE evaluation routines and the element-wise polynomial matrix multiplication application. Future work will focus on extending our HE library to multi-GPU and heterogeneous platforms.

or its subsidiaries. Other names and brands may be claimed as the property of others. No product or component can be absolutely secure. Performance/Benchmarking Disclaimer: Performance varies by use, configuration and other factors. Learn more at www.intel.com/performanceindex. Intel technologies may require enabled hardware, software or service activation. Your results may vary. No product or component can be absolutely secure. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates.

Chapter 4

ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs

4.1 Introduction

The transformer model [182]is a proven effective architecture widely used in a variety of deep learning (DL) applications, such as language modeling [55, 201], neural machine translation [61, 182] and recommendation systems [33, 172]. The last decade has witnessed rapid developments in natural language processing (NLP) pre-training models based on the transformer model, such as Seq2seq [182], GPT-2 [150], XLNET [201] and ChatGPT [141], which have also greatly accelerated the progress of NLP. Of all the pre-training models based on transformers, Bidirectional Encoder Representations from Transformers (BERT),

proposed in 2018 [55], is arguably the most seminal, inspiring a series of subsequent works and outperforming reference models on a dozen NLP tasks at the time of creation.

BERT-like models consume increasingly larger parameter space and correspondingly more computational resources. When BERT was discovered, a large model required 340 million parameters [205], but currently a full GPT-3 model requires 170 billion parameters [22]. The base BERT model requires 6.9 billion floating-point operations to inference a 40-word sentence, and this number increases to 20 billion when translating a 20-word sentence using a base Seq2Seq model [63]. The size of the parameter space and the computational demands increase the cost of the training and inference for BERT-like models, which requires the attention of the DL community in order to accelerate these models.

To exploit hardware efficiency, DL frameworks adopt a batching strategy, where multiple batches are executed concurrently. Since batched execution requires task shapes in different batches to be identical, DL frameworks presume fixed-length inputs when designing the software [151, 152, 163, 188]. However, this assumption cannot always hold, because transformer models are often faced with variable-length input problems [63,205]. In order to deploy models with variable-length inputs directly to conventional frameworks that support only fixed-length models, a straightforward solution is to pad all sequences with zeros to the maximal sequence length. However, this immediately brings in redundant computations on wasted padded tokens. These padded zeros also introduce significant memory overhead that can hinder a large transformer model from being efficiently deployed.

Existing popular DL frameworks, such as Google TensorFlow with XLA [3, 72], Meta PyTorch with JIT [145], and OctoML TVM [48], leverage the domain-specific just-

in-time compilation technique to boost performance. Another widely-adopted strategy to generate low-level performance optimization is delicate manual tuning: NVIDIA TensorRT [131], a DL runtime, falls into this category. Yet all of these frameworks require the input sequence lengths to be identical to exploit the speedup of batch processing. To lift the restriction on fixed sequence lengths, Tencent [63] and Baidu [205] provide explicit support for models with variable sequence lengths. They group sequences with similar lengths before launching batched kernels to minimize the padding overhead. However, this proactive grouping approach still introduces irremovable padding overhead when grouping and padding sequences with similar yet different lengths.

In contrast to training processes that can be computed offline, the inference stage of a serving system must be processed online with low latency, which imposes high performance requirements on DL frameworks. A highly efficient DL inference framework for NLP models requires delicate kernel-level optimizations and explicit end-to-end designs to avoid wasted computations on zero tokens when handling variable-length inputs. However, existing DL frameworks do not meet these expectations. In order to remedy this deficit, we present ByteTransformer, a highly efficient transformer framework optimized for variable-length inputs in NLP problems. We not only design an algorithm that frees the entire transformer of padding when dealing with variable-length sequences, but also provide a set of hand-tuned fused GPU kernels to minimize the cost of accessing GPU global memory. More specifically, our contributions include:

- We design and develop ByteTransformer, a high-performance GPU-accelerated transformer optimized for variable-length inputs. ByteTransformer has been deployed to serve world-class applications including TikTok and Douyin of ByteDance.

- We propose a padding-free algorithm that packs the input tensor with variable-length sequences and calculates the positioning offset vector for all transformer operations to index, which keeps the whole transformer pipeline free from padding and calculations on zero tokens.

- We propose a fused Multi-Head Attention (MHA) to alleviate the memory overhead of the intermediate matrix, which is quadratic to the sequence length, in MHA without introducing redundant calculations due to padding for variable-length inputs. Part of our fused MHA has been deployed in the production code base of NVIDIA CUTLASS.

- We hand-tune the memory footprints of layer normalization, adding bias and activation to squeeze the final performance of the system.

- We benchmark the performance of ByteTransformer on an NVIDIA A100 GPU for forward pass of BERT-like transformers, including BERT, ALBERT, DistilBERT, and DeBERTa. Experimental results demonstrate our fused MHA outperforms standard PyTorch attention by 6.13X. Regarding the end-to-end performance of standard BERT transformer, ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed and NVIDIA FasterTransformer by 87%, 131%, 138%, 74%, and 55%, respectively.

The rest of the chapter is organized as follows: we introduce background and related works in Section 4.2, and then detail our systematic optimization approach in Section 4.3. Evaluation results are given in Section 4.4. We conclude our chapter and present future work in Section 4.5.

4.2 Background and Related Works

We provide an overview of the transformer model, including its encoder-decoder architecture and multi-head attention layer. We also survey related works on DL framework acceleration.

4.2.1 The Transformer Architecture

Figure 4.1 shows the encoder-decoder model architecture of the transformer. It consists of stacks of multiple encoder and decoder layers. In an encoder layer, there is a multi-head attention layer followed by a feed-forward network (FFN) layer. A layer normalization (layernorm) operation is applied after both MHA and FFN. In a decoder layer, there are two sets of consecutive MHA layers and one FFN layer, and each operation is normalized with a layernorm. The FFN is used to improve the capacity of the model. In practice, FFN is implemented by multiplying the tensor by a larger scaled tensor using GEMM. Here we skip the embedding descriptions in the figure, and refer an interested reader to [182] for details. Although we show both encoder and decoder modules for this transformer, a BERT transformer model only contains the encoder section [55]. In this chapter, we present optimizations for BERT-like transformer models.

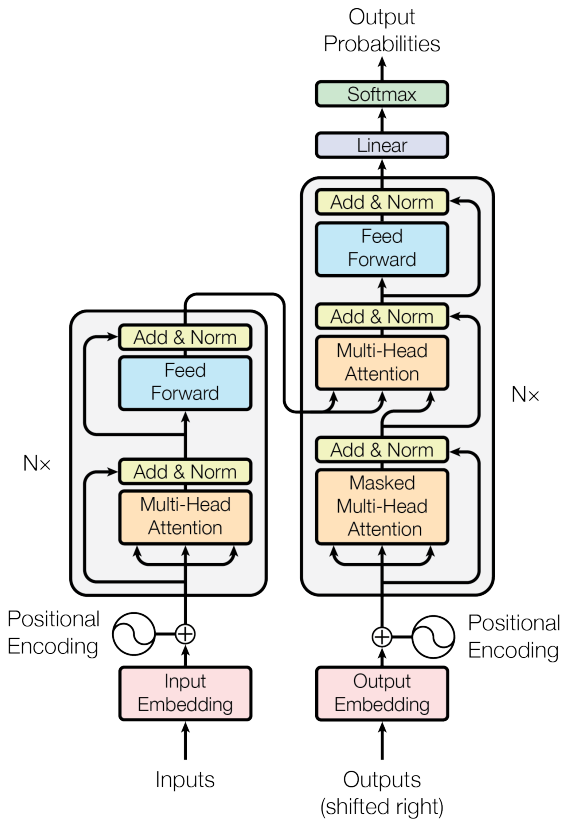


Figure 4.1: The transformer architecture [182]

Self-attention is a key module of the transformer architecture. Conceptually, self-attention computes the significance of each position of the input sequence, with the information from other positions considered. A self-attention receives three input tensors: query (Q), key (K), and value (V). Self-attention can be split into multiple heads. The Q and K tensors are first multiplied (1^{st} GEMM) to compute the dot product of the query against all keys. This dot product is then scaled by the hidden dimension d_k and passed through a softmax function to calculate the weights corresponding to the value tensor. Each head of the output tensor is concatenated before going through another linear layer by multiplying against tensor V (2^{nd} GEMM). Expressing self-attention as a mathematical formula, we have:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}}) \times V \tag{4.1}$$

4.2.2 Related Works on DL Acceleration

Performance is a crucial aspect in the real-world deployment of software systems, attracting significant attention across various applications [206, 208, 210], including DL frameworks. The conventional DL frameworks, such as PyTorch, TensorFlow, TVM, and TensorRT are designed explicitly for fixed-length input tensors. When dealing with NLP problems with variable-length input, all sequences are padded to the maximal length, which leads to significant wasted calculations on zero tokens. A few DL frameworks, such as Tencent TurboTransformer [63] and NVIDIA FasterTransformer [134], employ explicit designs for variable-length inputs. TurboTransformer designs run-time algorithms to group and pad sequences with similar lengths to minimize the padding overhead. TurboTransformer also uses a run-time memory scheduling strategy to improve end-to-end performance. Kernel-

level optimizations are of the same significance as algorithmic optimizations. NVIDIA's FasterTransformer uses vendor-specific libraries such as TensorRT and cuBLAS [132] as its back-end, which provide optimized implementations of various operations at the kernel level.

Other end-to-end DL frameworks have also presented optimizations for BERT-like transformers, such as E.T. [34] and DeepSpeed-Inference [7]. E.T. introduces a novel MHA architecture for NVIDIA Volta GPUs and includes pruning designs for end-to-end transformer models. In contrast, ByteTransformer targets unpruned models and is optimized for NVIDIA Ampere GPUs. DeepSpeed-Inference is optimized for large distributed models on multiple GPUs, while ByteTransformer currently focuses on lighter single-GPU models.

In addition to end-to-end performance acceleration, the research community has also made focused efforts to improve a key algorithm of the transformer, multi-head attention. PyTorch provides a standard implementation of MHA [149]. NVIDIA TensorRT utilizes a fused MHA for short sequences with lengths up to 512, as described in [133]. To handle longer sequences, FlashAttention was proposed by Stanford researchers in [54]. FlashAttention assigns the workload of a whole attention unit to a single threadblock (CTA). However, this approach can result in underutilization on wide GPUs when there are not enough attention units assigned. Our fused MHA, on the other hand, provides high performance for both short and long sequences for variable-length inputs without leading to performance degradation in small-batch scenarios.

Table 4.1 surveys state-of-the-art transformers. TensorFlow and PyTorch provide tuned kernels but require padding for variable-length inputs. NVIDIA FasterTransformer

Table 4.1. Summarizing state-of-the-art transformers.

	variable-len support	kernel tuning	fused MHA	kernel fusion
Tensorflow XLA	no	yes	no	no
PyTorch JIT	no	yes	no	no
FasterTransformer	yes	yes	≤ 512	no
TurboTransformer	yes	yes	no	partially
ByteTransformer	**yes**	**yes**	**yes**	**yes**

and Tencent TurboTransformer, although providing support for variable-length inputs, do not perform comprehensive kernel fusion or explicit optimization for the hot-spot algorithm MHA for any length of sequence. In addition, TurboTransformer only optimizes part of the fusible operations in the transformer model, such as layernorm and activation, namely 'partial kernel fusion' in the table. Our ByteTransformer, in contrast, starting with a systemic profiling to locate bottleneck algorithms, precisely tunes a series of kernels including the key algorithm MHA. We also propose a padding-free algorithm which completely removes redundant calculations for variable-length inputs from the entire transformer.

4.3 Designs and Optimizations

In this section, we present our algorithmic and kernel-level optimizations to improve the end-to-end performance of BERT transformer under variable-length inputs.

4.3.1 Math Expression of BERT Transformer Encoder

Figure 4.2(a) illustrates the architecture of the transformer encoder. The input tensor is first processed through the BERT pipeline, where it is multiplied by a built-in attribute matrix to perform Q, K, and V positioning encoding. This operation can be implemented using three separate GEMM operations or in batch mode. Realizing that the corresponding attribute matrices to Q, K, and V are all the same shape (hidden_dim x hidden_dim), we pack them to continuous memory space and launch a single batched GEMM kernel that calculates Q, K, and V to reduce the kernel launch overhead at runtime. Bias matrices for Q, K, and V are then added to the encoded tensor, which is passed through the self-attention module. In addition to the multi-head attention module, the BERT transformer encoder includes projection, feed forward network, and layer normalization. The encoder pipeline can be represented as a series of mathematical operations, including six GEMMs (shown in light purple) and other memory-bound operations (shown in light blue).

4.3.2 Profiling for Single-Layer Standard BERT Transformer

We implement the pipeline of Figure 4.2 (a) by calling cuBLAS and profile its single-layer performance on an NVIDIA A100 GPU. We adopt the standard BERT transformer configuration (batch size: 16, head number: 12, head size: 64) and profile for two different sequence lengths: 256 and 1024.

Figure 4.3 shows the performance breakdown for two sequence lengths. `GEMM0` to `GEMM3` refer to the consecutive four GEMMs that are enumerated from GEMM #0 to

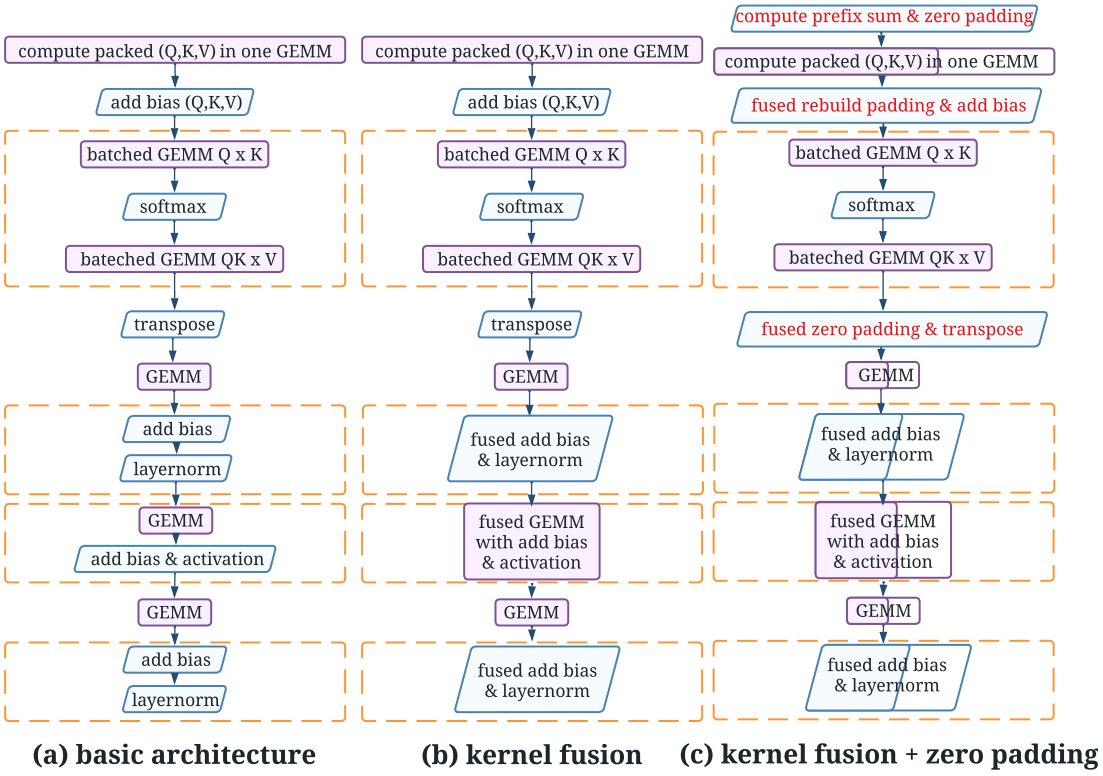
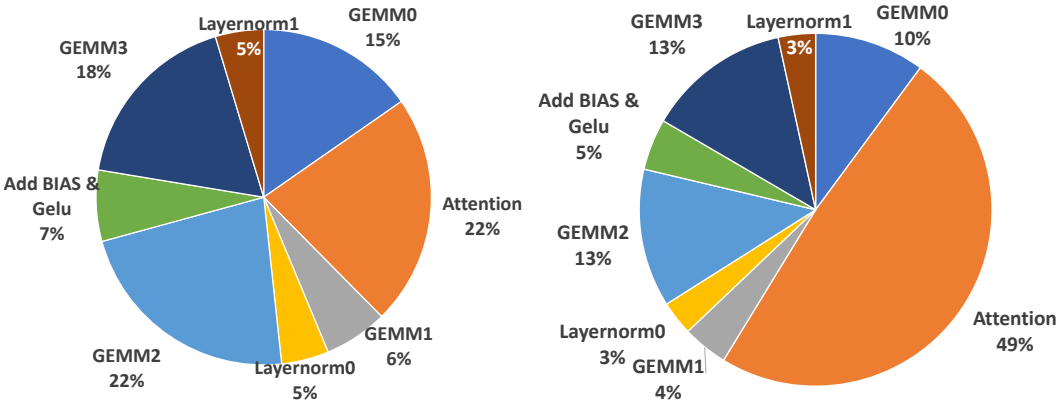


Figure 4.2: BERT transformer architecture and optimizations



(a) Sequence lengths 256

(b) Sequence lengths 1024

Figure 4.3: Performance breakdown of BERT transformer

GEMM #3 in Figure 4.2 (a). The other two batched GEMMs are part of the attention module and are therefore profiled together with the softmax as a whole, referred to as `MHA` in Figure 4.3. The two sets of "add bias and layernorm" operations are referred to as `layernorm0` and `layernorm1`. The profiling results show that the compute-bound GEMM operations account for 61% and 40% of the total execution time for both test cases. The attention module, which includes a softmax and two batched GEMMs, is the most time-consuming part of the transformer. As the sequence length increases to that of a GPT-2 model (1024), attention accounts for 49% of the total execution time, while the remaining memory-bound operations (layernorm, add bias and activation) only take up 11%-17%.

4.3.3 Fusing Memory-Bound Operations of BERT Transformer

Since cuBLAS uses architectural-aware optimizations for high performance GEMMs, presumably there remain limited opportunities for further acceleration. Therefore, we turn our eyes to optimizing the modules containing memory-bound operations, such as attention (with softmax), feed forward network (with layernorm) and add bias followed by element-wise activation. We improve these operations by fusing distinct kernels and reusing data in registers to reduce global memory access. Figure 4.2 (b) presents the BERT transformer pipeline with memory-bound kernel fusion, where we fuse layernorm and activation with their consecutive kernels.

Add Bias and Layer Normalization

These operations account for 10% and 6% of the overall execution time for sequence lengths 256 and 1024, respectively. After MHA, the result tensor needs to first be added

upon the input tensor (bias) and perform layer normalization. Here hidden dimension (`hidden_dim`) equals `head_num` $\times$ `head_size`. In standard BERT configuration, head number and head size are fixed to 12 and 64. The naive implementation introduces two rounds of memory access to load and store the tensor. We provide a fused kernel that only needs to access the global memory in one round to finish both layernorm and adding bias. Kernel fusion for this sub-kernel improves the performance by 61%, which accordingly increases the single-layer BERT transformer performance by 3.2% for sequence lengths ranging 128 to 1024 in average.

Add Bias and Activation

These operations account for 7% and 5% of the overall execution time for sequence lengths 256 and 1024, respectively. After the projection via matrix multiplication, the result tensor will be added against the input tensor and perform an element-wise activation using GELU [83]. Our fused implementation, rather than storing the GEMM output to global memory and loading it again to conduct adding bias and activation, re-uses the GEMM result matrix at the register level by implementing a customized and fused CUTLASS [135] epilogue. Experimental results validate that our fused GEMM perfectly hides the memory latency of bias and GELU into GEMM. After this step, we further improve the single-layer BERT transformer by 3.8%.

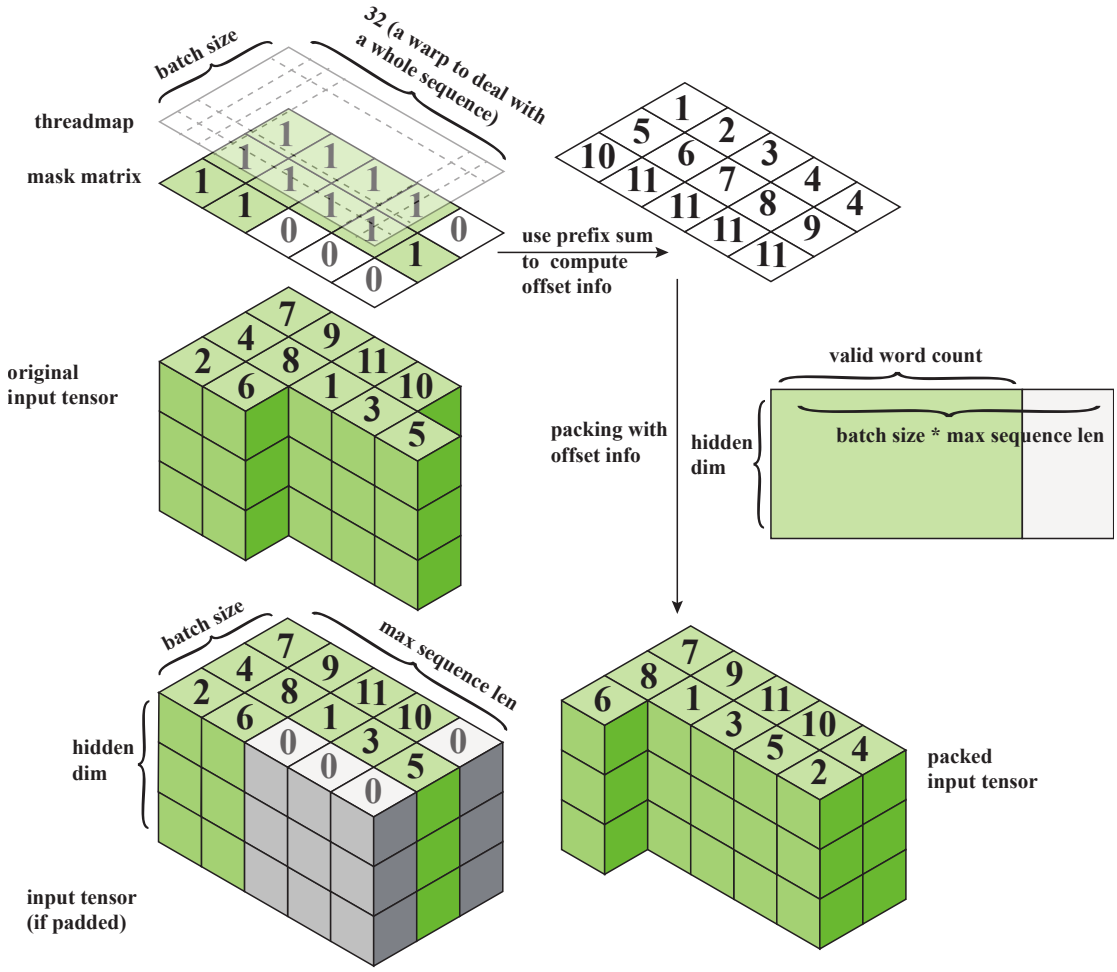


Figure 4.4: The zero padding algorithm.

4.3.4 The Zero Padding Algorithm for Variable-Length Inputs

Because the real-time serving process receives sentences with various words as input tensor, the sequence lengths can often be different among batches. For such an input tensor composed of sentences with variable lengths, the conventional solution is to pad them to the maximal sequence length with useless tokens, which leads to significant computational and memory overhead. In order to address this issue, we propose the zero padding algorithm to pack the input tensor and store the positioning information for other transformer operations to index the original sequences.

Figure 4.4 presents the details of the zero padding algorithm. We use an input tensor with 3 sentences (proceeded in 3 batches) as an example. The longest sentence contains 5 word tokens while the other two have 2 and 4 words. The height of the sample input tensor is 3, which is equal to the hidden dimension. The conventional method is to pad all sentences to the maximal sequence length by filling zeros. The elements, either 1 or 0, of the mask matrix correspond respectively to a valid token or a padded token of an input tensor with variable size. By calculating the prefix sum of the mask matrix, we can skip the padded tokens and provide the position indices of all valid tokens. We implement an efficient CUDA kernel to calculate the prefix sum and the position offset. Each warp computes the prefix sum for tokens of a whole sentence, so in total there are `batch_size` warps assigned in each threadblock for prefix sum calculation. Once the prefix sum is computed, we pack the input tensor to a continuous memory area so that the total number of words used in future calculations is reduced from `seq_len` $\times$ `batch_size` to the actual valid word count of the packed tensor.

Figure 4.2 (c) presents the detailed modifications on BERT by introducing our zero padding algorithm. Before conducting the positioning encoding, we calculate the prefix sum of the mask matrix to pack the input tensor so that we avoid computations on useless tokens in the first GEMM. Since batched GEMM in MHA requires identical problem shapes among different batches, we unpack the tensor before entering the attention module. Once MHA is completed, we pack the tensor again such that all remaining operations can benefit from the zero padding algorithm. The final result tensors are validated element-by-element against TensorFlow such that the correctness and accuracy are ensured. It is worth mentioning that padding and remove padding operations are fused with existing memory-bound footprints such as adding bias and transpose to minimize the overhead led by this feature.

Our presented padding-free algorithm is designed to ensure semantic preservation. We maintain an array that stores the mapping relationship of the valid tokens between the original tensor and the packed tensor. The transformer operates on the packed tensor, and intermediate operations, such as MHA, layernorm and activation, refer to this position array to ensure the correctness. At the end of each layer, we reconstruct the output tensor according to the position array such that the whole pipeline is semantic preserving.

Table 4.2 counts the floating point computations of a single-layer BERT transformer. The computations of memory-bound operations are not included since they are negligible compared with the listed modules. Enabling the zero padding algorithm eliminates redundant computations for all compute-intensive modules other than MHA due to the restrictions of batched GEMM. When the average sequence length is equal to 60% of the maximum, the padding-free algorithm further accelerates the BERT transformer by 24.7%.

	Baseline	Zero Padding	Zero Padding + fused MHA
GEMM0	$6mk^2$	$6(\alpha \cdot m)k^2$	$6(\alpha \cdot m)k^2$
MHA	$4\frac{m^2}{bs}k$	$4\frac{m^2}{bs}k$	$4\frac{(\alpha \cdot m)^2}{bs}k$
GEMM1	$2mk^2$	$2(\alpha \cdot m)k^2$	$2(\alpha \cdot m)k^2$
GEMM2	$8mk^2$	$8(\alpha \cdot m)k^2$	$8(\alpha \cdot m)k^2$
GEMM3	$8mk^2$	$8(\alpha \cdot m)k^2$	$8(\alpha \cdot m)k^2$

Table 4.2. The computation number needed for variable-length inputs.

4.3.5 Optimizing Multi-Head Attention

The zero-padding algorithm, although it effectively reduces wasted calculations for variable-length inputs, cannot directly benefit batched GEMM operations in MHA. This disadvantage becomes increasingly significant when the sequence length increases, as demonstrated in Table 4.2. The complexity of MHA is quadratic to the sequence length, while the complexity of all other GEMMs is linear to the sequence length. This motivates us to provide a high-performance fused MHA while maintaining the benefits of the zero-padding algorithm. With our fused MHA, attention no longer faces redundant calculations on useless tokens, as shown in Table 4.2.

Unpadded Fused MHA for Short Sequences

For short input sequences, we hold the intermediate matrix in shared memory and registers throughout the MHA computation kernel to fully eliminate the quadratic memory overhead. We also access Q, K, and V tensors according to the positioning information obtained in the prefix sum calculation step to avoid redundant calculations on padding zeros for

the MHA module. Algorithm 1 shows the pseudo code of our fused MHA for short sequences. We launch a 3-dimensional grid map: {`head_num`, `seq_len`/`split_seq_len`, `batch_size`}. Here `split_seq_len` is a user-defined parameter to determine the size of a sequence tile preceded by a threadblock (typically set to 32 or 48). The warp count of a threadblock is computed by the maximal sequence length: `split_seq_len`/16×(`seq_len`/16). Each threadblock loads a chunk of Q (`split_seq_len` × `head_size`), K (`max_seq_len` × `head_size`) and V ((`head_size` × `max_seq_len`)) into shared memory and computes MHA for a tile of the result tensor. We allocate three shared-memory buffers to hold Q, K, V sub-matrices. Due to the algorithmic nature of MHA, we can re-use K and V chunks in the same shared-memory buffer `s_kv`. The intermediate matrix of MHA is held and re-used in another pre-allocated shared-memory buffer `s_logits`.

The workflow of fused MHA for short sequences is straightforward yet efficient. Each thread first loads its own tile of Q and K into shared memory and computes GEMM for $P = Q \times K$. The element-wise adding bias and scaling operations are both fused with the load process to hide the memory latency. GEMM is computed using the CUDA `wmma` intrinsic to leverage tensor cores of NVIDIA Ampere GPUs. The intermediate matrix P is held in shared memory during the reduction. Because we explicitly design this algorithm for short sequences, each thread can load a whole sequence of P from shared memory into register files for both reduction and element-wise exponential transform in softmax. Once the softmax operation is completed, we load a K tile to shared memory to compute the second GEMM $O = P \times V$, and then store the result tensor O to the global memory.

Algorithm 1: Unpadded fused MHA for short sequences

```
/* define skew offset to avoid bank conflict */
#define SKEW_HALF 8
Shared memory:
__half s_kv [max_seq_len][size_per_head + SKEW_HALF];
__half s_query [split_seq_len][size_per_head + SKEW_HALF];
__half s_logits [max_seq_len][size_per_head + SKEW_HALF];
/* warps collaboratively fill s_query with adding bias fused */
Load __half2 q_bias
for seq_id = warp_id : warp_num : split_seq_len do
    query = Q[batch_seq_offset + seq_id + thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_query[offset] = fast_add(query, k_bias);
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 k_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    key = K[batch_seq_offset + seq_id + thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(key, k_bias);
/* compute Q*K using WMMA */
Clear wmma fragment QK to zero
for k_id = 0 : head_size / 16 do
    Load 16x16 wmma fragments of Q
    Load 16x16 wmma fragments of K
    Update QK = Q * K + QK using wmma::mma_sync
Store fragment QK to s_logits using wmma::store_matrix_sync
/* Compute softmax */
for seq_id = warp_id : warp_num : batch_seq_len do
    float logits[max_seq_len];
    each thread loads a whole sequence to fill local registers
    /* 1st round of reduction with register-level data re-use*/
    compute max_val in local registers
    /* register-level data re-use*/
    compute P = exp(P - max) and update local registers
    /* 2st round of reduction with register-level data re-use*/
    compute sum_val in local registers
    /* register-level data re-use*/
    compute P = P/sum_val and stream to s_logits
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 v_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    value = V[batch_seq_offset + seq_id + thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(value, v_bias);
/* Similar to Q * K so omitting the details here */
Compute P * V using wmma and stream to global memory
```

Unpadded Fused MHA for Long Sequences

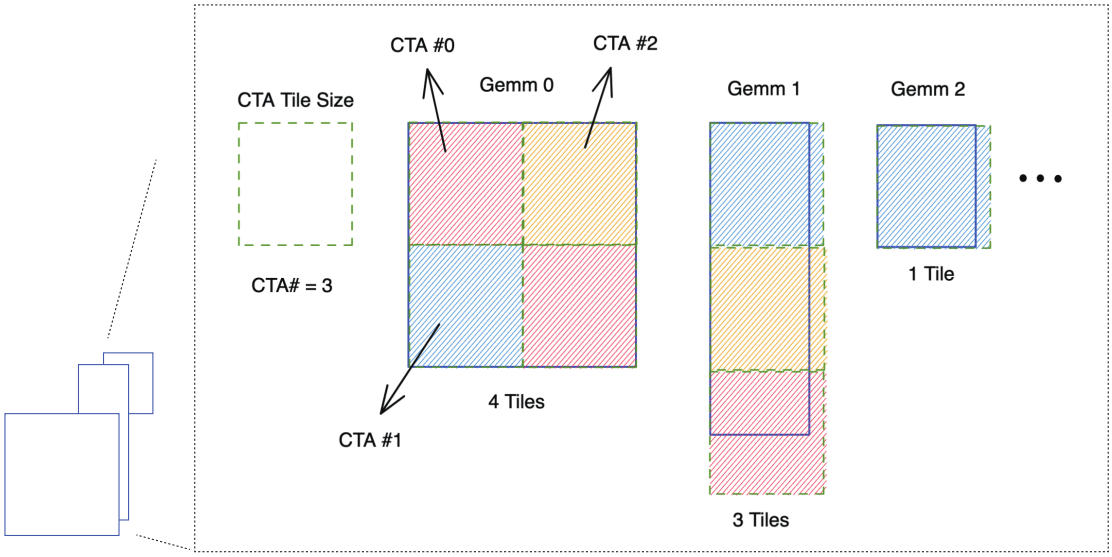


Figure 4.5: Grouped GEMM demonstration

Because of the limited resources of register files and shared memory, the previous fused MHA is no longer feasible for long sequences. Therefore, we set 384 to be the cut-off sequence length and propose a grouped GEMM based fused MHA for large models.

The Grouped GEMM idea is first presented by NVIDIA CUTLASS [135]. Different from batched GEMM, where all GEMM sub-problems are required to have an identical shape, grouped GEMM allows arbitrary shapes for sub-problems. This is enabled by a built-in scheduler that iterates over all GEMM sub-problems in a round-robin manner. Figure 4.5 demonstrates the idea of grouped GEMM using an example with 3 sub-problems. Supposing 3 threadblocks (CTAs) are launched, each CTA calculates a fix-sized CTA tile at each step until all GEMM sub-problems have been covered. GPU computes in waves, logically. In the first wave, All three CTAs calculate 3 tiles (light red, light yellow and light

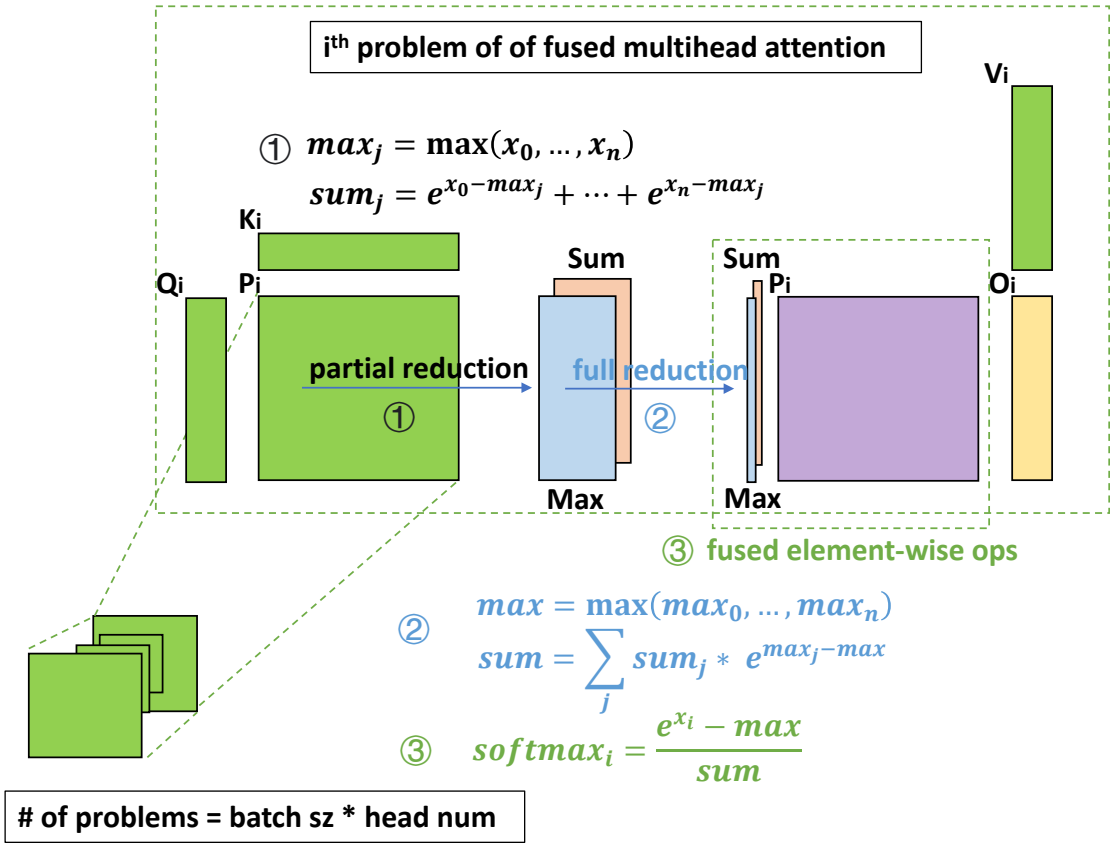


Figure 4.6: Grouped GEMM based FMHA. Source codes are available at [130].

blue in the figure). And then in the second CTA wave, CTA #0 moves to the bottom-right tile of GEMM 0 while CTA #1 and CTA #2 move to sub-problems of GEMM 1. In the final CTA wave, CTA #0 and CTA #1 continue to compute tasks in GEMM 1 and GEMM 2 while CTA #2 keeps idle because there are no more available tiles in the computational graph.

Since grouped GEMM lifts the restriction on the shape of sub-problems, it can directly benefit MHA problems with variable-length inputs. Figure 4.6 presents our grouped-GEMM-based fused MHA for long sequences. The total number of MHA problems is equal to `batch_size` $\times$ `head_num`. The MHA problems among different batches have different sequence lengths, while sequence lengths within the same batch are identical. The grouped GEMM scheduler iterates over all attention units in a round-robin manner. In each attention unit, we first compute GEMM $P_i = Q_i \times K_i$, and conduct softmax on P_i. The second GEMM $O_i = P_i \times V_i$ provides us with the final attention result. Here i indicates the i^{th} problem of grouped MHA with variable shapes. The softmax operation is fused with GEMMs to hide the memory latency. We have upstreamed the prototype of our grouped GEMM based fused MHA into NVIDIA CUTLASS [130].

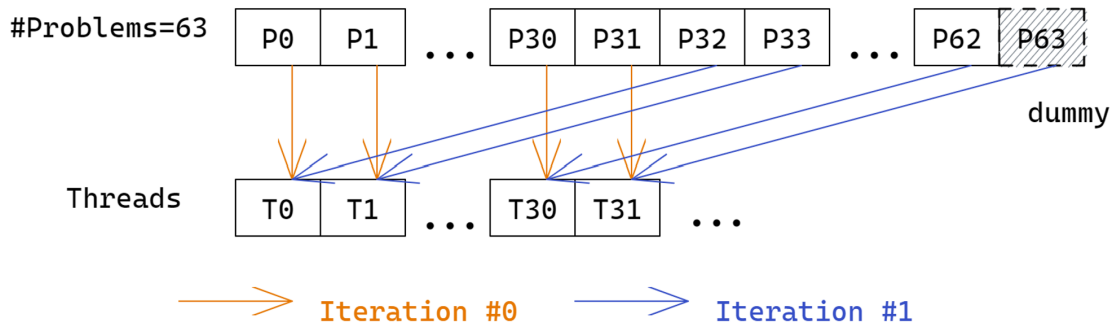


Figure 4.7: Warp prefetching for grouped GEMM

Grouped GEMM frequently checks with the built-in scheduler on the current task assignments, which leads to the runtime overhead. To address this issue, we propose an optimization over the built-in CUTLASS group GEMM scheduler. Figure 4.7 shows our optimization for the original CUTLASS grouped GEMM scheduler. Rather than asking one thread to compute the current tasks metadata, we have all 32 threads in a warp compute the tile indices to visit at one time. Therefore, we achieve 32X fewer scheduler visit overhead. In practice, this strategy brings a ∼10% improvement over the original CUTLASS grouped GEMM for standard BERT configurations. The prototype of this optimization has also been upstreamed to NVIDIA CUTLASS. We would refer an interested reader to [136] for detailed source codes.

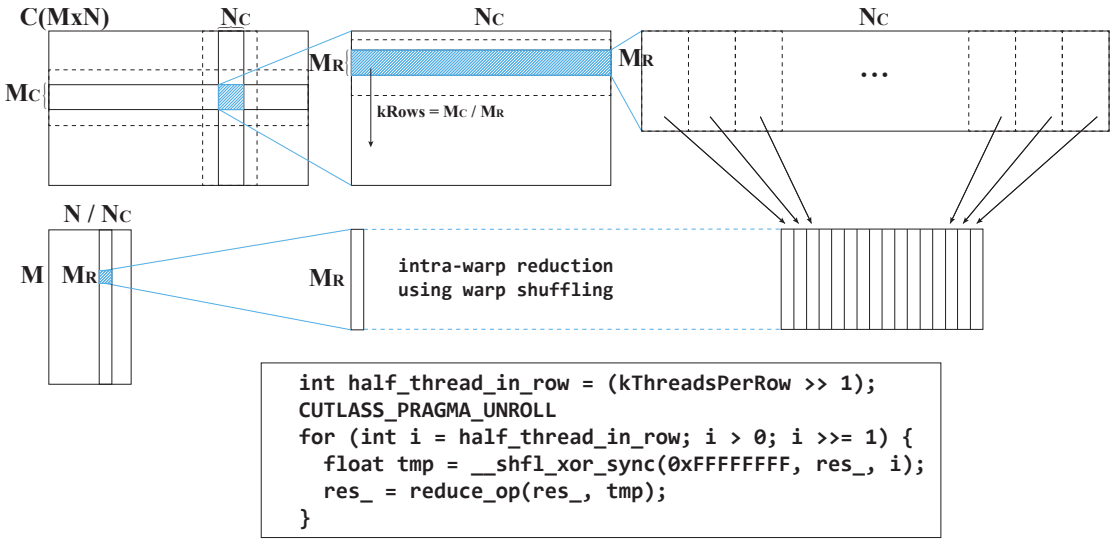


Figure 4.8: Fused softmax reduction in grouped GEMM epilogue

In addition to optimizing the grouped GEMM scheduler, we fuse the memory footprints of softmax into two grouped GEMMs of MHA. Figure 4.8 shows the details of epilogue fusion for softmax reduction. A CTA computes an $M_C \times N_C$ sub-matrix. M_C

and M_C are both set to 128 to maximize the performance of GEMM. Under the default CUTLASS threadmap assignment, there are 128 threads per CTA, and the threadmap is arranged as 8×16, where each thread holds a 128-bit register tile in each step. After the intra-thread reduction, the $M_R \times N_C$ (8×128) sub-matrix is reduced to 8×16, with one reduced result held by one thread. We then conduct an intra-warp reduction to further reduce from the column dimension, which is implemented via CUDA warp shuffling for efficiency. Similar reductions (intra-thread followed by intra-warp reduction) are performed to compute both max and sum in epilogue. Once max and sum are both reduced, we store them to global memory.

The reduction in epilogue only provides us with partial reduction within a threadblock because cross-threadblock communication is impractical under the current CUDA programming model. Hence, we need to launch a separated lightweight kernel, as shown in Figure 4.6, to conduct the full reduction. In partial reduction, the target tensor of each attention unit is `seq_len`$\times$`seq_len` while the full reduction just reduces a `seq_len`$\times$`seq_len`/128. Therefore, the workload of full reduction is negligible to that of partial reduction In practice, the full reduction kernel only accounts for $\sim 2\%$ of total execution time in fused MHA.

Once we have obtained the fully reduced *max* and *sum* vectors, we are ready to proceed element-wise transform $\frac{e^{x_{ij}} - max}{sum}$ on the first GEMM's output matrix. To hide the memory latency, we fuse these element-wise operations into the mainloop of the second GEMM. Algorithm 2 presents our modifications (marked in red) of the original CUTLASS GEMM mainloop to enable softmax fusion. The original GEMM mainloop adopts the pipelining strategy to alleviate memory access latencies on both global memory and shared

Algorithm 2: Mainloop fusion of grouped FMHA

```
Register Tiles:
WarpLoadedFragmentA warp_loaded_frag_A[2];
WarpLoadedFragmentB warp_loaded_frag_B[2];
WarpLoadedFragmentNormSum warp_loaded_frag_norm_sum;
Shared memory:  (kStages + 1) shared-memory tiles for A and B
/* prologue */
Load k-invariant fused softmax tile to warp_loaded_frag_norm_sum
Prefetch kStages - 1 tiles of A to shared memory using cp.async
Prefetch kStages - 1 tiles of B to shared memory using cp.async
Prefetch a tile of A from shared memory to warp_loaded_frag_A[0]
Prefetch a tile of B from shared memory to warp_loaded_frag_B[0]
/* fused element-wise operation */
/* A = exp(A−max) / sum */
elementwise_transform(
 warp_loaded_frag_A[0],
 warp_loaded_frag_norm_sum);
/* mainloop */
for k to -kStages + 1 do
    /* Computes a warp-level GEMM */
    /* with pipelined load during iterations */
    for warp_mma_k = 0 to kWarpGemmIterations - 1 do
        Prefetch warp_loaded_frag_A[(warp_mma_k + 1) % 2]
        Prefetch warp_loaded_frag_B[(warp_mma_k + 1) % 2]
        /* fused element-wise transform */
        elementwise_transform(
         warp_loaded_frag_A[(warp_mma_k + 1) % 2],
         warp_loaded_frag_norm_sum);
        /* Computes a warp-level GEMM*/
        /* on data loaded in previous iteration */
        warp_mma(
         accum,
         warp_loaded_frag_A[warp_mma_k % 2],
         warp_loaded_frag_B[warp_mma_k % 2],
         accum);
        Prefetch a tile of A to shared memory using cp.async
        Prefetch a tile of B to shared memory using cp.async
```

memory. For shared memory accesses, double register tiles are utilized to ensure that what is consumed in the current iteration has always been loaded in the previous iteration. For global memory accesses, a multi-stage loading strategy is employed with the help of the `cp.async` instruction of NVIDIA Ampere GPUs. The `cp.async` instruction allows loading data asynchronously from global memory to shared memory without consuming registers. Multiple such transactions can be proceeded concurrently, and a stage barrier ensures selected stages to be synchronized. The number of load stages (`kStages`) is a compile-time constant defined by a user. Similar to shared memory accesses, loading from global memory is also pipelined to overlap memory latency with computation. Therefore, `kStages` pieces of shared memory buffers are needed under the multi-stage pipeline scheme. As shown in Algorithm 2, we preload the k-invariant vectors *sum* and *max* in prologue, and conduct element-wise transform right after the matrix elements are loaded into registers. Since the fused vectors are loaded outside of the GEMM mainloop, only negligible overhead is brought into the baseline GEMM and the memory latency to perform element-wise transform is perfectly hidden with GEMM computations.

The baseline MHA is a computational chain containing a batched GEMM, a softmax, and another batched GEMM. The time and memory complexity of all these operations are quadratic in the sequence length. Because the padding-free algorithm directly reduces the effective sequence length, MHA with variable-length input also gains a direct improvement. Our fused MHA, which is explicitly designed to handle both short and long sequences, incorporates the padding-free algorithm to alleviate the memory overhead of the intermediate matrix in MHA caused by padding for variable-length inputs. Our highly

optimized MHA outperforms the standard PyTorch MHA by 6.13X and further accelerates the single-layer BERT transformer by 19% compared to the previous step. As a result, this fully optimized version surpasses the baseline implementation in Figure 4.2 (a) by 60%. Since the remaining operations of a forward BERT transformer are all near-optimal GEMM operations, we conclude our optimizations at this step.

4.4 Evaluation

We evaluate our optimizations on an NVIDIA A100 GPU. The GPU device is connected to a node with four 32-core Intel Xeon Platinum 8336C CPUs, whose boost frequency is up to 4.00 GHz. The associated CPU main memory system has a capacity of 2TB at 3200 MHz. We compile programs using CUDA `11.6u2` with the optimization flag `O3`. We compare the performance of ByteTransformer with latest versions of state-of-the-art transformers, such as TensorFlow `2.8`, PyTorch `1.13`, Tencent TurboTransformer `0.5.1`, Microsoft DeepSpeed-Inference `0.7.7`, and NVIDIA FasterTransformer `5.1`. All the tensors benchmarked in this paper, unless specified, are in the half-precision floating-point format (FP16) to leverage tensor cores of NVIDIA GPUs. The variable sequence lengths in this section are generated randomly based on a uniform distribution with a range from 1 to the maximum length. We average the reported performance data over tens of runs to minimize fluctuations.

4.4.1 Kernel Fusion for Layernorm and Add-Bias Operations

As depicted in Figure 4.2, BERT transformer is composed of a series of GEMM and memory-bound operations. Since GEMM are accelerated by near-optimal vendor's li-

braries cuBLAS and CUTLASS, we focus on optimizing the functional modules that involve memory-bound operations.

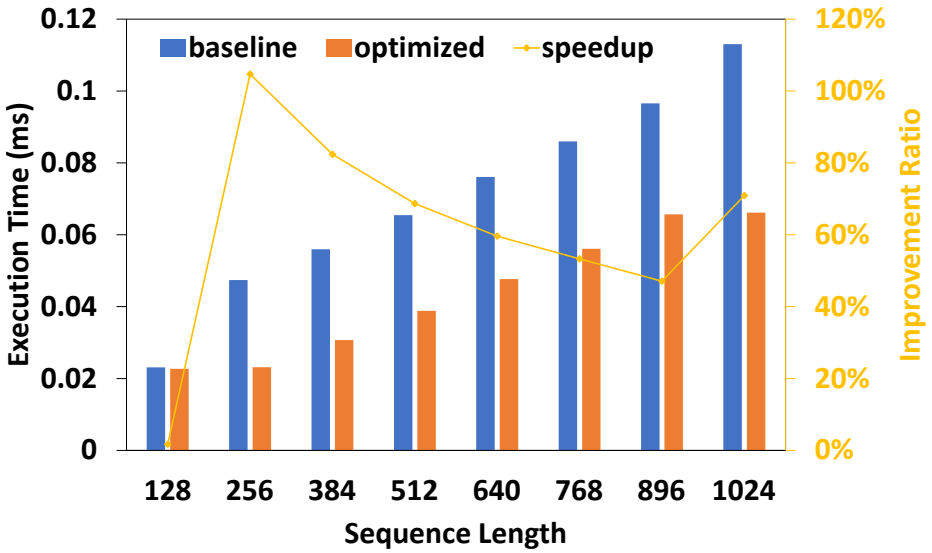


Figure 4.9: Kernel fusion for add-bias and layernorm under the standard BERT.

The result tensor needs to be added by the input tensor and normalized after projection and feed forward network of BERT transformer. Rather than launching two separated kernels, we fuse them into a single kernel and re-use data at the register level. In addition to kernel fusion, we leverage FP16 SIMD2 to increase the computational throughput of layernorm by assigning more workload to each thread. We normalize the execution time by that of the optimized layernorm and present the results in Figure 4.9: the improved version with kernel fusion provides us with a 69% improvement on average over the unfused baseline for sequence lengths ranging 128 to 1024.

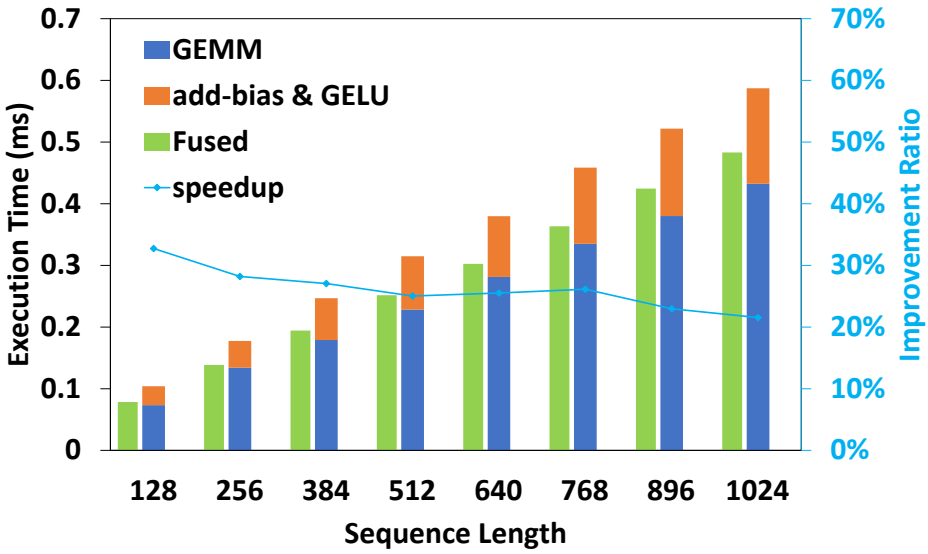


Figure 4.10: Kernel fusion for GEMM, add-bias, and GELU.

4.4.2 Kernel Fusion for GEMM and Add-Bias & Activation

Regarding the GEMM, add-bias and activation pattern in BERT transformer, we also provide a fused kernel to reduce the global memory access. An unfused implementation is to call vendor's GEMM, store the output to global memory, and then load the result matrix from global memory for further element-wise operations. In our optimized version, when the result matrix of GEMM is held in registers, we conduct fused element-wise operations that re-use data at the register level. Once the element-wise transform (add-bias and GELU) is completed, we then store the results to the global memory. Figure 4.10 compares the performance of fused and unfused versions. In each clustered bar plot, the detailed execution time breakdown of the unfused implementation, normalized by the fused execution time (shown in the left bars), is shown in the stacked bar on the right. By fusing element-wise operations into the GEMM epilogue, we improve the performance by 24% on average for sequence lengths ranging 128 to 1024. It is worth mentioning that we

feed *packed* tensors into both fused and non-fused kernels, such that the performance gain in Sec IV A and B are solely from kernel fusion.

4.4.3 Optimizing Multi-Head Attention

Figure 4.3 shows that MHA accounts for 22% - 49% of the total execution time. We optimize this key algorithm by fusing softmax into GEMMs without calculating for useless padded tokens under variable-length inputs. For short sequences, we hold the intermediate matrix in registers and shared memory. For long sequences, we adopt a grouped GEMM based fused MHA and fuse softmax operations into our customized GEMM epilogue and mainloop to hide the memory latency. In both implementations, the input matrices are accessed according to the position information obtained from the zero padding algorithm so that no redundant calculations are introduced.

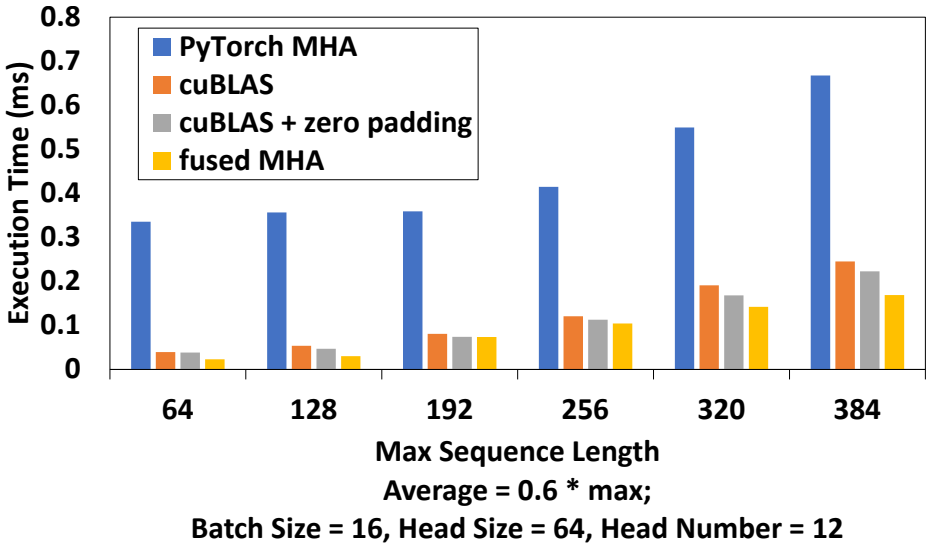


Figure 4.11: Fused MHA for short sequences.

Figure 4.11 compares the MHA performance for sequences shorter than 384. Here cuBLAS denotes the unfused implementation that calls cuBLAS for batched GEMM. The softmax operation between two batched GEMM can benefit from the zero padding algorithm, by only accessing unpadded tokens according to the known indices. This variant is denoted as *cuBLAS + zero padding* in the figure. cuBLAS batched GEMM improves the performance over stand PyTorch MHA by 5 folds while enabling the zero padding algorithm for softmax further improves the performance by 9%. Our MHA fully fuses the softmax and two batched GEMMs into one kernel, resulting in average speedups of 617%, 42%, and 30% over all three variants for variable sequence lengths ranging from 64 to 384.

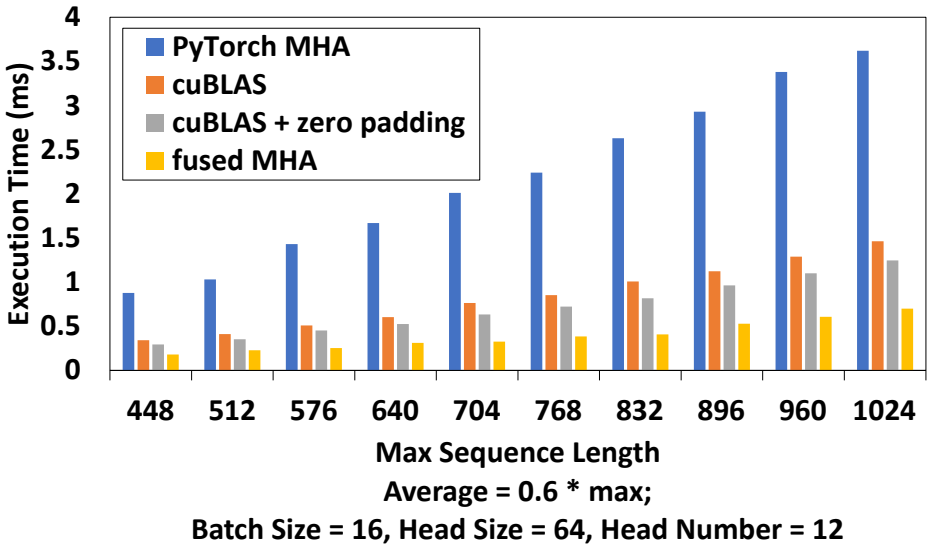


Figure 4.12: Fused MHA for long sequences

Figure 4.12 compares the performance of the MHA for sequences longer than 448. The cuBLAS batched GEMM triples the MHA performance over PyTorch, while eliminating wasted calculations in softmax further brings a 17% improvement. By introducing the high-performance grouped GEMM and fusing softmax into GEMMs, our fused MHA outperforms

the variant MHA implementations by 451%, 110% and 79% for maximal sequence lengths ranging 448 to 1024, where the average sequence length is 60% of the maximum.

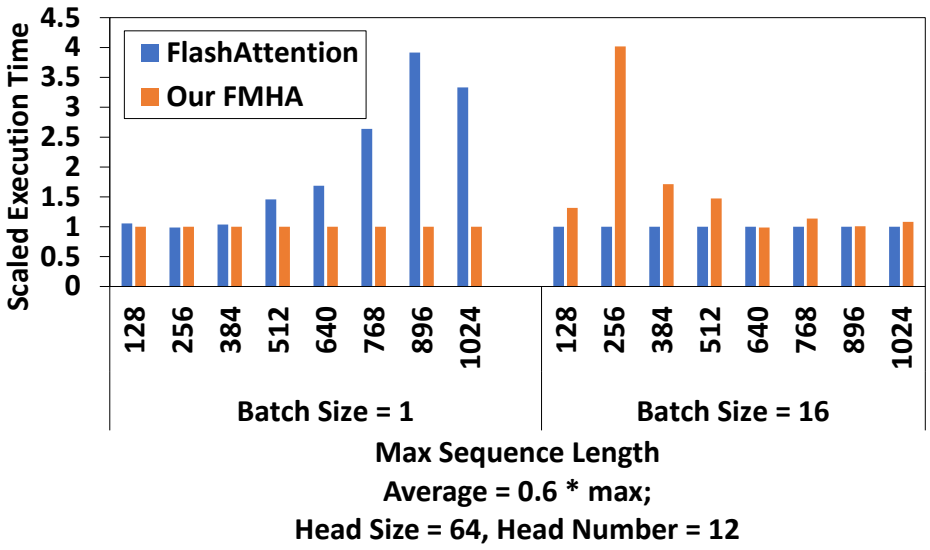


Figure 4.13: Comparisons of our FMHA with FlashAttention.

Figure 4.13 compares the scaled execution time of the FMHA module of our ByteTransformer against FlashAttention under the standard BERT setup. As shown in the figure, our FMHA presents advantages for small batch sizes (101% faster on average) while FlashAttention becomes more efficient for large batch sizes (59% faster on average). This is because FlashAttention maps a whole attention unit to a threadblock, which, although allows for the complete preservation of the intermediate matrix of an attention unit within shared-memory for any sequence length, results in performance degradation when there are insufficient tasks assigned.

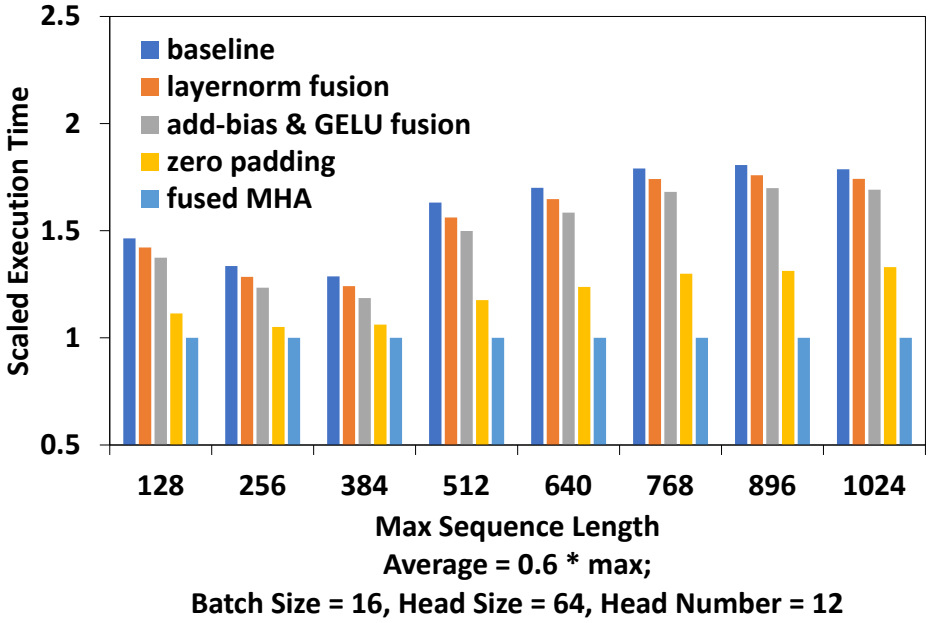


Figure 4.14: Single-layer BERT transformer with step-wise optimizations.

4.4.4 Benchmarking Single-Layer BERT Transformer With Step-Wise Optimizations

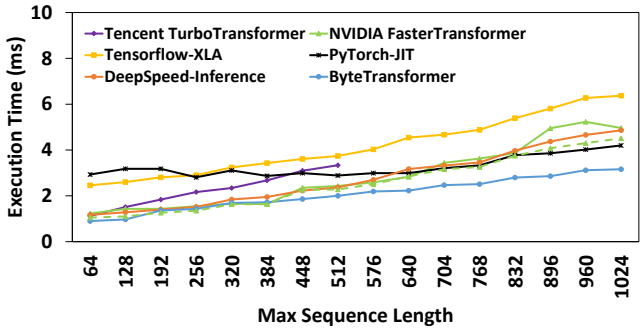
Figure 4.14 compares the performance of a single-layer BERT transformer to reflect our step-wise optimizations. At each step, we add a new optimization upon the previous variant. The baseline transformer implements the workflow in Figure 4.2 (a) with padding. We then enable kernel fusion for adding bias and layernorm, which corresponds to *layernorm fusion* in the figure. The next step is to fuse adding bias and GELU into GEMM, denoted by *add bias & GELU fusion*. In order to avoid calculating padded tokens for the variable-length inputs, we further propose the zero padding algorithm as shown in Figure 4.2 (c). This is denoted by *rm padding* in the figure. Our optimized transformer includes our high-performance fused MHA, as well as all previous optimizations.

Fusing adding bias and layernorm into one kernel improves the performance by 3.2%. Fusing adding bias and activation into GEMM epilogue further improves the performance by 3.8%. These two optimizations together improve the overall performance by 7.1%. After bringing in the zero padding algorithm, the redundant calculations are eliminated in most modules other than MHA. We observe a 24% improvement from the previous step. Finally, our fused MHA removes wasted calculations on padded tokens and enables an additional 20% improvement. To summarize, the final version achieves 60% improvement over the baseline version on single-layer BERT.

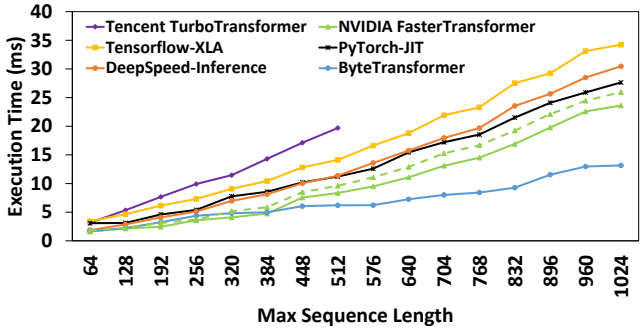
Table 4.3. Single-layer BERT versus E.T. on A100.

Sequence Length	E.T. (ms)	ByteTransformer (ms)	Speedup
256	0.25	0.07	3.57×
1024	1.04	0.09	11.56×

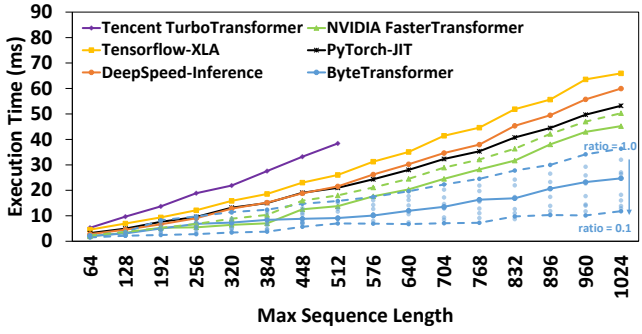
Table 4.3 compares the execution time for a single-layer, non-pruned BERT (batch size = 1) between E.T. and ByteTransformer, as E.T. has only open-sourced its single-layer, single-batch prototype. We achieve a speed-up of up to 11 times over E.T., which is optimized specifically for pruned models on legacy Volta GPUs. Since a pruned model can lead to significant reduction in total computations but with possible accuracy trade-offs, we do not include E.T. in our further end-to-end performance evaluations for non-pruned models on an A100 GPU for fairness and comparability.



(a) Batch size = 1



(b) Batch size = 8



(c) Batch size = 16

Figure 4.15: End-to-end benchmark for standard BERT transformer.

4.4.5 Benchmarking End-to-End Performance of BERT

The standard BERT transformer is a stacked structure of 12 layers of the encoder module. The output of each encoder module is utilized as an input tensor in the next iteration. Figure 4.15 shows the end-to-end performance of ByteTransformer and compares it against state-of-the-art transformer implementations: PyTorch with JIT, TensorFlow with XLA acceleration, Micorsoft DeepSpeed-Inference, NVIDIA FasterTransformer and Tencent TurboTransformer. We adopt the standard BERT transformer configuration for end-to-end benchmark: 12 heads, head size equal to 64 and 12 iterations (layers). We benchmark for cases whose batch sizes are equal to 1, 8 and 16 and change sequence lengths from 64 to 1024.

Compared with popular DL frameworks PyTorch, TensorFlow, and Microsoft DeepSpeed-Inference, our ByteTransformer achieves 87%, 131%, and 74% faster end-to-end performance on average. When benchmarking Tencent TurboTransformer, we turn on its `SmartBatch` mode to reach optimal batching performance. Since TurboTransformer only supports sequence lengths smaller than or equal to 512, we do not benchmark longer sequences for it. TurboTransformer re-groups and pads similar sequences into a batch so it launches excessive kernels at the run-time. It is faced with significant performance degradation for models with large batch numbers and sequence lengths. NVIDIA FasterTransformer, although it supports long sequences regarding the functionality, its back-end TensorRT fused MHA cannot be scaled to long sequences due to the limited register, its end-to-end efficiency cannot be maintained when the sequence length becomes longer than 512. Experimental results in Figure 4.15 show that ByteTransformer outperforms TurboTrans-

former and FasterTransformer by 138% and 55% on average, respectively. Figure 4.15 (c) further includes the end-to-end performance of ByteTransformer for average-to-maximum sequence length ratios ranging from 0.1 to 1.0. The upper dashed blue line represents the execution time of ByteTransformer at a ratio of 1.0, while the lower dashed line corresponds to a ratio of 0.1. Our padding-free algorithm reduces the runtime by up to 66% for a ratio of 0.1 compared to a fixed-sequence-length input. When disabling the support for variable-length inputs of FasterTransformer, as shown by the dashed green lines in Figure 4.15, we observe a moderate decrease in performance for larger batch sizes (batch sizes = 8 and 16) but an improvement in performance for a small batch size (batch size = 1). In contrast, our FMHA-enabled padding-free algorithm significantly improves the performance of the end-to-end BERT transformer for variable-length input with an average-to-maximum ratio of 0.6, outpacing NVIDIA FasterTransformer by a notable difference of 54% to 16%.

Table 4.4. Configurations of other BERT-like transformers.

Model	layer number	head number	head size
ALBERT	12	16	64
DistilBERT	6	12	64
DeBERTa	12	12	64

4.4.6 Extending to Other BERT-Like Transformers

We extend the optimizations on kernel fusion and the padding-free algorithm presented in our work to other BERT-like transformers, including ALBERT, DistilBERT,

and DeBERTa. Table 4.4 summarizes the model configurations, and readers can refer to [82, 109, 160] for more detailed information about their architectures. Figure 4.16 compares the performance of the ByteTransformer with state-of-the-art DL frameworks under these models. Following the setup for our demonstrated standard BERT benchmarks, the average sequence length is set to 60% of the maximal sequence length. TurboTransfomer only supports sequences shorter than 512, so its performance data for long sequences are not presented. FasterTransformer and TurboTransformer do not support DeBERTa, so their results are not included in that model. It is worth noting that TensorFlow encountered an out-of-memory error for sequence length 1024 in the DeBERTa model, resulting in this data point being excluded. For ALBERT and DistilBERT, our ByteTransformer on average outperforms PyTorch, TensorFlow, Tencent TurboTransformer, DeepSpeed-Inference, and NVIDIA FasterTransformer by 98%, 158%, 256%, 93%, and 53%, respectively. For the DeBERTa model, our ByteTransformer outperforms PyTorch, TensorFlow, and DeepSpeed by 44%, 243%, and 74%, respectively.

4.5 Conclusions

We have presented ByteTransformer, a high-performance transformer optimized for variable-length sequences. ByteTransformer not only brings algorithmic level innovation that frees the transformer from padding overhead, but also incorporates architecture-aware optimizations to accelerate functioning modules of the transformer. Our optimized fused MHA, as well as other step-wise optimizations, together provide us with significant speedup over current state-of-the-art transformers. The end-to-end performance of the stan-

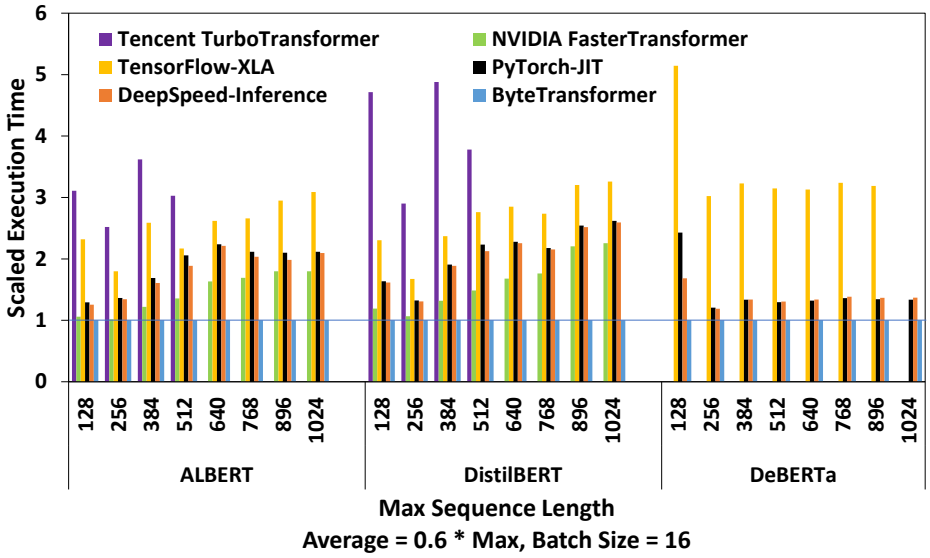


Figure 4.16: End-to-end benchmark for other BERT-like models.

dard BERT transformer benchmarked on an NVIDIA A100 GPU demonstrates that our ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed-Inference, and NVIDIA FasterTransformer by 87%, 131%, 138%, 74% and 55%, respectively. Moreover, we have shown that our optimizations are not specific to BERT, but can be applied to other BERT-like transformers, including ALBERT, DistilBERT, and DeBERTa. We are striving to make ByteTransformer completely open-source. This will allow the wider research community to benefit from our optimized implementation and to continue advancing the field. We are also dedicated to further expanding the presented strategies to accelerate a wider range of BERT-like transformer models, both in inference and training.

Chapter 5

Conclusions

This dissertation has presented a collection of performance optimization techniques applied to diverse software systems on modern computing platforms, including Intel and AMD x86 CPUs, and Intel and NVIDIA GPUs. The target applications range from the foundational Basic Linear Algebra Subprograms (BLAS) library to emerging applications in homomorphic encryption (HE) and machine learning systems. As hardware advancements slow down, software-level optimizations have emerged as an increasingly vital approach to enhancing performance, energy efficiency, and cost-effectiveness.

In this thesis, we have explored architectural-aware performance optimizations in three critical areas: BLAS, HE, and transformer-based machine learning models. Our key contributions include the development of FT-BLAS, XeHE, and ByteTransformer, all optimized to deliver significant performance improvements on modern processors.

FT-BLAS is a novel implementation of BLAS routines that incorporates fault tolerance while achieving competitive performance compared to state-of-the-art BLAS libraries

on widely-used processors. Our approach leverages a hybrid strategy, combining instruction duplication for memory-bound Level-1 and Level-2 BLAS routines with Algorithm-Based Fault Tolerance for compute-bound Level-3 BLAS routines. Assembly-level optimization and kernel fusion techniques contribute to our high performance and low overhead.

XeHE is a software framework that accelerates privacy-preserving computations on Intel GPUs, providing the first-ever GPU backend for the Microsoft SEAL library. Our optimizations span the instruction, algorithm, and application levels, resulting in significant performance gains for HE operations based on the CKKS scheme. The roofline analysis confirms the effectiveness of our optimizations, and the performance improvements extend to encrypted element-wise polynomial matrix operations.

ByteTransformer is a high-performance transformer model optimized for variable-length inputs, eliminating redundant computations caused by zero-padding. We implement architecture-aware optimizations for transformer functional modules, with a particular focus on the performance-critical Multi-Head Attention (MHA) algorithm. Our experimental results demonstrate that ByteTransformer surpasses the performance of state-of-the-art transformer frameworks for a forward BERT transformer, and our optimization methods are applicable to other BERT-like models, such as ALBERT, DistilBERT, and DeBERTa.

In summary, this dissertation demonstrates the importance and effectiveness of software-level optimizations in addressing the performance, energy efficiency, and cost challenges of modern computing platforms. Our work on BLAS, HE, and transformer-based machine learning systems showcases the potential for architectural-aware optimizations to accelerate a wide range of applications. As hardware advancements continue to slow, our

research contributes to the ongoing effort to push the boundaries of software performance, delivering sustainable and efficient computing solutions for the future.

Bibliography

[1] Intel Math Kernel Library. Reference Manual. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In 12th USENIX symposium on operating systems design and implementation (OSDI 16), pages 265–283, 2016.

[4] AGNER. `https://www.agner.org/optimize/instruction_tables.pdf`, 2019. Online.

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 70–95, 2018.

[6] Yasmin Alkady, Fifi Farouk, and Rawya Rizk. Fully homomorphic encryption with aes in cloud computing security. In International Conference on Advanced Intelligent Systems and Informatics, pages 370–382. Springer, 2018.

[7] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase,

et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. arXiv preprint arXiv:2207.00032, 2022.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[9] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In 2018 IEEE symposium on security and privacy (SP), pages 962–979. IEEE, 2018.

[10] Anna Antola, Roberto Negrini, MG Sami, and Nello Scarabottolo. Fault tolerance in FFT arrays: time redundancy approaches. Journal of VLSI signal processing systems for signal, image and video technology, 4(4):295–316, 1992.

[11] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for nvidia gpgpus. In 2019 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–8. IEEE, 2019.

[12] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In International Conference on Selected Areas in Cryptography, pages 423–442. Springer, 2016.

[13] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. IEEE Transactions on dependable and secure computing, 1(1):87–96, 2004.

[14] Robert Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, 7, 2002.

[15] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150, 2020.

[16] S Blackford, G Corliss, J Demmel, J Dongarra, I Duff, S Hammarling, G Henry, M Heroux, C Hu, W Kahan, et al. Basic linear algebra subprograms technical (blast) forum standard. Int. J. High Perform. Comput, 15:3–4, 2001.

[17] David Blythe. The xe gpu architecture. In 2020 IEEE Hot Chips 32 Symposium (HCS), pages 1–27. IEEE Computer Society, 2020.

[18] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). `https://github.com/intel/hexl`, September 2021.

[19] 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, pages 3–13, 2019.

[20] Joppe W Bos, Kristin Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. Journal of biomedical informatics, 50:234–243, 2014.

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

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[23] Aydin Buluc, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In 2011 IEEE International Parallel & Distributed Processing Symposium, pages 721–733. IEEE, 2011.

[24] Jon Calhoun, Marc Snir, Luke N Olson, and William D Gropp. Towards a more complete understanding of SDC propagation. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pages 131–142. ACM, 2017.

[25] Gizem S Cetin, Erkay Savas, and Berk Sunar. Homomorphic sorting with better scalability. IEEE Transactions on Parallel & Distributed Systems, 32(04):760–771, 2021.

[26] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In International Conference on Cryptology in India, pages 262–273. Springer, 2013.

[27] Chao Chen, Greg Eisenhauer, Santosh Pande, and Qiang Guan. Care: compiler-assisted recovery from soft failures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 58. ACM, 2019.

[28] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. Ladr: Low-cost application-level detector for reducing silent output corruptions. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, pages 156–167. ACM, 2018.

[29] Hongwei Chen, Yujia Zhai, Joshua J Turner, and Adrian Feiguin. A high-performance implementation of atomistic spin dynamics simulations on x86 cpus. arXiv preprint arXiv:2304.10966, 2023.

[30] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, et al. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, page 68. IEEE Press, 2018.

[31] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with GPUs. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 993–1002. IEEE, 2016.

[32] Longxiang Chen, Dingwen Tao, Panruo Wu, and Zizhong Chen. Extending checksum-based abft to tolerate soft errors online in iterative methods. In 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pages 344–351. IEEE, 2014.

[33] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for e-commerce recommendation in alibaba. In Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data, pages 1–4, 2019.

[34] Shiyang Chen, Shaoyi Huang, Santosh Pandey, Bingbing Li, Guang R Gao, Long Zheng, Caiwen Ding, and Hang Liu. Et: re-thinking self-attention for transformer models on gpus. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–18, 2021.

[35] Zhi Chen, Alexandru Nicolau, and Alexander V Veidenbaum. SIMD-based soft error detection. In Proceedings of the ACM International Conference on Computing Frontiers, pages 45–54. ACM, 2016.

[36] Zizhong Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In 2008 IEEE International Symposium on Parallel and Distributed Processing, pages 1–8. IEEE, 2008.

[37] Zizhong Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In ACM SIGPLAN Notices, volume 48, pages 167–176. ACM, 2013.

[38] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. IEEE Transactions on Parallel and Distributed Systems, 19(12):1628–1641, 2008.

[39] Zizhong Chen and Jack Dongarra. A scalable checkpoint encoding algorithm for diskless checkpointing. In 2008 11th IEEE High Assurance Systems Engineering Symposium, pages 71–79. IEEE, 2008.

[40] L Cheng, M Shib, Y t Bian, et al. A study on improved cockroach swarm optimization algorithm. In 7th International Conference on Computer Engineering and Networks, page 4, 2017.

[41] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 360–384. Springer, 2018.

[42] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In International Conference on Selected Areas in Cryptography, pages 347–368. Springer, 2018.

[43] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In International Conference on the Theory and Application of Cryptology and Information Security, pages 409–437. Springer, 2017.

[44] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Secure dna-sequence analysis on encrypted dna nucleotides. Manuscript at http://media. eurekalert. org/aaasnewsroom/MCM/-FIL, 1439.

[45] Chen-Yong Cher, Meeta S Gupta, Pradip Bose, and K Paul Muller. Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 587–596. IEEE, 2014.

[46] Andrew Chien, Pavan Balaji, Peter Beckman, Nan Dun, Aiman Fang, Hajime Fujita, Kamil Iskra, Zachary Rubenstein, Ziming Zheng, Rob Schreiber, et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. Procedia Computer Science, 51:29–38, 2015.

[47] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption library over the torus, 2016.

[48] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259, 2014.

[49] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled psi from homomorphic encryption with reduced computation and communication. 2021.

[50] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. Intel Corporation, Sept, 2019.

[51] cpuwiki. `https://en.wikichip.org/wiki/amd/microarchitectures/zen`, 2021. Online.

[52] Majid Dadashi, Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Hardware-software integrated diagnosis for intermittent hardware faults. In 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 363–374. IEEE, 2014.

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

[54] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. arXiv preprint arXiv:2205.14135, 2022.

[55] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[56] Sheng Di and Franck Cappello. Adaptive impact-driven detection of silent data corruption for HPC applications. IEEE Transactions on Parallel and Distributed Systems, 27(10):2809–2823, 2016.

[57] Dirk Schmidl,Christian Terboven. `http://prace.it4i.cz/sites/prace.it4i.cz/files/files/advancedopenmptutorial_2.pdf`, 2019. Online.

[58] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. International Journal of High Performance Computing Applications, 25(1):3–60, 2011.

[59] Yarkin Doröz, Yin Hu, and Berk Sunar. Homomorphic aes evaluation using ntru. IACR Cryptol. ePrint Arch., 2014:39, 2014.

[60] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient pir from ntru. In International Conference on Financial Cryptography and Data Security, pages 195–207. Springer, 2014.

[61] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale. arXiv preprint arXiv:1808.09381, 2018.

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

[63] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 389–402, 2021.

[64] Flexera. `https://www.flexera.com/blog/cloud/cloud-computing-trends-2021-state-of-the-cloud-report/`, Retrieved in 2021. Online.

[65] Al Geist. Supercomputing's monster in the closet. IEEE Spectrum, 53(3):30–35, 2016.

[66] Craig Gentry. A fully homomorphic encryption scheme. Stanford university, 2009.

[67] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Proceedings of the forty-first annual ACM symposium on Theory of computing, pages 169–178, 2009.

[68] Craig Gentry. Computing arbitrary functions of encrypted data. Communications of the ACM, 53(3):97–105, 2010.

[69] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In Annual Cryptology Conference, pages 850–867. Springer, 2012.

[70] Jia-Zheng Goey, Wai-Kong Lee, Bok-Min Goi, and Wun-She Yap. Accelerating number theoretic transform in gpu platform for fully homomorphic encryption. The Journal of Supercomputing, 77:1455–1474, 2021.

[71] Leonardo Arturo Bautista Gomez and Franck Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In 2015 IEEE International Conference on Cluster Computing, pages 595–602. IEEE, 2015.

[72] Google. `https://www.tensorflow.org/xla`, Retrieved in 2022. Online.

[73] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS), 34(3):1–25, 2008.

[74] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. ACM Transactions on Mathematical Software (TOMS), 35(1):1–14, 2008.

[75] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pages 1–12. Ieee, 2008.

[76] Thore Graepel, Kristin Lauter, and Michael Naehrig. Ml confidential: Machine learning on encrypted data. In International Conference on Information Security and Cryptology, pages 1–21. Springer, 2012.

[77] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 1245–1254. IEEE, 2014.

[78] John A Gunnels, Daniel S Katz, Enrique S Quintana-Orti, and RA Van de Gejin. Fault-tolerant high-performance matrix multiplication: Theory and practice. In 2001 International Conference on Dependable Systems and Networks, pages 47–56. IEEE, 2001.

[79] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. IEEE Transactions on Parallel and Distributed Systems, 26(5):1323–1335, 2014.

[80] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. IBM Research (Manuscript), 6(12-15):8–36, 2013.

[81] David Harvey. Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation, 60:113–119, 2014.

[82] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. arXiv preprint arXiv:2006.03654, 2020.

[83] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415, 2016.

[84] Michael A Heroux. Toward resilient algorithms and applications. In Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale, pages 1–2. ACM, 2013.

[85] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In International Algorithmic Number Theory Symposium, pages 267–288. Springer, 1998.

[86] Robert W Horst, Richard L Harris, and Robert L Jardine. Multiple instruction issue in the nonstop cyclone processor. ACM SIGARCH Computer Architecture News, 18(2SI):216–226, 1990.

[87] P YT Hsu and Edward S Davidson. Highly concurrent scalar processing. ACM SIGARCH Computer Architecture News, 14(2):386–395, 1986.

[88] Jiajun Huang, Sheng Di, Xiaodong Yu, Yujia Zhai, Jinyang Liu, Ken Raffenetti, Hui Zhou, Kai Zhao, Zizhong Chen, Franck Cappello, et al. C-coll: Introducing error-bounded lossy compression into mpi collectives. arXiv preprint arXiv:2304.03890, 2023.

[89] Jianyu Huang, Tyler M Smith, Greg M Henry, and Robert A van de Geijn. Strassen's algorithm reloaded. In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 690–701. IEEE, 2016.

[90] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. IEEE transactions on computers, 100(6):518–528, 1984.

[91] Intel. `https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html`, 2014. Online.

[92] Intel. `https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor`, 2020. Online.

[93] Intel. `https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-icllp-vol02a-commandreference-instructions_2.pdf`, Retrieved in 2021. Online.

[94] Intel. `https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SubGroup/SYCL_INTEL_sub_group.asciidoc`, Retrieved in 2021. Online.

[95] Intel. https://github.com/intel/intel-graphics-compiler/tree/master/documentation/visa, Retrieved in 2022. Online.

[96] Intel Corporation. https://software.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf, Retrieved in 2021. Online.

[97] Intel Corporation. https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf, Retrieved in 2021. Online.

[98] Intel LLVM. https://intel.github.io/llvm-docs/MultiTileCardWithLevelZero.html, Retrieved in 2021. Online.

[99] Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html. Online.

[100] intelskylake. https://www.7-cpu.com/cpu/Skylake.html, 2019. Online.

[101] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning, pages 448–456. PMLR, 2015.

[102] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia, pages 675–678, 2014.

[103] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. Heaan demystified: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. arXiv preprint arXiv:2003.04510, 2020.

[104] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836, 2016.

[105] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In 2020 IEEE International Symposium on Workload Characterization (IISWC), pages 264–275. IEEE, 2020.

[106] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A Rutenbar. Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme. In 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 56–64. IEEE, 2020.

[107] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: hardware-assisted fault tolerance. In Proceedings of the Eleventh European Conference on Computer Systems, page 25. ACM, 2016.

[108] Kim Laine. Simple encrypted arithmetic library 2.3. 1. Microsoft Research https://www. microsoft. com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1. pdf, 2017.

[109] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942, 2019.

[110] Jean-Claude Laprie. Dependable computing and fault-tolerance. Digest of Papers FTCS-15, pages 2–11, 1985.

[111] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 648–677. Springer, 2021.

[112] Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, page 57. IEEE Computer Society Press, 2012.

[113] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 8. ACM, 2017.

[114] Sihuan Li, Hongbo Li, Xin Liang, Jieyang Chen, Elisabeth Giem, Kaiming Ouyang, Kai Zhao, Sheng Di, Franck Cappello, and Zizhong Chen. FT-iSort: efficient fault tolerance for introsort. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 71. ACM, 2019.

[115] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. Correcting soft errors online in fast Fourier transform. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 30. ACM, 2017.

[116] JH Lim, HK Kim, and YK Kim. Recent r&d trends for pretrained language model. Electronics and Telecommunications Trends, 35(3):9–19, 2020.

[117] Liyuan Liu, Jialu Liu, and Jiawei Han. Multi-head or single-head? an empirical comparison for transformer training. arXiv preprint arXiv:2106.09650, 2021.

[118] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In International Conference on Cryptology and Network Security, pages 124–139. Springer, 2016.

[119] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Proceedings of the forty-fourth annual ACM symposium on Theory of computing, pages 1219–1234, 2012.

[120] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. DOE advanced scientific computing advisory subcommittee (ASCAC) report: top ten exascale research challenges. Technical report, USDOE Office of Science (SC)(United States), 2014.

[121] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 40(6):190–200, 2005.

[122] Robyn R Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In [1993] Proceedings of the IEEE International Symposium on Requirements Engineering, pages 126–133. IEEE, 1993.

[123] Alexander Lyashevsky, Alexey Titov, Yiqin Qiu, and Yujia Zhai. XeHE: an Intel GPU Accelerated Fully Homomorphic Encryption Library, pages 1–115. Codeplay Software, 1 edition, 2023. Copyright by Codeplay Software, an independently managed wholly owned subsidiary of Intel Corporation.

[124] Timothy C May and Murray H Woods. Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices, 26(1):2–9, 1979.

[125] Subhasish Mitra, Pradip Bose, Eric Cheng, Chen-Yong Cher, Hyungmin Cho, Rajiv Joshi, Young Moon Kim, Charles R Lefurgy, Yanjing Li, Kenneth P Rodbell, et al. The resilience wall: Cross-layer solution strategies. In Proceedings of Technical Program-2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA), pages 1–11. IEEE, 2014.

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

[127] Michael Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146), pages 86–94. IEEE, 1999.

[128] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. When cache blocking of sparse matrix vector multiply works and why. Applicable Algebra in Engineering, Communication and Computing, 18(3):297–311, 2007.

[129] nucypher. https://github.com/nucypher/nufhe, Retrieved in 2021. Online.

[130] NVIDIA . https://github.com/NVIDIA/cutlass/tree/master/examples/41_multi_head_attention, Retrieved in 2022. Online.

[131] NVIDIA . https://developer.nvidia.com/tensorrt, Retrieved in 2022. Online.

[132] NVIDIA . https://developer.nvidia.com/cublas, Retrieved in 2022. Online.

[133] NVIDIA . https://github.com/NVIDIA/TensorRT/tree/main/plugin/bertQKVToContextPluginl, Retrieved in 2022. Online.

[134] NVIDIA. https://github.com/NVIDIA/FasterTransformer, Retrieved in 2022. Online.

[135] NVIDIA. https://github.com/NVIDIA/cutlass, Retrieved in 2022. Online.

[136] NVIDIA. https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/gemm/kernel/grouped_problem_visitor.h#L203-L322, Retrieved in 2022. Online.

[137] Nahmsuk Oh, Philip P Shirvani, and McCluskey. Control-flow checking by software signatures. IEEE transactions on Reliability, 51(1):111–122, 2002.

[138] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability, 51(1):63–75, 2002.

[139] Daniel Oliveira, Vinicius Frattin, Philippe Navaux, Israel Koren, and Paolo Rech. Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In Proceedings of the Computing Frontiers Conference, pages 295–298, 2017.

[140] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. Experimental and analytical study of Xeon Phi reliability. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 28. ACM, 2017.

[141] OpenAI. https://chat.openai.com/, Retrieved in 2023. Online.

[142] OpenBLAS. https://github.com/xianyi/OpenBLAS/blob/develop/common.h#L530, Retrieved in 2021. Online.

[143] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In International conference on the theory and applications of cryptographic techniques, pages 223–238. Springer, 1999.

[144] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.

Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[145] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32, 2019.

[146] David A Patterson and John L Hennessy. Computer Organization and Design ARM Edition: The Hardware Software Interface. Morgan kaufmann, 2016.

[147] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with NAMD. Journal of computational chemistry, 26(16):1781–1802, 2005.

[148] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. Palisade lattice cryptography library user manual. Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep, 15, 2017.

[149] PyTorch. `https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html`, Retrieved in 2022. Online.

[150] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9, 2019.

[151] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

[152] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 3505–3506, 2020.

[153] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 325–342, 2020.

[154] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Springer Nature, 2021.

[155] Steven K Reinhardt and Shubhendu S Mukherjee. Transient fault detection via simultaneous multithreading. In Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201), pages 25–36. IEEE, 2000.

[156] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In Proceedings of the international symposium on Code generation and optimization, pages 243–254. IEEE Computer Society, 2005.

[157] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture for computing on encrypted data. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 1295–1309, 2020.

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

[159] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In 2019 IEEE International symposium on high performance computer architecture (HPCA), pages 387–398. IEEE, 2019.

[160] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108, 2019.

[161] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, pages 1–8, 2013.

[162] A Shinsel. Intel advisor roofline.

[163] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[164] William C Skamarock, Joseph B Klemp, Jimy Dudhia, David O Gill, Dale M Barker, Wei Wang, and Jordan G Powers. A description of the advanced research wrf version 3. ncar technical note-475+ str. 2008.

[165] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), pages 1–12. IEEE, 2012.

[166] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In International Workshop on Public Key Cryptography, pages 420–443. Springer, 2010.

[167] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. Designs, codes and cryptography, 71(1):57–81, 2014.

[168] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 1049–1059. IEEE, 2014.

[169] Tyler M Smith, Robert A van de Geijn, Mikhail Smelyanskiy, and Enrique S Quintana-Ortı. Toward ABFT for BLIS GEMM.

[170] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. The International Journal of High Performance Computing Applications, 28(2):129–173, 2014.

[171] Damien Stehlé and Ron Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In Annual international conference on the theory and applications of cryptographic techniques, pages 27–47. Springer, 2011.

[172] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In Proceedings of the 28th ACM international conference on information and knowledge management, pages 1441–1450, 2019.

[173] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems, pages 1057–1063, 2000.

[174] Li Tan, Shashank Kothapalli, Longxiang Chen, Omar Hussaini, Ryan Bissiri, and Zizhong Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. Parallel Computing, 40(10):559–573, 2014.

[175] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In 2015 IEEE International Parallel and Distributed Processing Symposium, pages 786–796. IEEE, 2015.

[176] Wei Tang, Teague Tomesh, Jeffrey Larson, Martin Suchara, and Margaret Martonosi. Cutqc: Using small quantum computers for large quantum circuit evaluations. arXiv preprint arXiv:2012.02333, 2020.

[177] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Improving performance of iterative methods by lossy checkponting. In Proceedings of the 27th international symposium on high-performance parallel and distributed computing, pages 52–65, 2018.

[178] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. New-sum: A novel online ABFT scheme for general iterative methods. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 43–55. ACM, 2016.

[179] DL Tao and Carlos RP Hartmann. A novel concurrent error detection scheme for FFT networks. IEEE Transactions on Parallel and Distributed Systems, 4(2):198–221, 1993.

[180] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD. MIT Press, 2017.

[181] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software, 41(3):14:1–14:33, June 2015.

[182] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[183] vernamlab. `https://github.com/vernamlab/cuFHE`, Retrieved in 2021. Online.

[184] Richard Vuduc and Katherine Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. 2004.

[185] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–12. IEEE, 2013.

[186] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using gpu. In 2012 IEEE conference on high performance extreme computing, pages 1–5. IEEE, 2012.

[187] Xiaohui Wang, Ying Xiong, Xian Qian, Yang Wei, Lei Li, and Mingxuan Wang. Lightseq2: Accelerated training for transformer-based models on gpus. arXiv preprint arXiv:2110.05722, 2021.

[188] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A high performance inference library for transformers. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT), pages 113–120. Association for Computational Linguistics, June 2021.

[189] Zhongde Wang. Fast algorithms for the discrete w transform and for the discrete fourier transform. IEEE Transactions on Acoustics, Speech, and Signal Processing, 32(4):803–816, 1984.

[190] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. Parallel computing, 27(1-2):3–35, 2001.

[191] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing, pages 38–38. IEEE, 1998.

[192] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65–76, 2009.

[193] Panruo Wu and Zizhong Chen. Ft-scalapack: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, pages 49–60. ACM, 2014.

[194] Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. On-line soft error correction in matrix–matrix multiplication. Journal of Computational Science, 4(6):465–472, 2013.

[195] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 31–42, 2016.

[196] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S Vetter, and Sparsh Mittal. Algorithm-directed data placement in explicitly managed non-volatile memory. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 141–152, 2016.

[197] Shixun Wu, Yujia Zhai, Jiajun Huang, Zizhe Jian, and Zizhong Chen. Ft-gemm: A fault tolerant high performance gemm implementation on x86 cpus. arXiv preprint arXiv:2305.02444, 2023.

[198] Shixun Wu, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Bryan M Wong, and Zizhong Chen. Anatomy of high-performance gemm with online fault tolerance on gpus. arXiv preprint arXiv:2305.01024, 2023.

[199] Shuyin Xia, Xinyu Bai, Guoyin Wang, Yunlong Cheng, Deyu Meng, Xinbo Gao, Yujia Zhai, and Elisabeth Giem. An efficient and accurate rough set for feature selection, classification, and knowledge representation. IEEE Transactions on Knowledge and Data Engineering, 2022.

[200] SY Xia, C Wang, GY Wang, WP Ding, XB Gao, JH Yu, YJ Zhai, and ZZ Chen. A unified granular-ball learning model of pawlak rough set and neighborhood rough set. arXiv preprint arXiv:2201.03349, 2022.

[201] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. Advances in neural information processing systems, 32, 2019.

[202] Ying C Yeh. Triple-triple redundant 777 primary flight computer. In 1996 IEEE Aerospace Applications Conference. Proceedings, volume 1, pages 293–307. IEEE, 1998.

[203] Jing Yu, Maria Jesus Garzaran, and Marc Snir. Esoftcheck: Removal of non-vital checks for fault tolerance. In 2009 International Symposium on Code Generation and Optimization, pages 35–46. IEEE, 2009.

[204] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. Greenmm: energy efficient GPU matrix multiplication through undervolting. In Proceedings of the ACM International Conference on Supercomputing, pages 308–318, 2019.

[205] Jinle Zeng, Min Li, Zhihua Wu, Jiaqi Liu, Yuang Liu, Dianhai Yu, and Yanjun Ma. Boosting distributed training performance of the unpadded bert model. arXiv preprint arXiv:2208.08124, 2022.

[206] Yujia Zhai, Elisabeth Giem, Quan Fan, Kai Zhao, Jinyang Liu, and Zizhong Chen. Ft-blas: a high performance blas implementation with online fault tolerance. In Proceedings of the ACM International Conference on Supercomputing, pages 127–138, 2021.

[207] Yujia Zhai, Elisabeth Giem, Kai Zhao, Jinyang Liu, Jiajun Huang, Bryan Wong, Christian Shelton, and Zizhong Chen. Accelerating fault-tolerant blas on x86 cpus, July 2022.

[208] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. Accelerating encrypted computing on intel gpus. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 705–716. IEEE, 2022.

[209] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. ByteTransformer: A high-performance transformer boosted for variable-length inputs. arXiv preprint arXiv:2210.03052, 2022.

[210] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Algorithm-based fault tolerance for convolutional neural networks. IEEE Transactions on Parallel and Distributed Systems, 2020.