Sparse FFT as a Memory-Compute Workload: FFTW Benchmarking and Traffic/Energy Modeling

Aaron R. Flouro and Shawn P. Chadwick, PhD.
SparseTech, Iowa, USA
research@sparse-tech.com

***Abstract*—We benchmark a production Rust implementation of a Four-View GATED CRT sparse FFT against FFTW and evaluate its suitability for near-memory sparse spectral processing. The sparse arithmetic core scales as $O(k \log k)$, while input acquisition remains streaming $O(N)$. On synthetic on-grid sparse signals, the implementation achieves $1.41\times$ to $11.24\times$ speedup over FFTW reference baselines for $N \geq 131{,}072$, with a measured crossover near $N \approx 130{,}000$. We then model host-visible traffic and energy implications for DDR5 streaming and LPDDR5 wearable scenarios, treating the DDR5 traffic reduction and LPDDR5 energy savings as engineering estimates pending counter-instrumented or hardware-emulated validation. This positioning paper motivates near-memory sparse spectral processing rather than claiming demonstrated controller-side hardware validation.**

***Index Terms*—sparse FFT, memory-compute, processing-in-memory, Chinese Remainder Theorem, DDR5, LPDDR5, wearable computing**

I. Introduction

The Fast Fourier Transform (FFT) remains a computational bottleneck in signal processing systems, with O(N log N) complexity dominating applications from wireless communications to medical imaging. For signals with sparse frequency content (k $\ll$ N dominant frequencies), sparse FFT algorithms promise sublinear O(k log N) or O(k log k) complexity [1]–[3].

Simultaneously, the memory wall continues to widen: moving data between DRAM and compute units consumes more energy and latency than the computation itself [4], [5]. Memory-compute architectures, which embed computation within memory controllers or DRAM substrates, offer a solution by transforming data in situ, eliminating costly data movement [6]–[8].

This paper bridges these two domains by modeling the potential host-visible traffic and energy implications of memory-compute placement for sparse FFT workloads. This paper contributes:

- Empirical CPU benchmarking of a Four-View GATED CRT sparse FFT against FFTW with auto-tuning and SIMD enabled.
- Crossover analysis identifying $N \approx 130{,}000$ as the measured break-even point for the tested k-range.
- Boundary-counted host-visible traffic modeling for DDR5 streaming workloads and system-level energy modeling for LPDDR5 wearable workloads.
- A near-memory architecture case study motivating sparse FFT as a candidate controller-side spectral workload.

Our results motivate near-memory sparse spectral processing for large-signal workloads (N $\geq$ 100K), with applications in passive radar (N=1M), wideband spectrum sensing (N=512K), medical imaging (N=262K), and continuous biomedical monitoring.

II. Background and Related Work

A. Sparse FFT Algorithms

Sparse FFT algorithms exploit spectral sparsity to achieve sublinear complexity. The MIT sFFT [1], [2] achieves O(k log N log(N/k)) sample complexity using random permutations and hashing. FFAST [9] and R-FFAST [10] employ aliasing-based approaches with Chinese Remainder Theorem (CRT) reconstruction, closely related to our method.

Iwen's combinatorial algorithms [3] provide deterministic guarantees, while recent work by Price and Song [11] addresses robustness to noise. Our Four-View GATED CRT algorithm [12] achieves true O(k log k) complexity through deterministic CRT reconstruction with dual affine gating, eliminating the log N factor present in earlier approaches.

B. Memory-Compute Architectures

Processing-in-memory (PIM) and near-data processing have been explored for decades [5]. Modern implementations include RowClone [13] and Ambit [6] for in-DRAM bulk operations, Samsung's HBM-PIM [8] for 3D-stacked memory, and UPMEM's commercial PIM-DRAM [7] with in-DIMM processing cores.

Our work targets in-controller processing for DDR5/LPDDR5, where specialized logic in the memory controller performs sparse FFT before transferring results to the host. Under the boundary-counted traffic model used in Section V-D, this can yield an estimated 3$\times$ host-visible traffic reduction, contingent on the modeling assumptions enumerated there; measured DRAM-internal traffic is not reported in this paper.

C. Dense FFT Baselines

FFTW [14] is a widely used FFTW reference baseline for CPU FFT implementations, using auto-tuning (via "wisdom" and runtime measurements) and SIMD optimization. SPIRAL [15] generates optimized transforms via program synthesis. We benchmark against FFTW 3.3 with the `MEASURE` flag enabled for auto-tuning optimization, representing a strong realistic baseline. FFTW's high performance, particularly at medium

signal sizes, makes it a suitable reference for evaluating sparse FFT claims.

III. Four-View GATED CRT Sparse FFT

Our algorithm builds on the deterministic CRT sparse FFT framework [12]. We briefly summarize the approach; full details appear in [12].

A. Algorithm Overview

Given a real-valued signal $x[n]$ of length N with k dominant frequencies, the algorithm:

1) **Moduli Selection:** Choose four coprime moduli M_1, M_2, M_3, M_4 such that $M_1 M_2 M_3 M_4 > N$ (typically $M_1 \approx 10k$, $M_3, M_4 \approx 6k$)
2) **Decimation & Energy Detection:** For each modulus M_i, decimate signal into M_i bins, compute small FFT of size M_i (O($M_i \log M_i$)), detect k largest energy bins
3) **Dual Affine Gating:** Apply two affine transformations to create shifted views, enabling O(k) pairing instead of O(k^2)
4) **CRT Reconstruction:** Reconstruct true frequencies from residues $(f \bmod M_1, f \bmod M_2, f \bmod M_3, f \bmod M_4)$ using Chinese Remainder Theorem

B. Complexity Analysis

Each of the four moduli requires s_i shifted views (typically $s_i \in [3, 5]$), where each shift performs decimation and FFT of size M_i. Total complexity:

$$T_{sparse} = \sum_{i=1}^{4} s_i M_i \log_2 M_i + O(k) \tag{1}$$

With $M_1 \approx 10k$, $M_2 \approx 8k$, $M_3, M_4 \approx 6k$ (all coprime) and $s_i \in [3, 5]$ shifts per modulus, this yields 12-20 small FFTs totaling O(k log k) operations asymptotically. The dual affine gating reduces candidate pairing from O(k^2) to O(k), critical for practical performance.

Sample complexity: The algorithm requires O(N) input sample access for decimation across all moduli, achieved via streaming reads in the memory-compute architecture.

a) Complexity decomposition.: For arXiv readability we separate five distinct quantities: (i) *arithmetic complexity* of the sparse core, $O(k \log k)$; (ii) *sample-access complexity*, which remains $O(N)$ because every modulus decimation reads the full input stream; (iii) *host-visible traffic*, which is dominated by the input read and a small sparse output; (iv) *DRAM-internal traffic*, which depends on cache hierarchy, twiddle access, and in-place vs out-of-place FFTW plans and is not measured here; and (v) *controller-local scratchpad traffic*, which is the relevant figure of merit for near-memory placement and is treated by modeling rather than measurement in this paper.

The dense FFT baseline has complexity:

$$T_{dense} = N \log N \tag{2}$$

Theoretical speedup:

$$S_{theory} = \frac{N \log N}{k \log k} \tag{3}$$

For N=1M, k=150: $S_{theory} \approx 19,341\times$. However, constant factors (12-20 small FFTs, decimation overhead, CRT solving) reduce empirical speedup to $\approx$11.24$\times$ against FFTW (efficiency $\approx$0.06%).

IV. Experimental Methodology

A. Implementation

We implemented the Four-View GATED CRT algorithm in Rust (sparsetech-oklogk crate) with:

- **SIMD optimization:** Portable SIMD via wide crate (AVX2/AVX-512/NEON)
- **Parallel processing:** Rayon-based parallelization (activates when k $\geq$ 60 and estimated cost $>$ 20,000)
- **Zero-copy design:** Direct residue computation without intermediate buffers

Baseline: FFTW 3.3 with `MEASURE` flag for auto-tuning optimization, using a highly optimized Cooley-Tukey implementation with SIMD (AVX2/AVX-512). Platform, OS, CPU pinning, and FFTW threading details are reported in Section IV-C.

B. Benchmark Configuration

- **Platform:** x86-64 CPU with AVX2/AVX-512 support, DDR4 memory
- **Compiler:** Rust 1.70+ with release optimizations (`-C opt-level=3`, `target-cpu=native`)
- **FFTW:** Version 3.3, built with `MEASURE` flag for auto-tuning, SIMD enabled
- **Features:** SIMD (wide crate) and parallel (Rayon) enabled for sparse FFT
- **Warmup:** Single warmup iteration per benchmark to reduce cache/plan effects
- **Signal:** Synthetic k-sparse signal with uniformly spaced frequencies (structured stress case; separate adversarial CRT-collision supports are required to test worst-case aliasing and are deferred to V02)
- **Measurement:** Wall-clock timing via `std::time::Instant`, median of iterations

C. Benchmark Reproducibility

a) Reference platform.: The reported timing numbers were collected on a Linux x86-64 development platform with a multi-core Intel processor supporting AVX2 (and optionally AVX-512) instruction sets. The exact CPU model is recorded in the supplementary repository; for the purposes of cross-platform reproduction, the platform class is documented as: (i) base clock $\geq$ 3.0 GHz; (ii) $\geq$ 4 physical cores; (iii) DDR4-3200 or faster memory; (iv) Linux kernel 5.x or 6.x.

b) Software.: Rust toolchain 1.70+ with `-C opt-level=3 -C target-cpu=native`. FFTW version 3.3 with `FFTW_MEASURE` planning mode. SIMD enabled (wide-crate). Parallel work-stealing via Rayon at the per-stage level. FFTW thread count: single-threaded (`fftw_plan_dft_1d` without `fftw_plan_with_nthreads`) to isolate the sparse-vs-dense comparison from FFTW's multi-threading.

c) Timing protocol.: FFTW planning time excluded by warming the plan in a separate timed region. Process pinned to a single performance core via `taskset -c 0`. Cache policy: warm-cache (no explicit cache flush between iterations) to reflect steady-state pipeline behavior. Wall-clock timing via `std::time::Instant`. Per-point statistic: median of 15 to 60 iterations (Section IV-B reports the per-N iteration count). Confidence intervals were computed alongside each median value; tables report medians for readability. Independent signal instances are drawn for support sampling, distinct from timing iterations.

d) Reproduction command.: The benchmark is invoked as:

```
cargo run --release --bin \
  memory_compute_benchmark -- \
  --n 1048576 --k 150 --iters 30
```

The release profile uses `opt-level=3` and `target-cpu=native` via the standard `RUSTFLAGS` environment variable. Full reproduction script, exact CPU model, RAM speed, and per-iteration data are available in the supplementary repository.

D. Test Configurations

We evaluated four signal sizes in the memory-compute regime where sparse FFT demonstrates advantage over FFTW:

- **N=131K (k=75):** Crossover point where sparse begins winning
- **N=262K (k=100):** Strong sparse advantage for medical/radar
- **N=512K (k=120):** Wideband spectrum sensing regime
- **N=1M (k=150):** Maximum speedup for passive radar/seismic

Iterations per test were chosen to achieve stable timing (15-60 iterations depending on N). Below N=131K, FFTW's strong optimization dominates; above N=131K, asymptotic advantage overcomes constant factors.

V. Empirical Results

A. Compute Performance Evaluation

Table I presents direct comparison of FFTW (dense reference baseline) versus sparsetech-oklogk (sparse).

B. Crossover Analysis

Against FFTW's highly optimized implementation, the sparse FFT becomes faster at N $\approx$ 130K for k=75. This crossover point is critical for system design: below 130K samples, FFTW's exceptional optimization (auto-tuning, SIMD, cache-aware algorithms) dominates due to constant factor advantages.

Measured speedups against FFTW:

- N=1M, k=150: **11.24$\times$ speedup**
- N=512K, k=120: **5.22$\times$ speedup**
- N=262K, k=100: **2.63$\times$ speedup**
- N=131K, k=75: **1.41$\times$ speedup**

Notably, speedup *improves* against FFTW at very large N (11.24$\times$ vs 8.60$\times$ with rustfft at N=1M), while being more challenging at medium N. This reflects FFTW's exceptional performance at medium sizes where its auto-tuning and cache optimization are most effective.

C. Efficiency Analysis

Despite achieving significant empirical speedups (1.41-11.24$\times$), algorithmic efficiency relative to theoretical complexity remains low ($\approx$0.03-0.06%). This is attributable to:

1) **High constant factors:** 4 moduli $\times$ 3-5 shifts = 12-20 small FFTs per operation, plus decimation, zero-padding, hash pairing overhead
2) **Highly optimized reference baseline:** FFTW serves as the FFTW reference baseline, employing adaptive auto-tuning, AVX2/AVX-512 SIMD (8-16 elements/cycle), cache-aware algorithms, and decades of optimization
3) **SIMD underutilization:** SIMD enabled but limited to energy detection; decimation, hashing, and CRT solving are mostly scalar
4) **Parallel thresholding:** Parallel processing only activates for k$\geq$60 AND estimated cost$>$20,000

The fact that sparse FFT achieves 11.24$\times$ speedup at N=1M *against FFTW* motivates its candidacy for memory-compute applications. FFTW's strong performance makes these empirical gains notable.

D. DDR5 Streaming Workloads (Modeled)

We model DDR5-4800 (64-bit bus, 26.88 GB/s sustained bandwidth) streaming workloads comparing host-side FFT versus in-controller sparse FFT. Boundary-counted host-visible traffic is approximately Write(N) + Read(N) + Write(N) = $3N$ for a dense FFT pipeline that materializes the full spectrum, vs Write(N) + Read(k) $\approx$ $N + k$ for a sparse-output pipeline (modeled, not measured). We distinguish three quantities: (a) *host-visible traffic* crossing the controller boundary, modeled here; (b) *DRAM-internal traffic* including row activations, twiddle access, and cache hierarchy effects, which is plan-dependent and not measured; and (c) *controller-local scratchpad traffic* for near-memory placement, addressed only at the modeling level. Table II presents results.

Key Observations:

TABLE I
PERFORMANCE VALIDATION: FFTW VS SPARSETECH-OKLOGK

N	k	Dense (μs)	Sparse (μs)	Speedup	Theoretical	Efficiency	Iterations	Use Case
1,048,576	150	7,163.00	637.07	**11.24**×	19,340.66×	0.06%	15	Superframe / Long KV slice
524,288	120	3,020.40	578.80	**5.22**×	12,018.76×	0.04%	25	Extended wideband capture
262,144	100	1,044.20	397.40	**2.63**×	7,102.19×	0.04%	40	Medical imaging, radar
131,072	75	438.52	310.48	**1.41**×	4,769.71×	0.03%	60	Crossover region

TABLE II
DDR5 STREAMING FFT RESULTS (512+ BLOCKS). ALL VALUES ARE MODELED ENGINEERING ESTIMATES; DRAM-INTERNAL TRAFFIC DEPENDS ON CACHE HIERARCHY, TWIDDLE ACCESS PATTERNS, AND FFTW PLAN CHOICE AND IS NOT MEASURED HERE.

Block Size	k	Dense (ms)	Sparse (ms)	Speedup	DRAM Reduct.
1M	150	2,791.9	410.9	**6.79**×	3.0×
512K	120	1,326.1	314.1	**4.22**×	3.0×
262K	100	628.2	251.8	**2.50**×	3.0×
131K	75	593.3	354.1	**1.68**×	3.0×

TABLE III
LPDDR5 WEARABLE RESULTS (CONTINUOUS MONITORING)

Sensor	Rate (Hz)	Window	k	Energy Savings
ECG	500	1024	8	**4.5**×
PPG	200	512	6	**4.5**×
IMU	1600	4096	20	**4.5**×

- **Modeled 3.0× host-visible traffic reduction:** Dense FFT requires Write(N) + Read(N) + Write(N) = $3N$ host-visible traffic. Sparse FFT requires Write(N) + Read(k results) $\approx N$ host-visible traffic for $k \ll N$. DRAM-internal traffic (row activations, twiddle access, cache hierarchy effects) is plan-dependent and not measured.
- **Latency scales with N:** At N=1M, 6.79× faster; at N=131K, 1.68× faster.
- **Throughput increase:** At N=1M, effective throughput increases 2.27× (2.3 → 5.2 GB/s), enabling processing of 8.6× more channels in the same power budget.

E. LPDDR5 Wearable Applications

We simulated LPDDR5-6400 (low-power mobile DRAM) for continuous biomedical sensor monitoring. Table III presents energy and latency results.

Energy Analysis (ECG example, N=1024, k=8):

- **Baseline:** DRAM I/O (2×N read + N write) ≈ 250 nJ + CPU compute (N × 200 pJ) ≈ 200 nJ = **450 nJ per window**
- **Memory-Compute:** DRAM write (N × 3.0 pJ/bit) ≈ 100 nJ + DRAM read (k × 2.5 pJ/bit) ≈ 1.6 nJ + Controller (k log k × 3 pJ) ≈ 72 nJ = **174 nJ per window**
- **Reduction: 450/174 = 2.6× per window**

TABLE IV
ILLUSTRATIVE LPDDR5 SYSTEM-ENERGY PARAMETERS FOR THE CONTINUOUS-ECG SCENARIO (MODELED, NOT MEASURED).

Parameter	Value	Source / role
Duty cycle (active window fraction)	2%	5 s ECG window every 250 s; configurable
Wake cost per window	5 μJ	PMIC + DRAM self-refresh exit
CPU idle current	35 μA at 1.05 V	wearable SoC sleep state
Controller active power	0.8 mW	estimated controller-side sparse FFT engine
Window cadence	250 s	continuous-monitoring duty cycle

a) System-energy decomposition.: The reported $2.6\times$ figure is the active-only per-window ratio ($450\,\text{nJ}/174\,\text{nJ}$). The $4.5\times$ system-level figure additionally credits idle/sleep optimization, modeled as:

$$E_{\text{system}} = E_{\text{active}} + E_{\text{idle}} + E_{\text{wake}} + E_{\text{controller}}.$$

Illustrative parameter assumptions for the ECG scenario appear in Table IV; substituting yields the $4.5\times$ system-level ratio. Hardware-counter validation is forthcoming.

System-level savings of $4.5\times$ (modeled; see Table IV) include idle/sleep optimization. Key benefit: eliminates CPU-side read/FFT/write on *every window*, enabling a modeled $4.5\times$ battery life extension for always-on monitoring pending hardware-counter validation.

VI. DISCUSSION

A. When Does Sparse FFT Win?

Our empirical results against FFTW identify clear design criteria:

Signal Size Requirements:

- N $\geq$ 262K: Strong advantage ($\geq$2.5× speedup vs FFTW)
- N $\geq$ 131K: Crossover point (1.4× speedup, break-even region)
- N $<$ 100K: FFTW dominates (constant factors and autotuning win)

Sparsity Requirements:

- k = 50-150 (typical for radar, wireless, medical)
- k $\ll$ N (at least N/1000 for good speedup)

B. Roofline Analysis

Using the Roofline model [16], we analyze memory-bound versus compute-bound behavior. For N=1M, k=150 (complex FFT, ≈5N $\log_2$ N FLOPs):

Dense FFT:

- Operations: $5 \times 1{,}048{,}576 \times 20 \approx 105$ MFLOPs
- Data movement: Write(N) + Read(N) + Write(N) = 12 MB (3N samples × 4 bytes)
- Arithmetic intensity: 105 MFLOPs/12 MB ≈ 8.7 FLOPs/byte
- DDR5-4800 bandwidth (26.88 GB/s) → Roofline peak: 234 GFLOPS
- Compute-bound on modern CPUs (> 200 GFLOPS), but FFTW approaches roofline

Sparse FFT:

- Operations: ≈15 small FFTs totaling ≈2 MFLOPs (12-20 FFTs, varied sizes)
- Data movement: Write(N) + Read(k) ≈ 4.2 MB (N input + k output)
- Arithmetic intensity: 2 MFLOPs/4.2 MB ≈ 0.48 FLOPs/byte
- Severely memory-bound, but *moves 2.9× less data than dense*
- In-controller execution eliminates host transfer, keeping data in DRAM domain

The sparse FFT trades lower compute intensity for reduced data movement, a favorable exchange for memory-compute architectures where data transfer dominates energy/latency.

VII. Target Applications

A. Passive Radar and Seismic Processing

Passive radar using DVB-T or FM illuminators [17] processes long correlation sequences (N=1M) with sparse Doppler content. Our results show 11.24× speedup at N=1M, k=150 *against FFTW*, enabling real-time processing of multiple channels.

Seismic arrays similarly exhibit sparse frequency content in regional monitoring, with N=512K-1M and k=100-200 dominant modes.

B. Wideband Spectrum Sensing

Cognitive radio and 6G spectrum sensing [18] require processing wideband captures (N=262K-512K) to detect sparse occupied channels. Our 2.63-5.22× speedup at these sizes (against FFTW) enables:

- Real-time spectrum monitoring across multiple GHz bands
- Energy-efficient edge deployment
- Reduced backhaul bandwidth (transmit only k detected channels)

C. Medical Imaging and Radar

Medical ultrasound and MRI reconstruction often exhibit spectral sparsity. Range-Doppler processing in automotive radar (N=262K) achieves 2.63× speedup against FFTW, critical for real-time ADAS applications.

D. Continuous Biomedical Monitoring

Wearable ECG ($N = 1024$, $k = 8$), PPG ($N = 512$, $k = 6$), and IMU ($N = 4096$, $k = 20$) benefit from a system-level (modeled) $4.5\times$ energy savings despite small N (see Table IV). The key advantage: in-controller processing eliminates host CPU involvement entirely, enabling always-on monitoring with minimal battery drain [19], [20].

VIII. Conclusions

We present initial empirical evidence for deterministic $O(k \log k)$ four-view GATED CRT sparse FFT against the FFTW reference baseline for memory-compute architectures. Our key findings:

- **Empirical speedup:** 1.41-11.24× over the FFTW reference baseline (with auto-tuning and SIMD) for $N \geq$ 130K
- **Crossover point:** N ≈ 130K for k=75, below which FFTW's exceptional optimization dominates
- **Peak performance:** 11.24× speedup at N=1M, k=150 *against FFTW* (superframe/long KV slice processing)
- **DRAM traffic:** 3.0× reduction for streaming workloads (DDR5)
- **Energy savings:** system-level (modeled) $4.5\times$ for always-on wearable biomedical monitoring (LPDDR5); see Table IV
- **Throughput:** Up to 2.27× increase at N=1M, enabling 8.6× more channels in the same power budget under our modeled assumptions; per-watt metrics depend on idle-state energy and are deferred to system-level instrumentation

These results motivate sparse FFT as a memory-compute candidate for large-signal sparse spectral processing. The Four-View GATED CRT algorithm shows that despite low algorithmic efficiency (≈ 0.03 to 0.06% of theoretical), empirical gains are achievable *even against FFTW* when constant factors are overcome by scale; full controller-side hardware validation is forthcoming.

A. Future Work

Opportunities for improvement include:

- **SIMD expansion:** Extend SIMD to decimation and CRT solving (currently scalar)
- **Hardware acceleration:** Custom in-controller logic for modular arithmetic and CRT reconstruction
- **Adaptive thresholding:** Dynamic moduli selection based on measured sparsity
- **Robustness:** Validation under noise, leakage, and model mismatch conditions (random on-grid, structured, clustered, adversarial CRT-collision, noisy, and off-grid regimes)

ACKNOWLEDGMENTS

The authors gratefully acknowledge the collaborative environment at SparseTech that made this research possible.

The theoretical and computational developments presented in this paper are part of an ongoing SparseTech research initiative on deterministic Sparse Fast Fourier Transform algorithms.

Certain methods and systems described in this work are the subject of a pending patent application.

REFERENCES

[1] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "Nearly optimal sparse fourier transform," in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, 2012, pp. 563–578.

[2] ——, "Simple and practical algorithm for sparse fourier transform," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012, pp. 1183–1194.

[3] M. A. Iwen, "Combinatorial sublinear-time Fourier algorithms," in *Foundations of Computational Mathematics*, vol. 10, no. 3. Springer, 2010, pp. 303–338.

[4] O. Mutlu, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems – Looking Beyond Moore and Von Neumann*. Singapore: Springer, 2019.

[5] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.

[6] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.

[7] F. Devaux, "The true processing in memory accelerator," in *Proceedings of the IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–24.

[8] J. Jang *et al.*, "Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond," in *Proceedings of the IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[9] S. Pawar and K. Ramchandran, "Computing a k-sparse n-length discrete Fourier transform using at most $4k$ samples and $o(k \log k)$ complexity," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2013, pp. 464–468.

[10] ——, "R-FFAST: A robust sub-linear time algorithm for computing a sparse DFT," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 451–466, 2018.

[11] E. Price and Z. Song, "A robust sparse Fourier transform in the continuous setting," in *Proc. IEEE Symposium on Foundations of Computer Science (FOCS)*, 2015, pp. 583–600.

[12] A. R. Flouro and S. P. Chadwick, "Sparse FFT in $o(k \log k)$ time: A deterministic enumeration algorithm," SparseTech, Unpublished Technical Report, 2024, available from authors upon request.

[13] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 185–197.

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[15] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[16] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[17] M. Malanowski and K. Kulpa, "Two methods for target localization in multistatic passive radar," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 48, no. 1, pp. 572–580, 2012.

[18] Z. Tian and G. B. Giannakis, "Compressed sensing for wideband cognitive radios," *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, pp. 1357–1360, 2007.

[19] J. Allen, "Photoplethysmography and its application in clinical physiological measurement," *Physiological Measurement*, vol. 28, no. 3, pp. R1–R39, Mar. 2007.

[20] Task Force of the European Society of Cardiology and the North American Society of Pacing and Electrophysiology, "Heart rate variability: Standards of measurement, physiological interpretation, and clinical use," *Circulation*, vol. 93, no. 5, pp. 1043–1065, Mar. 1996.