Accuracy–Performance Trade-offs in Ozaki-Inspired FP16 Matrix Multiplication on Qualcomm Mobile Hardware

Asli Elif Ozgur

Durham University

Dr Christopher Marcotte

August 2026

Abstract

Low-precision arithmetic can accelerate matrix multiplication but increases rounding error. This study tested direct and two-component split multiplication on a Qualcomm SM8750P platform using square matrices of size 256, 512 and 1024. Both methods used QAIRT's 16-bit floating-point GPU configuration; the split graph evaluated four component products. Five independent matrix pairs were tested at each size, with a native eight-thread CPU FP64 implementation as a baseline. Splitting reduced relative error by approximately 20% at all three sizes, while taking 1.10×, 1.55× and 2.19× as long as direct FP16. Direct GPU FP16 was 17.4× to 67.4× faster than the documented CPU baseline. Cost per useful floating-point operation fell with size for both GPU methods, indicating improved device utilisation. Splitting improved accuracy without approaching FP64, and its runtime penalty increased with matrix size.

1. Introduction

General matrix multiplication (GEMM) is central to scientific computing and machine learning, but numerical precision affects both speed and accuracy. FP16 uses fewer bits than FP32 or FP64, reducing storage and enabling specialised low-precision arithmetic [1–3]. The trade-off is more aggressive rounding. When many products are accumulated, small errors can combine into a visible difference in the final matrix.

This tension is especially relevant on mobile hardware. Phones and embedded devices can carry out inference, image processing and signal analysis locally, where latency, memory and thermal limits matter. A GPU can make a calculation practical, but the numerical quality exchanged for speed still needs to be understood. The cited studies evaluated CPU systems or NVIDIA tensor-core GPUs rather than this two-component experiment on a mobile Qualcomm GPU [1–3]. This study therefore tests the idea on a different software and hardware stack.

One response is to split each input into several low-precision components, multiply the components separately and add the partial products. Ozaki and colleagues developed and generalised error-free transformations for matrix multiplication, and later studies used related schemes to exploit fast low-precision units while retaining more information [1–3]. Those high-accuracy implementations use more than two components. The present experiment keeps only one residual component per input, so it asks whether a small accuracy recovery justifies four low-precision products.

The broader design question is whether mobile-GPU FP16 can provide competitive GEMM performance and whether component splitting can reach a target output accuracy at lower cost than higher-precision execution. This study examines one restricted point in that design space.

Research question: On the tested Qualcomm platform, does mobile-GPU FP16 provide competitive GEMM runtime relative to the documented CPU FP64 baseline, and how much numerical accuracy does a one-residual, two-component decomposition recover for its additional runtime?

2. Background

2.1 GEMM, precision and work

For square N × N matrices, C = AB and each output is a dot product:

$$C_{ij} = \sum_{l=1}^{N} A_{il} B_{lj}$$

The calculation forms N^2 outputs and requires N^3 multiplications plus $N^2(N-1)$ additions, or $N^2(2N-1) \approx 2N^3$ floating-point operations. The arithmetic count is therefore $O(N^3)$, while matrix storage is $O(N^2)$. Reusing matrix tiles increases work per byte as N grows; at a fixed shape, narrower FP16 operands can also reduce bytes moved and raise attainable arithmetic intensity. This study did not measure memory traffic, so it does not classify any tested case as memory-bound.

Floating-point calculations are approximations. Results can differ with precision, evaluation order and hardware because many real numbers cannot be represented exactly and addition is not associative [4]. IEEE binary16 has 11 bits of significand precision for normal values, whereas binary64 has 53 [5]. Direct FP16 therefore benefits from compact data and accelerator support but has fewer significant digits and a narrower dynamic range than FP32 or FP64.

Rounding can occur when inputs are represented, when products are formed and as partial sums are accumulated [6]. The numerical error depends on the input values as well as the arithmetic used for multiplication and accumulation.

2.2 Two-component splitting

The Ozaki approach rewrites a matrix product as a sum of products between shorter-precision components [1]. Utsugiri and Kouya divide longer-precision matrices into shorter-precision parts so that optimised low-precision GEMM kernels can be used [2]. Mukunoki et al. implemented double-precision (DGEMM) and single-precision (SGEMM) algorithms on NVIDIA Tensor Cores using FP16 inputs and FP32 accumulation; their higher-accuracy variants also use higher-precision or accurate summation [3]. Accuracy therefore depends on component count, decomposition, product precision and accumulation.

In this experiment, each FP32 input x was decomposed into a high component and a rounded residual:

$$x_1 = FP16(x), \quad x_2 = FP16(x - x_1)$$

Writing $\hat{A} = A_1 + A_2 = A - \Delta A$ and $\hat{B} = B_1 + B_2 = B - \Delta B$ makes the reconstruction error explicit. The split graph evaluated:

$$C_{split} = A_1B_1 + A_1B_2 + A_2B_1 + A_2B_2$$

If E_{arith} collects rounding from the four products and three additions, the output error is:

$$C_{split} - AB = -\Delta A \cdot B - A \cdot \Delta B + \Delta A \cdot \Delta B + E_{arith}$$

The decomposition error is therefore multiplied by A and B and can be important when cancellation makes AB small relative to the operands. Classical numerical analysis can bound such terms, but the deployed error was also measured because QAIRT did not disclose its internal accumulator precision or kernel schedule.

The direct graph contained one MatMul; the split graph contained four MatMul and three Add operations. Both used QAIRT's 16-bit floating-point GPU setting. Deep Learning Container (DLC) metadata identified Float16 graph tensors and a final Float32 conversion for the app-readable output. The fixed, one-residual expansion is therefore described as “Ozaki-inspired” rather than a full error-free, multi-component Ozaki scheme [1–3].

3. Method

3.1 Hardware and software

Models were prepared in PyTorch 1.13.1+cpu and exported with ONNX 1.16.1 as fixed-shape graphs (opset 17). QAIRT 2.47.0.260601 compiled them using --float_bitwidth 16 and --target_backend GPU. The device reported a Qualcomm SM8750P, Android 15 (API 35), ARM64 and an Adreno GPU. Each size required a separate model; graph audits confirmed one MatMul node for direct and four for split. QIDK materials supplied the deployment workflow [7]. Here, “GPU FP16” names this configuration rather than a verified accumulator width.

3.2 Inputs and reference calculation

Square matrices with N = 256, 512 and 1024 were generated from a normal distribution with mean 0 and standard deviation 0.125. Five case identifiers were used for each size (20260732–20260736); the NumPy generator seed for each case was identifier + N. This produced 15 paired direct/split cases. FP64 matrices were sampled first, GPU inputs were derived by casting them to FP32, and the original FP64 matrices formed the NumPy reference product and CPU inputs.

Accuracy was measured as relative Frobenius error, where $\|X\|F = \sqrt{(\sum_{i,j}|x_{ij}|^2)}$ is the L2 norm of the flattened matrix:

$$relative\ error = \|C_{method} - C_{ref}\|_F / \|C_{ref}\|_F$$

The direct model received the FP32 arrays derived from the sampled matrices. For the split model, high and residual components were prepared from the same arrays and stored as separate FP32 interface files whose values were FP16-rounded. Every direct/split pair therefore used identical underlying matrix content.

3.3 Timing protocol

For each GPU case, QAIRT executed the model ten times. The first execution was treated as cold and excluded; the remaining nine executions were averaged. Excluding the first run was intended to reduce start-up effects in the repeated-execution measurement. Direct and split cases were run in a randomised order generated before testing, reducing the risk that one method was always favoured by an earlier, cooler device state.

The CPU FP64 baseline was a hand-written C++ triple-loop DGEMM, row-partitioned across eight std::thread workers and using a pre-transposed right-hand matrix; it called neither BLAS nor LAPACK. Each CPU case had one warm-up followed by three measured executions. Timing excluded input/output file I/O and matrix transposition but included thread creation. The resulting GPU speedups therefore apply only to these documented implementations.

3.4 Statistical analysis and validation

Means were first calculated within each matrix case and then aggregated across the five cases at each size. The matrix pair, not each repeated timing, was treated as the independent experimental unit. Two-sided 95% confidence intervals used Student's t distribution with four degrees of freedom. Error reduction, split-to-direct runtime and GPU speedup were calculated as paired quantities within each case before aggregation. Output files and graph profiles were checked against the run manifest. An early pilot run was discarded after a line-ending error caused both method labels to select the split model; the corrected dataset contains 15 direct and 15 split GPU cases and is the only dataset used here. Matrix seeds, the randomised run manifest, raw profiles, analysis scripts and per-case summaries were retained with the project record.

Attempts to obtain power data through the available Android interfaces were unsuccessful. Values were zero or implausibly quantised and could not be linked reliably to the processor or whole board, so no energy result was inferred.

4. Results

Table 1. Mean performance and numerical error across five independent matrix cases. Time intervals are 95% confidence-interval half-widths based on Student's t distribution (n=5).

N	Method	Mean time (ms)	95% CI half-width (ms)	Relative error
256	CPU FP64	12.0819	±4.0998	3.153×10^{-16}
256	Direct GPU FP16	0.6929	±0.0114	3.593×10^{-4}
256	Split GPU FP16	0.7649	±0.1221	2.882×10^{-4}
512	CPU FP64	37.9328	±6.6098	7.605×10^{-16}
512	Direct GPU FP16	1.3082	±0.0874	3.596×10^{-4}
512	Split GPU FP16	2.0182	±0.1318	2.869×10^{-4}
1024	CPU FP64	212.4624	±6.5099	1.130×10^{-15}
1024	Direct GPU FP16	3.1642	±0.3223	3.595×10^{-4}
1024	Split GPU FP16	6.9006	±0.3721	2.878×10^{-4}

The split method improved accuracy consistently. Relative error fell from about 3.59×10^{-4} for direct FP16 to about 2.88×10^{-4} for split FP16 at all three sizes. Paired error reductions were 19.78% ± 0.17 percentage points at N=256, 20.20% ± 0.14 at N=512, and 19.96% ± 0.06 at N=1024 (95% confidence intervals). The effect was consistent across the five Gaussian matrix pairs tested at each size.

Matrix size had little effect on the mean FP16 relative error for this input distribution: both FP16 curves were almost flat from N=256 to N=1024. The difference between methods was also similar at each size. By contrast, the native CPU FP64 baseline differed from the NumPy FP64 reference by between 3.15×10^{-16} and 1.13×10^{-15} in relative error, roughly 11–12 orders of magnitude below the GPU FP16 results. For these cases, this shows that the split method recovered only part of the lost precision.

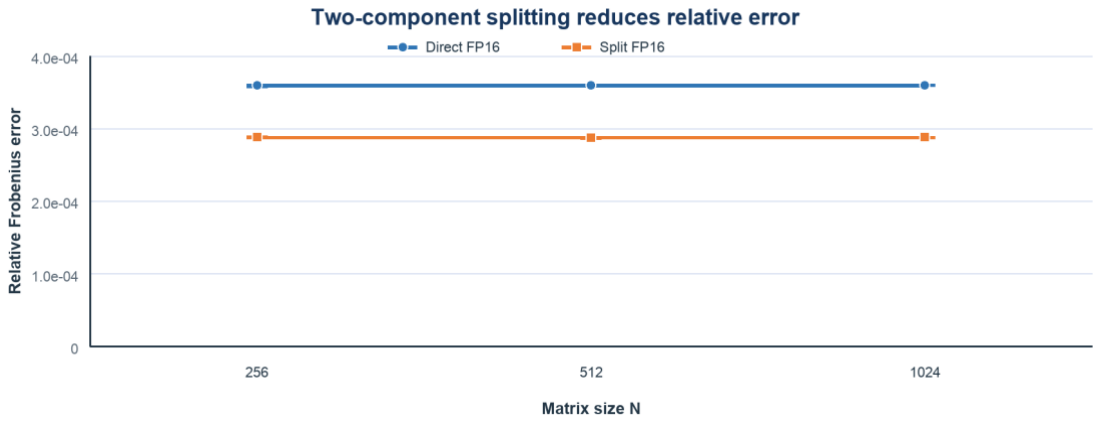

Figure 1. Mean relative Frobenius error (95% CI; n=5).

The speed cost was not constant. Split FP16 took 1.10× ± 0.18× the direct runtime at N=256, 1.55× ± 0.19× at N=512 and 2.19× ± 0.25× at N=1024. At N=256 the 95% confidence interval included parity

with direct execution, and the between-case coefficient of variation of the five split case means was 12.9%. The overhead was clearer at the two larger sizes.

Across the full size range, direct GPU time increased from 0.69 ms to 3.16 ms, whereas split time increased from 0.76 ms to 6.90 ms. Doubling N multiplies nominal GEMM work by eight, but direct time rose by 1.89× and then 2.42×; split time rose by 2.64× and then 3.42×. These sizes were therefore not in a simple asymptotic compute-bound regime. Fixed launch costs, occupancy, memory traffic and kernel selection may all contribute.

A heuristic model used fixed, N^2 data-movement and N^3 arithmetic terms:

$$T_d = L + MN^2 + CN^3, \quad T_s = L + 2MN^2 + 4CN^3$$

The model reproduced the six mean times with 0.09 ms root-mean-square error. With only three sizes and no kernel trace, this fit does not establish that QAIRT fused the products or used a block GEMM.

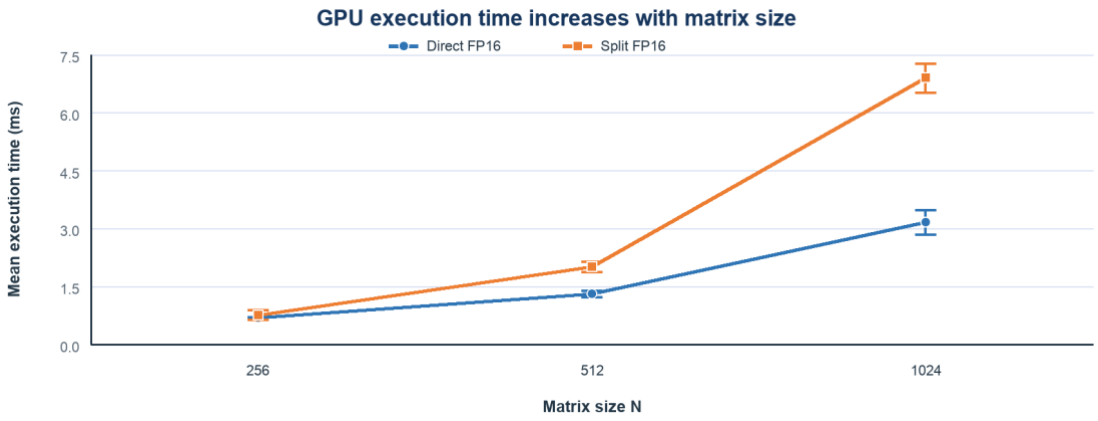

Figure 2. Mean QAIRT GPU execution time (95% CI; n=5).

Direct GPU FP16 was faster than the CPU FP64 baseline at every size: 17.4× ± 6.0×, 29.1× ± 6.2× and 67.4× ± 4.8× for N=256, 512 and 1024 respectively. These comparisons combine different precisions, hardware and implementations. GPU timing covered QAIRT backend execution, not FP64-to-FP32 preparation or the complete application data path, so the figures are not a general CPU-versus-GPU comparison.

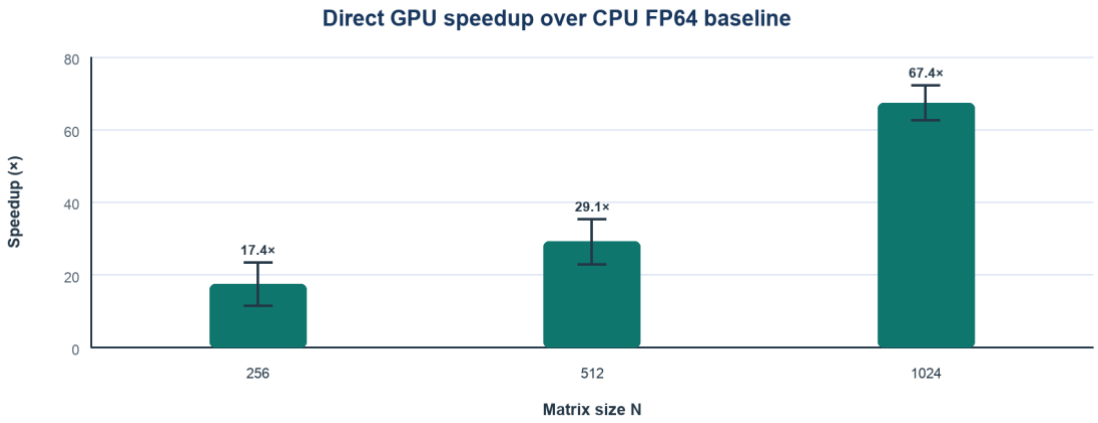

Figure 3. Direct GPU FP16 speedup over the CPU FP64 baseline (95% CI; n=5).

For comparison across sizes, execution time was divided by the same useful-work count, $2N^3$, for both GPU methods. Direct cost fell from 20.65 to 1.47 ps per useful FLOP, while split cost fell from 22.80 to 3.21 ps. This shows improving GPU utilisation with size; it does not change the split/direct ratio at a given N. Dividing split time by its larger executed workload would instead measure hardware throughput, not the cost of solving the same AB problem.

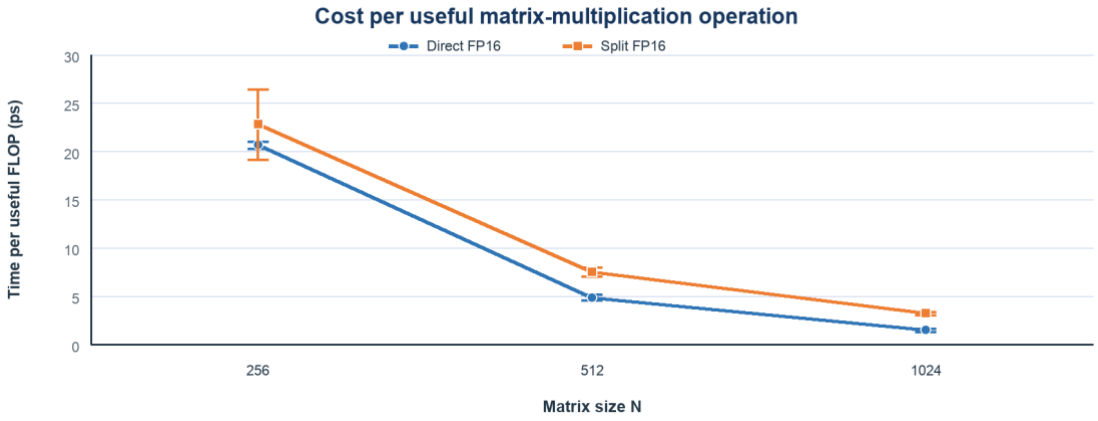

Figure 4. QAIRT GPU time per useful FLOP (95% CI; n=5; common useful-work count $2N^3$; lower is better).

5. Discussion

Retaining one rounded residual for each input reduced relative error by approximately 20% at every tested size. The final GPU error nevertheless remained on the order of 10^{-4}, while the native CPU FP64 baseline differed from the NumPy FP64 reference by only about 10^{-15}. Two components improved this FP16 configuration without making it equivalent to FP64.

The direction agrees with the motivation of Ozaki-style methods: low-precision components can preserve information lost in one direct rounding step [1–3]. A direct numerical comparison with full Ozaki schemes would be inappropriate because the cited studies use different component counts, arithmetic, inputs, accuracy targets and hardware. The present result applies only to this two-component QAIRT configuration.

The runtime measurements make the accuracy gain a trade-off. At N=256, the interval around the 1.10× ratio included parity, whereas the penalty reached 2.19× at N=1024. The rising ratio is consistent with extra arithmetic becoming more visible as fixed costs are amortised, but it does not identify the backend mechanism.

The value of this trade-off depends on the application. At N=512, splitting costs about 0.71 ms for the 20% error reduction. If m components per operand are expanded independently, m^2 low-precision products are required. The conservative model $Tm \approx m^2 T$ direct suggests that about 3, 4 and 7 residual terms at N=256, 512 and 1024 respectively could remain below the measured CPU FP64 time. This is only a runtime budget: zero and one residual were tested, the CPU was not tuned BLAS, and additions, storage, fusion, accuracy and energy may change with m.

Lower latency could reduce energy if the processor returns sooner to a low-power state. However, CPU FP64 and GPU FP16 need not consume the same power per FLOP, so runtime alone cannot establish energy efficiency. Reliable vendor telemetry or an external high-resolution monitor would be needed to report joules per multiplication.

6. Limitations

The study used one Qualcomm platform, three square sizes and five random cases per size. Runtime for a fixed dense graph is mainly shape-driven, so the timing results may apply to other dense matrices with the same layout. Numerical error is value-dependent, however: the observed reduction is specific to the sampled Gaussian inputs and may differ with wider value ranges or structured cancellation. Only relative Frobenius error was reported; a maximum-entry or downstream-task metric could support a different judgement.

The CPU baseline was a transparent eight-thread implementation rather than an optimised BLAS library, and no matched CPU FP32 SGEMM was run. CPU and GPU operands came from the same FP64 samples, but GPU operands were first rounded to FP32. The split graph also exposed four component inputs rather than two, increasing its buffer footprint. QAIRT timing excluded the complete application pipeline, and its metadata did not reveal internal memory traffic or accumulator width. The two-component result should not be generalised to full Ozaki algorithms.

7. Conclusion

On the Qualcomm SM8750P GPU, one residual component reduced direct FP16 matrix-multiplication error by about 20% across N=256, 512 and 1024, but remained far less accurate than FP64. The runtime ratio rose from an uncertain 1.10× to a clear 2.19×, while cost per useful FLOP fell with size for both methods. Splitting therefore offered an intermediate accuracy–runtime option rather than a solution to FP16 error. Testing additional components, a tuned FP32/FP64 CPU baseline, larger sizes and valid power measurement are the most useful next steps.

References

[1] K. Ozaki, T. Ogita, S. Oishi and S. M. Rump, 'Generalization of error-free transformation for matrix multiplication and its application', Nonlinear Theory and Its Applications, IEICE, 4(1), pp. 2–11, 2013. https://doi.org/10.1587/nolta.4.2

[2] T. Utsugiri and T. Kouya, 'Acceleration of Multiple Precision Matrix Multiplication using Ozaki scheme', arXiv:2301.09960 [math.NA], 2023. https://doi.org/10.48550/arXiv.2301.09960

[3] D. Mukunoki, K. Ozaki, T. Ogita and T. Imamura, 'DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions', in High Performance Computing: ISC High Performance 2020, Lecture Notes in Computer Science, vol. 12151, Springer, Cham, 2020, pp. 230–248. https://doi.org/10.1007/978-3-030-50743-5_12

[4] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics, 2002. https://doi.org/10.1137/1.9780898718027

[5] IEEE, IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84, 2019. https://doi.org/10.1109/IEEESTD.2019.8766229

[6] PyTorch, 'Numerical accuracy', PyTorch documentation. https://docs.pytorch.org/docs/stable/notes/numerical_accuracy.html (accessed 22 August 2026).

[7] Qualcomm Innovation Center, Inc., 'Qualcomm Innovators Development Kit (QIDK)', GitHub repository. https://github.com/qualcomm/qidk (accessed 22 August 2026).