

Double-Precision Floating-Point on AWS Trainium Using Integer Units: A First Look

Yunhao Lan, Larry Tang, Naifeng Zhang, Franz Franchetti

Department of Electrical and Computer Engineering

Carnegie Mellon University, Pittsburgh, PA

yunhaola@andrew.cmu.edu, lawrenct@andrew.cmu.edu, naifengz@cmu.edu, franzf@andrew.cmu.edu

Abstract—Modern accelerators increasingly emphasize low-precision arithmetic for performance and energy efficiency, while many scientific computing workloads still require double precision. This creates a need to understand whether software-emulated double-precision arithmetic can be efficiently mapped to such architectures. In this paper, we present a first-look study of double-precision floating-point (FP64) arithmetic using integer units on AWS Trainium through the custom C++ operator flow targeting its GPSIMD engine. This approach provides FP64-level accuracy on hardware that primarily targets lower-precision computation. Beyond accuracy, our results show that the batched radix-4 software-emulated floating-point kernel achieves a 1.88 times speedup over MKL-based CPU baselines for the equivalent 1D 4-point FFT computation on the Trainium host CPU. These results provide early evidence that our software-based approach can extend Trainium to FP64 kernels, while also revealing custom-operator limitations for larger FFT pipelines.

Index Terms—floating-point emulation, extended precision, fast Fourier transform, AWS Trainium

I. INTRODUCTION

Modern accelerators increasingly devote more hardware resources to reduced-precision arithmetic to improve throughput and energy efficiency. This trend appears across GPUs and domain-specific accelerators, where formats such as FP16, BF16, TF32, FP8, and integer arithmetic are commonly optimized for machine learning and other throughput-oriented workloads [1], [2]. However, many scientific applications still require higher precision. For example, FFT-based solvers, spectral methods, and iterative linear algebra kernels can accumulate numerical error over many stages or iterations, making IEEE 754 double precision (FP64) or higher precision important for obtaining reliable results [3]–[5]. This creates a gap between the arithmetic formats emphasized by modern accelerators and the precision requirements of many HPC and scientific computing workloads.

Software-emulated floating-point arithmetic provides a possible path for running arbitrary-precision numerical kernels on hardware without native support for the desired format. Such approaches represent these values using lower-level hardware operations, including integer arithmetic, lower-precision floating-point arithmetic, or combinations of multiple hardware-supported formats [6]–[10]. This flexibility allows software to recover a chosen range and precision, but the cost depends strongly on how well the emulated operations map to the target architecture. In practice, performance is limited not only by the number of arithmetic operations,

but also by whether intermediate values can remain close to the compute units or must be repeatedly moved through the memory hierarchy.

Among these approaches, this paper focuses on an integer-component software floating-point representation. In this representation, double-precision values are stored using multiple 32-bit integer components, and arithmetic is implemented through integer operations. We study this approach on AWS Trainium using batched radix- r butterfly kernels as a first-look workload. These kernels implement the local radix- r computation and twiddle-factor multiplication used in FFT stages, making them reusable computational building blocks for full FFT implementations. For brevity, we refer to them as radix kernels throughout the paper. Instead of claiming end-to-end FFT performance, this study characterizes batched radix-2, radix-4, and radix-8 software-emulated kernels implemented through the custom C++ operator flow targeting Trainium’s GPSIMD engine (gpsimdE) [2], [11], [12]. This allows us to evaluate which radix size provides the best balance for future FFT pipelines while also exposing the limitations of using custom C++ operators for larger FFT-style kernels.

This paper makes the following contributions:

- We present a first-look implementation of software-emulated radix kernels on AWS Trainium using the custom C++ operator flow targeting gpsimdE.
- We develop a tiled batched implementation for radix-2, radix-4, and radix-8 kernels using private tightly coupled memory (TCM) storage and Xtensa-IVP SIMD intrinsics for gpsimdE computation.
- We evaluate radix-2, radix-4, and radix-8 software-emulated kernels, show that radix-4 achieves the best steady-state efficiency, and compare it against Intel Math Kernel Library (MKL)-based CPU baselines.
- We identify custom C++ operator limitations for larger FFT-style kernels, including the lack of state buffer (SBUF), the on-chip scratchpad memory, exposure and compilation failures caused by generated `.text` section overflow beyond 1,024-point kernels.

The rest of this paper is organized as follows. Section II reviews related work and background. Section III presents our emulated floating-point representation and batched radix-kernel implementation. Section IV evaluates performance, including radix-size selection, CPU baseline comparisons,

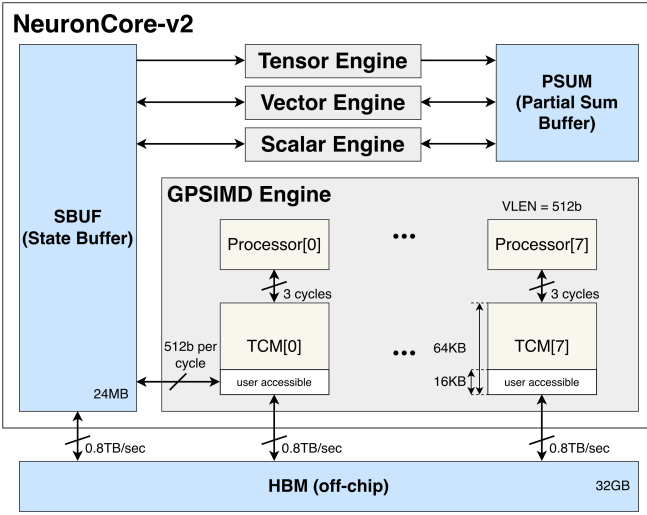


Fig. 1. Structure of a single NeuronCore, highlighting gpsimdE-related components. Each NeuronCore contains four compute engines: tensor engine (tensorE), vector engine (vectorE), scalar engine (scalarE), and GPSIMD engine (gpsimdE).

and scalability limitations of the custom C++ operator flow. Section V concludes the paper and outlines future work.

II. BACKGROUND AND RELATED WORK

This section reviews related work and introduces the background needed for our study, including IEEE floating-point formats, AWS Trainium gpsimdE programming, and radix kernels as FFT building blocks.

A. Related Work

Algorithmic precision recovery. Mixed-precision numerical methods are a widely used approach for balancing performance and accuracy: low-precision arithmetic is used where it is efficient, while higher precision is retained in accuracy-critical parts of the computation. This general principle has been studied across numerical linear algebra and scientific computing [3], [4], [13]. Iterative refinement is a representative example, where low-precision computation can be combined with higher-precision residuals or correction steps to recover accurate solutions [14], [15]. Another line of work uses reduced-precision tensor cores or integer matrix units to recover high-accuracy GEMM results through decomposition-based techniques such as the Ozaki scheme [8], [9], [16]–[18]. Together, these works show that reduced-precision hardware can be used effectively for high-accuracy computation when the algorithm provides a way to recover or preserve precision. Our work is complementary in that it focuses on the lower-level software-emulated arithmetic kernels needed to provide FP64 operations on AWS Trainium.

Software-emulated arithmetic. Software approaches have long been used to provide arithmetic formats beyond direct hardware support, from early interpretive floating-point systems to modern software libraries [19]–[24]. Berkeley SoftFloat implements IEEE-compliant binary floating-point

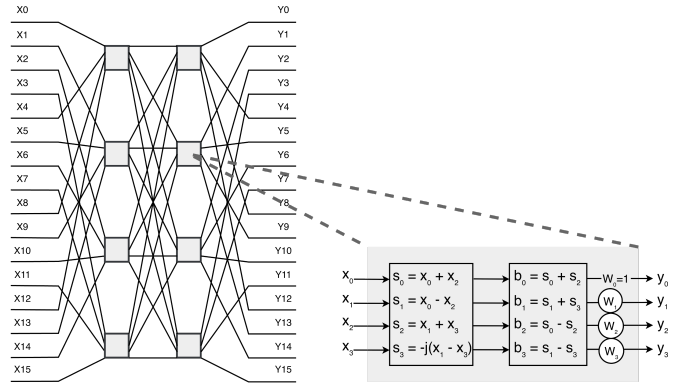


Fig. 2. A 16-point FFT composed of radix-4 kernels. Each stage applies multiple radix-4 kernels with stage-dependent twiddle factors. A radix-4 kernel operates on four complex values and serves as the reusable building block for larger FFTs, rather than as a standalone 4-point FFT.

arithmetic in software [6], while double-double, quad-double, arbitrary-precision, and multiple-precision libraries represent higher precision using multiple machine words or floating-point components [7], [24]–[26]. Other systems support customized, simulated, or software-defined floating-point formats for exploring precision and performance tradeoffs [27]–[29]. Prior work has also explored extended-precision arithmetic and floating-point emulation on GPUs and accelerator units [30]–[32]. These efforts demonstrate that precision beyond native hardware support can be recovered in software, but they primarily target scalar libraries, GPUs, graph workloads, tensor-core-style emulation, or GEMM-oriented computation. In contrast, this paper studies an integer-component floating-point representation for FFT radix kernels on Trainium gpsimdE.

B. IEEE Floating-Point Formats

Binary floating-point formats represent real-valued data by separating each number into a sign, an exponent, and a mantissa [33]. The sign determines whether the value is positive or negative, the exponent determines the scale of the value, and the mantissa determines the number of significant bits retained at that scale. This separation allows floating-point numbers to represent both very large and very small magnitudes within a fixed number of bits.

Different IEEE formats allocate different numbers of bits to the exponent and mantissa. A wider exponent increases the dynamic range by allowing values with larger or smaller magnitudes to be represented, while a wider mantissa increases precision by reducing the spacing between nearby representable values. For example, FP64 provides more exponent and mantissa bits than FP32, which gives it both greater range and higher precision.

This distinction between range and precision is important for software-emulated floating-point arithmetic. Instead of relying on a native hardware floating-point format, a software representation can explicitly store the exponent and mantissa using integer components. This allows the implementation to choose the range and precision needed by the target numerical

kernel, even when the underlying accelerator does not provide native support for that format.

C. AWS Trainium *gpsimdE* with Custom C++ Operators

AWS Trainium uses the NeuronCore as its basic programmable compute unit, with each NeuronCore containing multiple compute engines, including *gpsimdE* for general-purpose SIMD-style computation [2]. We focus on *gpsimdE* because the emulated floating-point arithmetic is dominated by 32-bit integer operations, bit-level manipulation, comparisons, shifts, and normalization logic, which map more naturally to a general-purpose SIMD engine than to tensor-oriented floating-point units.

As shown in Figure 1, *gpsimdE* consists of eight processors, each with a 512-bit vector length and a private tightly coupled memory (TCM). The TCM is local to each *gpsimdE* processor and is not shared across processors. Therefore, each processor must explicitly load the data it needs into its own TCM, and values stored in one processor’s TCM cannot be directly accessed by another processor [2].

Trainium custom C++ operators provide a programming path for user-defined kernels targeting *gpsimdE*. In this flow, data is explicitly moved between HBM and each *gpsimdE* processor’s private TCM through TCM accessors, while computation is performed using Xtensa-IVP intrinsics [11], [12].

However, the custom C++ operator flow does not expose the NeuronCore state buffer (SBUF), the 48 MB scratchpad used in the NKI programming flow. As a result, data movement is organized through explicit HBM–TCM transfers rather than SBUF-resident intermediate reuse [11]. This makes larger kernels, such as full FFT pipelines that require data reuse across multiple stages, difficult to express efficiently under this programming model.

D. Radix Kernels as FFT Building Blocks

A radix- r kernel computes the local butterfly operation over r complex values and applies the stage-dependent twiddle factors required by a larger FFT decomposition [34]. It differs from a standalone r -point FFT in that the twiddle factors are not fixed constants of the small transform, but variable inputs determined by the FFT stage and butterfly index. This makes the radix kernel reusable across different stages and problem sizes.

An N -point FFT can be constructed by applying radix- r kernels across multiple stages. Each stage consists of multiple radix- r kernel instances, each operating on a group of r complex values with stage-specific twiddle factors. Across stages, the grouping pattern and twiddle factors change according to the FFT decomposition. Therefore, a complete FFT can be viewed as a sequence of radix-kernel stages connected by data reordering and stride changes.

Figure 2 illustrates this idea using a 16-point FFT built from radix-4 kernels. The outer view shows how multiple radix-4 kernels form the FFT stages, while the zoomed-in view shows one radix-4 kernel at the complex-pair arithmetic level.

III. METHODOLOGY

This section first introduces the emulated floating-point representation used in this work and then describes how radix-2, radix-4, and radix-8 kernels are implemented using the custom C++ operator flow targeting *gpsimdE*.

A. Emulated Floating-Point Representation

Our software-emulated floating-point representation on AWS Trainium stores each real value using an explicit exponent and mantissa. The exponent is represented as a 32-bit signed integer and determines the dynamic range, while the mantissa uses 64 bits and determines the available precision. This separation allows the software format to preserve FP64-level precision while expressing arithmetic using integer operations supported by *gpsimdE*.

To match the 32-bit data granularity of AWS Trainium, we split the 64-bit mantissa into two 32-bit components, corresponding to the high and low mantissa words. Each emulated value is therefore represented by three 32-bit integer fields: one exponent field and two mantissa fields. For complex-valued numerical data, each complex number contains two emulated values, one for the real part and one for the imaginary part, resulting in six 32-bit integer components per complex value.

Our emulation consists of three basic operations: addition, subtraction, and multiplication, implemented in software using integer instructions on the three-component representation. We also provide a separate normalization kernel that can be invoked when needed to restore the mantissa and exponent to the expected normalized form.

B. Emulated Floating-Point Radix Kernel on *gpsimdE*

Using the custom C++ operator flow targeting *gpsimdE*, we implement batched software-emulated floating-point radix kernels. As shown in Figure 3, our radix-4 butterfly incorporates the fixed 4-point DFT rotation, so the subsequent twiddle multiplication applies only the stage-dependent twiddle factors. The precomputed twiddle factors are loaded into the private TCM of each *gpsimdE* processor and reused across batches as they remain unchanged. Because the private TCM is not shared across *gpsimdE* processors, each processor maintains its own copy of the twiddle factors.

The maximum tile batch size is constrained by the 16 KB TCM capacity available to each *gpsimdE* processor. For a radix- r kernel, the TCM footprint is estimated as

$$(2 \times 3 \times 2 \times r \times \text{TileBatch} + \text{Twid}) \times 4 < 16 \text{ KB}, \quad (1)$$

where the first term accounts for the input and output buffers, the three emulated components, the real and imaginary parts, the radix size, and the tile batch size. The second term accounts for the precomputed twiddle factors stored in TCM. Here,

$$\text{Twid} = 3 \times 2 \times (r - 1), \quad (2)$$

because each twiddle factor is represented by three emulated components for both real and imaginary parts, and the first

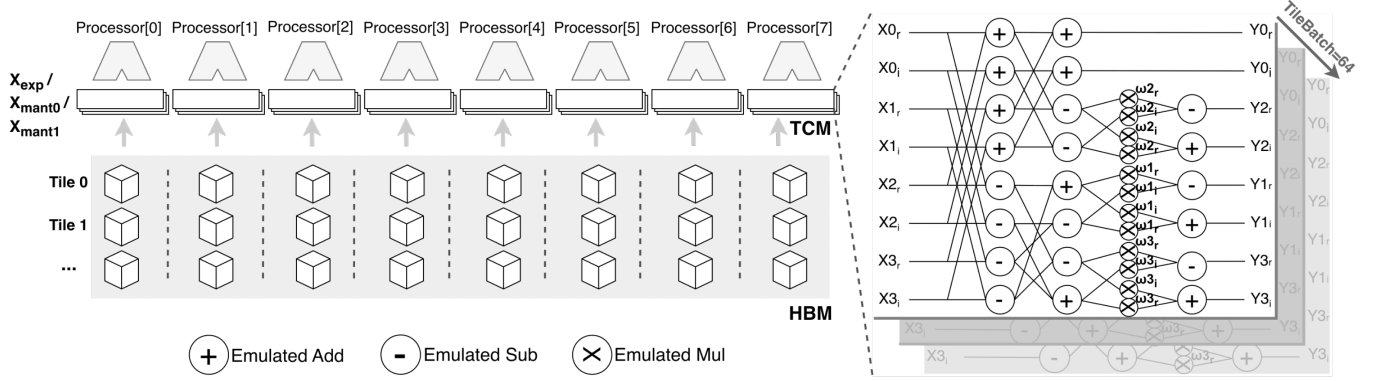


Fig. 3. Batched software-emulated floating-point radix-4 kernel implementation on gpsimdE using the custom C++ operator flow, where each gpsimdE processor loads one tile and precomputed twiddle factors from HBM into its private TCM, performs the radix-4 computation using our emulated operations, and stores the result back to HBM.

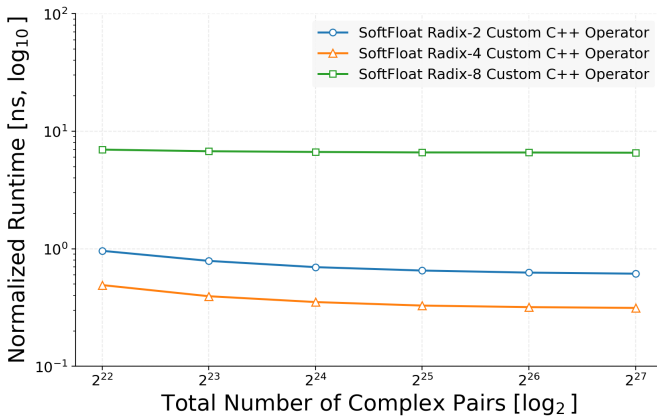


Fig. 4. Normalized runtime of batched software-emulated floating-point radix-2, radix-4, and radix-8 kernels on gpsimdE. Runtime is normalized by the number of equivalent radix-2 butterfly operations to compare different radix sizes on the same basis.

element in each radix group does not require a twiddle multiplication.

As shown in Figure 3, the largest batch size that fits in TCM for the radix-4 kernel is 64, which we refer to as $\text{TileBatch} = 64$. During execution, each gpsimdE processor loads one tile from HBM into its private TCM, performs the radix computation, and stores the result back to HBM. To evaluate larger batch sizes, we assign multiple tiles to each gpsimdE processor, as illustrated in Figure 3. We do not apply software pipelining or prefetching to overlap computation with HBM–TCM transfers because the HBM–TCM communication is synchronous and blocking.

We apply the same methodology to radix-2, radix-4, and radix-8 kernels to study the tradeoff between register pressure and arithmetic intensity. This comparison helps identify which radix is the most suitable fundamental building block for future FFT implementations.

We implement our kernels on an AWS trn1.32xlarge instance, which contains 32 Trainium NeuronCore-v2. The input

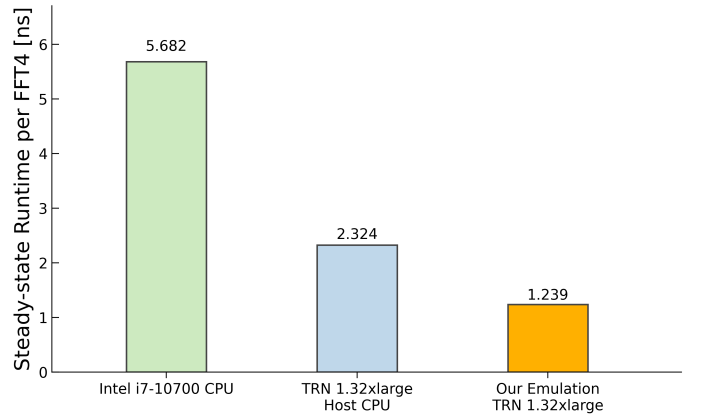


Fig. 5. Steady-state runtime per 1D 4-point FFT. CPU baselines use multi-threaded Intel MKL on the desktop Intel Core i7-10700 CPU and the trn1.32xlarge host Intel Xeon Platinum 8375C CPU, while the Trainium result uses our batched software-emulated floating-point radix-4 kernel on gpsimdE.

batch is partitioned across NeuronCores, and each NeuronCore further distributes its assigned tiles across the eight gpsimdE processors. This mapping allows the kernel to exploit the aggregate throughput of the full trn1.32xlarge device rather than being limited to a single NeuronCore.

IV. RESULTS

This section evaluates the performance of the proposed software-emulated floating-point radix kernels. We compare different radix sizes, evaluate the selected radix-4 kernel against CPU baselines, and discuss scalability limitations of the custom C++ operator flow for larger FFT-style kernels.

A. Radix-Size Selection

Figure 4 compares the performance of the batched emulated radix-2, radix-4, and radix-8 kernels implemented with Trainium custom C++ operators on gpsimdE. Our goal is to identify a radix size that balances arithmetic intensity and register pressure. Larger radix kernels require more live operands and intermediate values, which can increase register

pressure and reduce the maximum tile batch size that fits in TCM. At the same time, they perform more arithmetic per loaded tile and can reduce intermediate register-to-TCM communication, hence increasing arithmetic intensity.

To compare different radix sizes on the same basis, we report normalized runtime per equivalent radix-2 butterfly:

$$\text{Normalized Runtime} = \frac{\text{Kernel Runtime}}{\text{Batch} \times \frac{R}{2} \times \log_2(R)}, \quad (3)$$

where R is the radix size and $\text{Batch} \times R$ is the total number of complex pairs processed by the kernel. The denominator converts a radix- R kernel into the corresponding number of equivalent radix-2 butterfly operations. Therefore, a lower normalized runtime indicates better per-butterfly efficiency.

As the total number of complex pairs increases, the normalized runtime of each kernel approaches a steady-state value, indicating that the fixed overheads are spread across a larger batch. In the steady-state region, radix-2 reaches about 0.613 ns per equivalent radix-2 butterfly, while radix-4 reaches about 0.314 ns. This shows that radix-4 improves efficiency over radix-2 by increasing arithmetic intensity without overwhelming the gpsimdE register resources. In contrast, radix-8 remains much slower, with a steady-state normalized runtime of about 6.558 ns. This suggests that the additional arithmetic intensity of radix-8 is outweighed by increased register pressure, resulting in substantial register spilling with 248 stack loads and 168 stack stores. Overall, radix-4 provides the best balance among the three kernels and is the most suitable radix choice for future FFT implementations.

B. Comparison with CPU Baselines

To put our emulated radix-kernel performance in context, we compare the radix-4 emulated floating-point kernel against multi-threaded Intel Math Kernel Library (MKL) FFT baselines, an optimized numerical library that includes FFT routines [35], on two CPU baselines: the Intel Xeon Platinum 8375C host CPU of the trn1.32xlarge instance and a desktop-class Intel Core i7-10700 CPU. Because MKL does not expose an isolated radix-4 butterfly interface, we use MKL 1D 4-point FFT as the closest CPU reference and set the emulated radix-4 twiddle factors to one for this comparison. All results are reported as steady-state runtime per radix-4 operation, matching the granularity of the emulated radix-4 kernel.

As shown in Figure 5, the Intel Core i7-10700 CPU achieves 5.682 ns per radix-4 operation, while the trn1.32xlarge host CPU achieves 2.324 ns. In comparison, our emulated radix-4 kernel on Trainium reaches 1.239 ns per radix-4 operation. This result indicates that the gpsimdE-based floating-point emulation is not only functional, but also reaches a performance level comparable to, and in this microbenchmark faster than, optimized MKL CPU baselines.

This comparison should be interpreted as a kernel-level reference rather than a full application comparison. Nevertheless, the result provides useful context: despite the overhead of software emulation, the batched radix-4 software-

emulated floating-point kernel can achieve competitive per-kernel throughput when enough parallelism is available. This performance is enabled in part by the high degree of parallelism across Trainium NeuronCores and the high-bandwidth HBM memory system, which supports streaming large batches of radix-kernel tiles efficiently.

C. Custom-Operator Scalability Limitation

While the custom C++ operator flow is effective for evaluating isolated software-emulated floating-point radix kernels, we found it difficult to scale this approach to larger FFT-style kernels. Full FFT pipelines require multiple radix stages with intermediate data reuse across stages. Because the custom C++ operator flow does not expose SBUF, these intermediate values cannot be kept in the on-chip SBUF across stages, leading to repeated HBM-TCM transfers when composing larger kernels. We also observed a compilation limitation: for transform sizes larger than 1,024 points, the generated `.text` section overflows, and the custom-operator benchmark fails to compile. This suggests that the custom C++ operator flow is better suited for lightweight user-defined kernels, while full FFT pipelines composed of our emulated radix kernels push the toolchain close to its practical limits.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a first-look study of software-emulated FP64 radix kernels on AWS Trainium using the custom C++ operator flow targeting gpsimdE. We implemented batched software-emulated floating-point radix-2, radix-4, and radix-8 kernels using Xtensa-IVP intrinsics and evaluated their normalized runtime per equivalent radix-2 butterfly. Our results show that radix-4 provides the best balance between arithmetic intensity and register pressure among the evaluated kernels, reaching 0.314 ns per equivalent radix-2 butterfly and achieving a 1.88 times kernel-level speedup over the MKL-based 1D 4-point FFT baseline on the Trainium host CPU. These results provide early evidence that our floating-point emulation can support FP64 numerical kernels on Trainium efficiently.

For future work, we plan to develop full FFT pipeline kernels using the NKI compilation flow. The custom C++ operator flow is useful for this first-look radix-kernel evaluation, but our results also suggest that it is not the most suitable programming path for larger FFT pipelines. In particular, the lack of SBUF exposure prevents efficient on-chip reuse of intermediate values across FFT stages, and larger generated operators can exceed the custom-operator compilation code-size limits. Therefore, future software-emulated floating-point FFT implementations should move to the NKI compilation flow, where SBUF is explicitly exposed and intermediate data can remain on chip across multiple radix stages. This provides a natural next step from the custom-operator radix-kernel study in this paper toward efficient SBUF-resident emulated floating-point FFT pipelines.

ACKNOWLEDGMENT

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and the U.S. Department of Energy under award number DE-AC05-00OR22725. The author was supported in part by the Department of Energy award DE-AC05-00OR22725, subcontract #CW62109. The authors thank Amazon Web Services for providing access to AWS Trainium resources and technical support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy and DARPA.

REFERENCES

- [1] NVIDIA Corporation, “NVIDIA Hopper Architecture In-Depth,” <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>, 2022, accessed: 2026-07-07.
- [2] Amazon Web Services, “Trainium Architecture,” <https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/arch/neuron-hardware/trainium.html>, 2026, accessed: 2026-07-07.
- [3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, “Accelerating scientific computations with mixed precision algorithms,” *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.
- [4] N. J. Higham and T. Mary, “Mixed precision algorithms in numerical linear algebra,” *Acta Numerica*, vol. 31, pp. 347–414, 2022.
- [5] A. Kashi, H. Lu, W. Brewer, D. Rogers, M. Matheson, M. Shankar, and F. Wang, “Mixed-precision numerics in scientific applications: Survey and perspectives,” *The Journal of Supercomputing*, vol. 82, no. 5, 2026.
- [6] J. R. Hauser, “Berkeley SoftFloat release 3e,” <https://www.jhauser.us/arithmetic/SoftFloat.html>, 2018, accessed: 2026-07-07.
- [7] Y. Hida, X. S. Li, and D. H. Bailey, “Algorithms for quad-double precision floating point arithmetic,” in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 155–162.
- [8] K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, “Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications,” *Numerical Algorithms*, vol. 59, no. 1, pp. 95–118, 2012.
- [9] H. Ootomo, K. Ozaki, and R. Yokota, “DGEMM on integer matrix multiplication unit,” *The International Journal of High Performance Computing Applications*, vol. 38, no. 4, pp. 297–313, 2024.
- [10] D. Mukunoki, “DGEMM without FP64 arithmetic: Using FP64 emulation and FP8 tensor cores with ozaki scheme,” 2025.
- [11] Amazon Web Services, “Neuron Custom C++ Operators,” <https://awsdocs-neuron.readthedocs-hosted.com/en/latest/neuron-customops/index.html>, 2026, accessed: 2026-07-10.
- [12] —, “How to Extend the Functionality of AWS Trainium with Custom Operators,” <https://aws.amazon.com/blogs/machine-learning/how-to-extend-the-functionality-of-aws-trainium-with-custom-operators/>, 2023, accessed: 2026-07-07.
- [13] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, “A survey of numerical linear algebra methods utilizing mixed-precision arithmetic,” *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 344–369, 2021.
- [14] E. Carson and N. J. Higham, “A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems,” *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, 2017.
- [15] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, “Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 476, no. 2243, p. 20200110, 2020.
- [16] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, “DGEMM using tensor cores, and its accurate and reproducible versions,” in *International Conference on Computational Science*, ser. Lecture Notes in Computer Science, vol. 12151. Springer, 2020, pp. 230–248.
- [17] Y. Uchino, K. Ozaki, and T. Imamura, “Performance enhancement of the ozaki scheme on integer matrix multiplication unit,” *The International Journal of High Performance Computing Applications*, vol. 39, no. 3, pp. 462–476, 2025.
- [18] A. Schwarz, A. Anders, C. Brower, H. Bayraktar, J. Gunnels, K. Clark, R. G. Xu, S. Rodriguez, S. Cayrols, P. Tabaszewski, and V. Podlozhnyuk, “Guaranteed DGEMM accuracy while using reduced precision tensor cores through extensions of the ozaki scheme,” in *Proceedings of the Supercomputing Asia / International Conference on High Performance Computing in Asia-Pacific Region*, 2026.
- [19] R. A. Brooker and D. J. Wheeler, “Floating operations on the EDSAC,” *Mathematical Tables and Other Aids to Computation*, vol. 7, no. 41, pp. 37–47, 1953.
- [20] J. W. Backus, “The IBM 701 speedcoding system,” *Journal of the ACM*, vol. 1, no. 1, pp. 4–6, 1954.
- [21] N. Zhang and F. Franchetti, “Code generation for cryptographic kernels using multi-word modular arithmetic on gpu,” in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 476–492.
- [22] N. Zhang, S. Fu, and F. Franchetti, “Towards closing the performance gap for cryptographic kernels between cpus and specialized hardware,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1704–1718.
- [23] Y. Lan, L. Tang, N. Zhang, Y. Eum, J. Hoe, and F. Franchetti, “A risc-v vector extension for multi-word arithmetic,” in *Proceedings of the SC’25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025, pp. 1684–1693.
- [24] The GMP Development Team, “GNU Multiple Precision Arithmetic Library,” <https://gmplib.org/>, accessed: 2026-07-07.
- [25] Y. Hida, X. S. Li, and D. H. Bailey, “Library for double-double and quad-double arithmetic,” Lawrence Berkeley National Laboratory, Tech. Rep., 2008.
- [26] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson, “ARPREC: An arbitrary precision computation package,” Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-53651, 2002.
- [27] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, and P. Bientinesi, “FloatX: A C++ library for customized floating-point arithmetic,” *ACM Transactions on Mathematical Software*, vol. 45, no. 4, pp. 1–23, 2019.
- [28] M. Fasi and M. Mikaitis, “CPFloat: A C library for simulating low-precision arithmetic,” *ACM Transactions on Mathematical Software*, vol. 49, no. 2, pp. 1–32, 2023.
- [29] H. Vandierendonck, “Software-defined floating-point number formats and their application to graph processing,” in *Proceedings of the 2022 International Conference on Supercomputing*. Association for Computing Machinery, 2022, pp. 491–503.
- [30] A. Thall, “Extended-precision floating-point numbers for GPU computation,” in *ACM SIGGRAPH 2006 Research Posters*. New York, NY, USA: Association for Computing Machinery, 2006.
- [31] M. Lu, B. He, and Q. Luo, “Supporting extended precision on graphics processors,” in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. Association for Computing Machinery, 2010, pp. 19–26.
- [32] Z. Lin, A. Sun, X. Zhang, and Y. Lu, “MixPert: Optimizing mixed-precision floating-point emulation on GPU integer tensor cores,” in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. Association for Computing Machinery, 2024, pp. 132–144.
- [33] IEEE Computer Society, “IEEE Standard for Floating-Point Arithmetic,” IEEE Std 754-2019, 2019.
- [34] F. Franchetti and M. Püschel, “FFT (Fast Fourier Transform),” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer, 2011, pp. 658–671.
- [35] Intel Corporation, “Developer Reference for Intel oneAPI Math Kernel Library,” <https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-0/overview.html>, 2024, accessed: 2026-07-07.