FASTKRON: EFFICIENT QUANTIZATION WITH KRONECKER-FACTORED HESSIANS

Anonymous authors
Paper under double-blind review

ABSTRACT

We accelerate a family of algorithms for neural network quantization which utilize a two-sided version of the GPTQ/LDLQ algorithm. Standard GPTQ-style adaptive rounding uses one-sided correlation information derived from input activations. A natural two-sided extension can additionally capture correlations across output channels. It utilizes a general Kronecker-factored approximation of the weight matrix curvature. This approach has been used in BoA and YAQA. But making a concrete algorithmic implementation of this two-sided GPTQ variant is nontrivial. BoA uses a large number of sequential steps, while YAQA improves over the sequential depth but still has quartic total cost. We introduce FastKron, an efficient algorithmic implementation that combines anti-diagonal parallelism with a recursive divide-and-conquer construction. For an $m \times n$ weight matrix, FastKron uses $\mathcal{O}(m+n)$ sequential steps while reducing the total work from $\mathcal{O}(m^2n^2)$ to $\mathcal{O}(mn(m+n))$. Thus, it matches the cubic scaling of GPTQ while exploiting richer curvature information. Moreover, FastKron is modular with respect to both the base quantizer and the Hessian estimator. We also provide practical benchmarks, consider a range of Hessian approximations that FastKron can be used with, and provide an efficient technique to compute these Hessians.

1 INTRODUCTION

Consider the problem of quantizing the weights of a neural network post-training. We are given a trained neural network, and for each linear unit with weight matrix $W \in \mathbb{R}^{m\times n}$, we wish to replace W by another weight matrix V of low numerical precision, while maintaining overall accuracy of the network. Each linear unit W is quantized separately. However, simply rounding each entry of W to the closest element of the quantization alphabet is suboptimal. Instead, one can take the curvature of the activation or model space into account.

The well-known GPTQ/OPTQ algorithm by Frantar et al. (2023) uses *input correlations* to inform the quantization algorithm. Concretely, the algorithm aims to find a V that approximately minimizes $\mathbb{E}\|Wa - Va\|_2^2$, where $a \in \mathbb{R}^n$ is an input to the linear unit, and in practice the expectation is taken over a calibration dataset. Hence the Gram matrix $\mathbb{E}[aa^T]$ is used to inform the geometry of the GPTQ algorithm. More concretely, when V is vectorized into $v = \mathrm{vec}(V) \in \mathbb{R}^{mn}$, the Hessian of the above quadratic optimization problem (up to a factor of 2) is exactly $\mathbb{E}[aa^T] \otimes I_m$.

The identity matrix I_m that appears as the second factor is the manifestation of the fact that GPTQ quantizes all *output features* independently and identically. In this paper we consider a natural extension of the GPTQ algorithm to the more general setting of an arbitrary Kronecker-factored Hessian of the form $A \otimes B$, where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{m\times m}$ are arbitrary positive definite matrices. In particular, B is not required to be the identity matrix, but if it is, then the algorithm recovers standard GPTQ.

This natural extension has been used in a functionally equivalent form in previous papers by Kim et al. (2025) (BoA) and Tseng et al. (2026) (YAQA); however, the concrete algorithmic implementation in these previous works is computationally suboptimal. Kim et al. (2025) run a straightforward implementation which requires $\mathcal{O}(mn)$ sequential steps and $\mathcal{O}(m^2n^2)$ total work. Tseng et al. (2026) improve on that with regard to parallelization, requiring only $\mathcal{O}(m+n)$ sequential steps, but still using $\mathcal{O}(m^2n^2)$ total work.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Contributions. This paper concerns the efficient implementation of this quantization method.

1. We describe and accelerate the two-sided GPTQ algorithm to bring the total work down to just $\mathcal{O}(mn(m+n))$, while still requiring only $\mathcal{O}(m+n)$ sequential steps. This is achieved with a recursive divide-and-conquer approach as an additional algorithmic ingredient. The resulting algorithm, ***FastKron***, matches the complexity of standard GPTQ, which also needs cubic $\mathcal{O}(mn^2)$ total cost and linear $\mathcal{O}(n)$ sequential steps, while allowing the quantization algorithm to capture richer geometric information; see Table 1. We also compare benchmarked empirical runtimes and list different Hessian approximation methods.
2. We propose an improved quantization flow for computing the Kronecker-factored Hessian approximation with backpropagation. This improves over the way YAQA computes the Hessians, significantly decreasing memory requirements; see Table 3.

Concurrent work. Chen et al. (2026a) also consider the acceleration of two-sided GPTQ and obtain an algorithm with the same cubic complexity as FastKron. Our work is fully independent, and the cubic-time algorithm presented here predates that preprint.[1] We also note that they use a different algorithmic technique to get the complexity down to $\mathcal{O}(mn(m+n))$.

2 The FastKron Algorithm

2.1 Notation and Conventions

Recall that the goal is to approximate $W \in \mathbb{R}^{m\times n}$ with a low-precision matrix V. We model the quantized matrix V as an integer matrix, $V \in \mathbb{Z}^{m\times n}$. In practice, each row and/or column would be additionally scaled by a factor $s \in \mathbb{R}_{>0}$, the integers might be bounded (clipped), or entirely different alphabets, e.g., special lattices, might be used for V. This, however, does not change the algorithms. For all algorithms described in this paper, one may simply replace "round" by a "$\mathrm{quantize}$" function that maps a scalar to the closest element of the quantization alphabet. For example, Tseng et al. (2026) (YAQA) use Algorithm 3 with a nontrivial base quantizer.

If A is a positive definite matrix, then $\mathrm{LDL}(A)$ denotes the unique lower-triangular matrix L with unit diagonal entries $L_{i,i} = 1$ that satisfies $A = LDL^T$ with some diagonal matrix D. In other words, it is the Cholesky factor of A with columns normalized by their diagonal entries. The positive definite matrices used in this paper are to be interpreted as Gram matrices of some inner product, and are often referred to as *curvature* matrices or Hessians; see Appendix E for more details.

Recall that the Kronecker product satisfies $(A\otimes B)(A'\otimes B') = (AA')\otimes(BB')$. In particular, matrix factorizations like the Cholesky/QR/LDL decompositions distribute over the Kronecker product. For an $m\times n$ matrix W we define $\mathrm{vec}(W)$ to be the column-major flattening of W into a vector of length $m\cdot n$. This defines an isomorphism of vector spaces $\mathrm{vec}\colon \mathbb{R}^{m\times n} \to \mathbb{R}^{mn}$ which satisfies:

$$(A\otimes B)\cdot \mathrm{vec}(W) = \mathrm{vec}(BWA^T) \qquad\qquad \|\mathrm{vec}(W)\|_2 = \|W\|_F \tag{1}$$

Submatrices of an $m\times n$ matrix W are denoted as $W_{i:i',j:j'}$. Indices start at *one*, and ranges *exclude* the start and *include* the end, e.g., $W_{0:m,0:n}$ denotes the full matrix. When A is a square matrix, define $\mathrm{diag}(A)$ to be the diagonal *square matrix of the same size* (not a vector), which has the same diagonal as A and zeros everywhere else. When W is an $m\times n$ matrix, we define its kth anti-diagonal as the collection of all entries $W_{i,j}$ with $i+j=k$. Thus, W has $m+n-1$ anti-diagonals, with indices $k = 2,\dots,m+n$. We further define $\mathrm{antidiag}_k(W)$ to be the $m\times n$ *matrix* which keeps the kth anti-diagonal of W and has all other entries set to 0. Similarly, $\mathrm{antidiag}_{k:k'}(W)$ denotes a matrix that keeps the anti-diagonals $k+1,\dots,k'$ and is zero elsewhere.

2.2 FastKron-naive: GPTQ in the Vectorized Weight Domain

Recall the GPTQ algorithm, described in Algorithm 1. It aims to minimize $D_A = \|(V-W)A^{1/2}\|_F^2$ for some positive definite $A \in \mathbb{R}^{n\times n}$. The matrix A informs the algorithm of the correlation of module inputs $a \in \mathbb{R}^n$. In practice, $A = \mathbb{E}[aa^T]$, where the expectation is over a calibration dataset.

[1] A version of this work has been submitted previously for publication. The cubic-time algorithm presented here was documented in the written record of the review process before the above preprint appeared on arXiv.

Algorithm 1 GPTQ

Input: $W \in \mathbb{R}^{m\times n}$, $L = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$
for $j = 1$ **to** n **do**
  $v \leftarrow \mathrm{round}(W_{:,j})$
  $\Delta \leftarrow v - W_{:,j}$
  $W \leftarrow W + \Delta \cdot (L_{:,j})^T$ ▷ update W (propagate error)
end for
Output: W (which satisfies $W \in \mathbb{Z}^{m\times n}$)

The Hessian of the quadratic loss $\frac{1}{2}D_A$ with respect to $\mathrm{vec}(V)$ is $A \otimes I_m$. The I_m factor is the manifestation of the fact that Algorithm 1 treats all output channels (all rows of W) independently.

Naturally, one can additionally consider a positive definite output curvature matrix $B \in \mathbb{R}^{m\times m}$ and the resulting loss $D_{A,B} = \|B^{1/2}(V-W)A^{1/2}\|_F^2$, whose Hessian with respect to $\mathrm{vec}(V)$ is $A \otimes B$.

As already observed by Kim et al. (2025), there is a natural extension of GPTQ that targets $D_{A,B}$, and it can be described as simply applying Algorithm 1 (standard GPTQ) to the vectorized weight matrix $\hat{W} := \mathrm{vec}(W)^T \in \mathbb{R}^{1\times mn}$ and "input" curvature $\hat{A} := A \otimes B \in \mathbb{R}^{mn\times mn}$. Via Equation (1) we can describe this in unvectorized weight space, giving Algorithm 2. This is the basic extension of GPTQ to a two-sided curvature. The goal of the following subsections is to accelerate Algorithm 2.

Algorithm 2 FastKron-naive (GPTQ in vectorized weight domain)

Input: $W \in \mathbb{R}^{m\times n}$, $L^{(A)} = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, $L^{(B)} = \mathrm{LDL}(B^{-1}) \in \mathbb{R}^{m\times m}$
for $j = 1$ **to** n **do**
  for $i = 1$ **to** m **do**
    $v \leftarrow \mathrm{round}(W_{i,j})$
    $\Delta \leftarrow v - W_{i,j}$
    $W \leftarrow W + L^{(B)}_{:,i} \cdot \Delta \cdot (L^{(A)}_{:,j})^T$ ▷ update W (propagate error)
  end for
end for
Output: W (which satisfies $W \in \mathbb{Z}^{m\times n}$)

2.3 FastKron-antidiagonal: Batch Processing for $m + n$ Sequential Steps

A big issue with Algorithm 2 is that it requires $m \cdot n$ sequential steps. Therefore it can only be employed for small matrices. For example, Kim et al. (2025) use the algorithm in this form, and they overcome the computational burden by only quantizing matrices from the attention block and taking a separate curvature measure for each attention head, so that they can process multiple attention heads in parallel while each attention head only yields small matrices.

To bring the sequential depth down, one has to process some of the $m \cdot n$ steps in parallel. This can be done by analyzing the dependency graph among the $W_{i,j}$, as first shown by Tseng et al. (2026).

The main observation is that the update caused by an entry $W_{i,j}$ only updates entries to the bottom-right of itself, i.e., it only updates entries $W_{i',j'}$ which satisfy $i' \geq i$ and $j' \geq j$. Conversely, this means that $W_{i',j'}$ will only ever be changed by updates caused by entries to the top-left of itself. Thus one can round $W_{i,j}$ and perform the corresponding update to W as soon as all entries above the anti-diagonal $i + j$ have been processed. As shown in Theorem 2.1, this allows one to go from anti-diagonal to anti-diagonal, and process each one in parallel, via:

for $k = 2$ **to** $m + n$ **do** ▷ iterate sequentially over anti-diagonals
  for (i, j) in anti-diagonal k **in parallel do** ▷ process each anti-diagonal in parallel
    $v \leftarrow \mathrm{round}(W_{i,j})$
    $\Delta \leftarrow v - W_{i,j}$
    $W \leftarrow W + L^{(B)}_{:,i} \cdot \Delta \cdot (L^{(A)}_{:,j})^T$
  end for
end for

Now the inner loop adds outer products of vectors to W. Since a sum of outer products can be described by a matrix multiplication, one can batch all updates in the inner loop together using a single matrix multiplication. This yields the FastKron-antidiagonal algorithm, described in Algorithm 3.

Algorithm 3 FastKron-antidiagonal (FastKron-naive with parallel processing of anti-diagonals)

Input: $W \in \mathbb{R}^{m\times n}$, $L^{(A)} = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, $L^{(B)} = \mathrm{LDL}(B^{-1}) \in \mathbb{R}^{m\times m}$
for $k = 2$ **to** $m + n$ **do**
  $D \leftarrow \mathrm{antidiag}_k(W) \in \mathbb{R}^{m\times n}$
  $V \leftarrow \mathrm{round}(D)$
  $\Delta \leftarrow V - D$
  $W \leftarrow W + L^{(B)} \cdot \Delta \cdot (L^{(A)})^T$
end for
Output: W (which satisfies $W \in \mathbb{Z}^{m\times n}$)

The iterative version of the YAQA algorithm described by Tseng et al. (2026, Appendix 6) in the LDLQ framework is essentially equivalent to Algorithm 3, although they use a different base quantizer and the LDLQ description needs a careful ("iterative") implementation. While it requires only $\mathcal{O}(m + n)$ sequential steps, note that it still requires $\mathcal{O}(m^2n^2)$ total work, just like the naive Algorithm 2. This is because each step costs $\mathcal{O}(\min(m, n)^2(m + n))$ due to two matrix multiplications.

2.4 FastKron-recursive: Divide-and-Conquer Approach for $\mathcal{O}(mn(m + n))$ Total Work

We would like to further reduce the total cost of the algorithm. To illustrate the algorithmic idea, in this section we present an algorithm based on FastKron-naive, not using any anti-diagonal batching, which brings down *only* the total algorithmic cost, keeping suboptimal $\mathcal{O}(m \cdot n)$ sequential steps.

The main observation is that the outer product update $W \leftarrow W + L^{(B)}_{:,i} \cdot \Delta \cdot (L^{(A)}_{:,j})^T$ costs $\mathcal{O}(m \cdot n)$, but most of the matrix entries that are updated, namely the values in the bottom-right region of the matrix, will only be accessed much later. One would like to defer these updates, so that later one can perform a batched update of the form $W \leftarrow W + L^{(B)} \cdot \Delta \cdot (L^{(A)})^T$ with Δ being a *dense matrix*.

A similar idea was already used by Frantar et al. (2023) in the original GPTQ paper; however, they used a fixed block size for the lazy batch updates. We will use a block size that depends on the remaining problem size, making the algorithm divide-and-conquer style, and also cache-oblivious. We solve the first half of the problem recursively, propagate the introduced errors to the second half, and then solve the second half of the problem recursively. For GPTQ, this yields Algorithm 4.

But while for GPTQ this modification does not change the total cost, if we apply the same approach to FastKron-naive in a two-dimensional way as shown in Algorithm 5, it *does* bring the total cost down to just $\mathcal{O}(mn(m + n))$, which is essentially the same total cubic cost as that of GPTQ; see Table 1.

Algorithm 4 GPTQ-recursive

Input: $W \in \mathbb{R}^{m\times n}$, $L = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$
if $n = 1$ **then**
  return $\mathrm{round}(W)$ ▷ base case
else
  $s \leftarrow \lfloor n/2 \rfloor$
  $V_{:,0:s} \leftarrow$ GPTQ-recursive$(W_{:,0:s}, L_{0:s,0:s})$ ▷ process first half recursively
  $W_{:,s:n} \leftarrow W_{:,s:n} + (V_{:,0:s} - W_{:,0:s}) \cdot (L_{s:n,0:s})^T$ ▷ propagate errors to second half
  $V_{:,s:n} \leftarrow$ GPTQ-recursive$(W_{:,s:n}, L_{s:n,s:n})$ ▷ process second half recursively
  return V
end if

Algorithm 5 FastKron-recursive (FastKron-naive with recursive divide-and-conquer approach)

Input: $W \in \mathbb{R}^{m\times n}$, $L^{(A)} = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, $L^{(B)} = \mathrm{LDL}(B^{-1}) \in \mathbb{R}^{m\times m}$
if $n = m = 1$ **then**
  return $\mathrm{round}(W)$
else if $m < n$ **then**
  $s \leftarrow \lfloor n/2 \rfloor$
  $V_{:,0:s} \leftarrow$ FastKron-recursive$(W_{:,0:s}, L^{(A)}_{0:s,0:s}, L^{(B)})$ ▷ process first half of columns
  $W_{:,s:n} \leftarrow W_{:,s:n} + L^{(B)} \cdot (V_{:,0:s} - W_{:,0:s}) \cdot (L^{(A)}_{s:n,0:s})^T$ ▷ propagate error
  $V_{:,s:n} \leftarrow$ FastKron-recursive$(W_{:,s:n}, L^{(A)}_{s:n,s:n}, L^{(B)})$ ▷ process second half of columns
  return V
else
  $s \leftarrow \lfloor m/2 \rfloor$
  $V_{0:s,:} \leftarrow$ FastKron-recursive$(W_{0:s,:}, L^{(A)}, L^{(B)}_{0:s,0:s})$ ▷ process first half of rows
  $W_{s:m,:} \leftarrow W_{s:m,:} + L^{(B)}_{s:m,0:s} \cdot (V_{0:s,:} - W_{0:s,:}) \cdot (L^{(A)})^T$ ▷ propagate error
  $V_{s:m,:} \leftarrow$ FastKron-recursive$(W_{s:m,:}, L^{(A)}, L^{(B)}_{s:m,s:m})$ ▷ process second half of rows
  return V
end if

2.5 FastKron: Combining the Two Algorithmic Approaches

Finally, we can combine the two algorithmic techniques from Section 2.3 and Section 2.4, yielding the final *FastKron* algorithm: We process anti-diagonals in parallel, and we do a recursive divide-and-conquer split over the "anti-diagonal dimension". This means we recursively process the first half of anti-diagonals, then we propagate the error to the second half of anti-diagonals, and finally we recursively process the second half of anti-diagonals. See Algorithm 6. This unites the best of both worlds: Just $\mathcal{O}(m+n)$ sequential steps and just $\mathcal{O}(mn(m+n))$ total work; see Table 1.

Algorithm 6 FastKron (combination of FastKron-antidiagonal and FastKron-recursive)

Input: $W \in \mathbb{R}^{m\times n}$, $L^{(A)} = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, $L^{(B)} = \mathrm{LDL}(B^{-1}) \in \mathbb{R}^{m\times m}$, k, l
if $l = k + 1$ **then** ▷ $k : l$ is the range of anti-diagonals to process, initially $k = 1$ and $l = m + n$
  return $\mathrm{round}(\mathrm{antidiag}_l(W))$
else
  $s \leftarrow \lfloor (k + l)/2 \rfloor$
  $V \leftarrow$ FastKron(k, s) ▷ process first half of anti-diagonals
  $W \leftarrow W + \mathrm{antidiag}_{s:l}\left(L^{(B)} \cdot (V - \mathrm{antidiag}_{k:s}(W)) \cdot (L^{(A)})^T\right)$ ▷ propagate error
  $V \leftarrow V +$ FastKron(s, l) ▷ process second half of anti-diagonals
  return V
end if

The following theorem, proved in Appendix A, asserts that all the modifications we have done are valid, i.e., they do not change the output of the algorithm. Appendix B also provides error bounds.

Theorem 2.1. *The algorithms FastKron-naive, FastKron-antidiagonal, FastKron-recursive, and FastKron are all equivalent, i.e., on the same inputs they produce the same outputs.*

2.6 Computational Complexity

Table 1 summarizes the number of sequential steps and total cost of all the described algorithms. These results are proved in Appendix C. We can see that FastKron recovers essentially the same complexity as GPTQ both with respect to parallelizability (linear sequential depth) and total cost (cubic), while handling an arbitrary Kronecker-factored Hessian $A \otimes B$, making the algorithm informed not just by the geometry of input features but also by the geometry of output features.

We note that these costs include neither the Cholesky decomposition, nor the accumulation of the Hessian from the calibration dataset, nor the matrix multiplications needed to send the calibration dataset through the neural network. The last of these turns out to actually be the dominant cost.

Table 1: Asymptotic computational cost of the algorithms for quantizing $W \in \mathbb{R}^{m \times n}$. Lists sequential depth and total cost. Only the core algorithm without Cholesky decomposition is considered.

		Sequential Steps		Total Cost	
Algorithm	Curvature	general	$m \asymp n$	general	$m \asymp n$
GPTQ	one-sided	n	n	mn^2	n^3
FastKron-naive (essentially BoA)	two-sided	mn	n^2	m^2n^2	n^4
FastKron-antidiagonal (essentially YAQA)	two-sided	$m+n$	n	m^2n^2	n^4
FastKron-recursive	two-sided	mn	n^2	$mn(m+n)$	n^3
FastKron	two-sided	$m+n$	n	$mn(m+n)$	n^3

Indeed, suppose we quantize a linear module $W \in \mathbb{R}^{m \times n}$, and we collect the curvature information (Hessian) from a calibration dataset of k sample inputs. For GPTQ, the cost is distributed as follows:

- $\mathcal{O}(kn^2)$ for accumulating the Hessian $A \otimes I_m$ (where $A = \mathbb{E}[aa^T]$)
- $\mathcal{O}(n^3)$ for computing the inverse and the Cholesky decomposition of the Hessian
- $\mathcal{O}(mn^2)$ for the core GPTQ algorithm (n sequential steps with $\mathcal{O}(mn)$ operations each)
- $\mathcal{O}(kmn)$ for (forward) propagating the calibration data through the linear module

Note that, since usually $k \gg m$ and $k \gg n$, the first and last terms dominate. (See Section 4 for realistic sizes of m, n, k.) For FastKron, similarly, the first and last terms dominate:

- $\mathcal{O}(k(n^2+m^2)) = \mathcal{O}(k \max(m,n)^2)$ for accumulating the Hessian $A \otimes B$
- $\mathcal{O}(n^3+m^3) = \mathcal{O}(\max(m,n)^3)$ for computing inverses and Cholesky decompositions
- $\mathcal{O}(mn(m+n))$ for the core FastKron algorithm (split over $\mathcal{O}(m+n)$ sequential steps)
- $\mathcal{O}(kmn)$ for (forward/backward) propagating the calibration data through the module

Thus, in this calibration regime, as with GPTQ, the core FastKron algorithm does not dominate the end-to-end quantization cost. Instead, the principal costs arise from accumulating the Hessian and propagating the calibration data through the module.

Note that this would *not* be the case without our algorithmic improvement: The previous best implementation of two-sided GPTQ was YAQA (i.e., FastKron-antidiagonal), which has quartic total cost instead of cubic, thus becoming the dominant cost when the size of matrices increases.

2.7 Benchmarks

We implemented all algorithms with comparable implementations and benchmarked them on square matrices. See Table 2 and Figure 1 for the results. The implementations are described in Appendix D. Please note that the main concern of the implementations is *comparability*; for example, we use base kernels on equally sized blocks/tiles. Each implementation can be further accelerated with kernel engineering, block size tuning, or even custom memory layout, so we do not claim that these are the fastest implementations, but rather that they are suitable for comparison.

Even if the implementations can still be accelerated by some constant factor, the important takeaway is that FastKron-antidiagonal (i.e. YAQA) becomes impractical for matrices that are used in practice. For example, square matrices with side length $\approx 10^5$ (or equivalent rectangular matrices) can appear in large Transformer models, but are impractical to quantize with FastKron-antidiagonal in large numbers. FastKron resolves this issue by reducing the asymptotic cost from quartic to cubic.

2.8 Role of A, B

FastKron can be used with any positive (semi-)[2]definite matrices $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{m \times m}$. A, B inform the algorithm of the parameter curvature, and they are the most critical part of the whole

[2]If A, B are not positive definite, they can be regularized by adding a small multiple of the identity matrix.

Table 2: Runtime in milliseconds of the relevant described algorithms on square matrices $W \in \mathbb{R}^{n \times n}$, measured on an NVIDIA A40. Only the core algorithm is measured, excluding matrix inversion and Cholesky factorization. FastKron is only a constant factor ($\approx 4\times$) slower than GPTQ-recursive. This is expected and hidden in the $\mathcal{O}$-notation. The important part is that it is *only* a constant factor: FastKron has the same asymptotic growth as GPTQ, whereas FastKron-antidiagonal becomes impractical for matrices of larger size.

n	1536	2048	3072	4096	6144	8192	12288	16384
GPTQ	4	6	17	38	137	293	1089	2285
GPTQ-recursive	5	5	7	12	29	59	183	408
FastKron-antidiagonal (ess. YAQA)	76	165	595	1644	7579	23084	113952	$> 2 \cdot 10^5$
FastKron-recursive	1260	2237	5046	8960	20217	35906	81118	144204
FastKron	12	19	33	58	127	250	731	1546

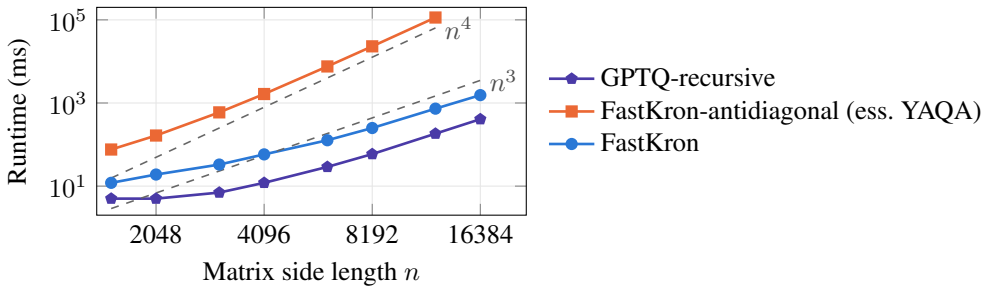

Figure 1: Benchmarked runtimes on logarithmic axes. One can see that FastKron-antidiagonal grows as $\mathcal{O}(n^4)$, while GPTQ-recursive and FastKron only grow as $\mathcal{O}(n^3)$ for larger matrices.

quantization pipeline. On the one hand, their choice determines the quality of the quantized model. On the other hand, their computation dominates the cost of the whole quantization pipeline. For the remainder of the paper, Section 3 outlines various ways to choose A, B, and Section 4 discusses an efficient method to compute the most important ("global") variants for A, B practically.

3 Curvature Approximations

The role of $A \otimes B$ is to give a Kronecker-factored approximation of the curvature of the parameter W. Here, curvature refers to the Hessian of some loss with respect to $\mathrm{vec}(W)$. In other words, FastKron targets $D_{A,B} = \|B^{1/2}(V - W)A^{1/2}\|_F^2$, and we want this to be a second-order approximation of some loss that we care about. It only remains to decide:

1. Which loss to consider for W.
2. How to approximate the resulting Hessian by a Kronecker product.

These two questions are discussed briefly in the next two subsections, and in detail in Appendix E.

For the loss, we list two possibilities: The first is the global negative log-likelihood (or any loss of the network outputs followed by a Gauss–Newton approximation of the Hessian). The second is a local loss function based on the surrounding modules of the linear module W. See Section 3.1.

For the Kronecker factorization, we list three methods from the literature: One is based on an independence assumption ("Marginal"), another is based on an alternating power iteration scheme ("Frobenius"), and a third is an alternative iteration scheme ("FlipFlop"). See Section 3.2.

This gives six possibilities for the Kronecker-factored curvature approximations in FastKron. Kim et al. (2025) have quantized and evaluated models using the local attention module Hessian (BoA). Tseng et al. (2026) evaluated the global Hessian with Frobenius approximation (YAQA). Birnick & Saab (2026) evaluated all listed Hessian approximations (Marginal, Frobenius, FlipFlop) for the

global Hessian (WaterKron). While these papers give the empirical validation for using the two-sided GPTQ algorithm, this paper focuses on the algorithmic efficiency of the algorithm, and therefore intentionally leaves out empirical evaluation beyond benchmarks.

3.1 Choosing the Loss: Local vs. Global

Denote the input to the linear unit W by a and the output by z, so that $z = Wa$. If we write $\mathcal{L}_{x,y}(W) := -\log p_{f_W}(y \mid x)$ for the negative log-likelihood of a sample (x, y) and $b := \nabla_z \mathcal{L}_{x,y}$, then $\nabla_{\mathrm{vec}(W)} \mathcal{L}_{x,y} = a \otimes b$. The Hessian of the expected cross entropy loss of the network with respect to its own predictive distribution is the Fisher information matrix:

$$H_{\mathrm{FIM}} = \mathbb{E}_{x,y\sim f_W(x)}[\nabla^2_{\mathrm{vec}(W)} \mathcal{L}_{x,y}] = \mathbb{E}_{x,y\sim f_W(x)}\left[(a \otimes b)(a \otimes b)^T\right] = \mathbb{E}_{x,y\sim f_W(x)}\left[aa^T \otimes bb^T\right]$$

This is the most natural curvature to consider; however, it comes at the cost of backpropagation through the whole network. If done naively, this can incur high computational and memory requirements. See Section 4 for an improved method of computing with this Hessian.

Alternatively, one can make a local curvature approximation that only requires data from the surrounding modules. This approach is used by the BoA algorithm (Kim et al., 2025). They consider a local loss for the self-attention module of a Transformer and derive the Hessians:

$$H_{W_Q} = \mathbb{E}[aa^T \otimes kk^T] \quad \text{where } k = W_K a \text{ and } a \in \mathbb{R}^{n\times N} \text{ represents a sequence of length } N$$
$$H_{W_K} = \mathbb{E}[aa^T \otimes qq^T] \quad \text{where } q = W_Q a \text{ and } a \in \mathbb{R}^{n\times N} \text{ represents a sequence of length } N$$

We derive similar local Hessians for the MLP module of a Transformer (see Section E.1):

$$H_{W_{\mathrm{up}}} = \mathbb{E}[aa^T \otimes (c_{\mathrm{up}} c_{\mathrm{up}}^T \odot W_{\mathrm{down}}^T W_{\mathrm{down}})] \qquad \text{where } c_{\mathrm{up}} = \sigma(W_{\mathrm{gate}} a)$$
$$H_{W_{\mathrm{gate}}} = \mathbb{E}[aa^T \otimes (c_{\mathrm{gate}} c_{\mathrm{gate}}^T \odot W_{\mathrm{down}}^T W_{\mathrm{down}})] \qquad \text{where } c_{\mathrm{gate}} = \sigma'(W_{\mathrm{gate}} a) \odot W_{\mathrm{up}} a$$

3.2 Choosing a Kronecker Approximation

Once we have a Hessian of the form $H = \mathbb{E}[aa^T \otimes bb^T]$, we still have to find a Kronecker-factored approximation $H \approx A \otimes B$ of it. We list three methods of doing that, with details in Section E.2.

The first ("Marginal") makes an independence assumption to pull the expectation out, yielding:

$$H \approx \mathbb{E}[aa^T] \otimes \mathbb{E}[bb^T] \qquad \text{i.e.} \qquad A := \mathbb{E}[aa^T] \qquad B := \mathbb{E}[bb^T]$$

The second ("Frobenius") targets $\|H - A \otimes B\|_F$ by using an alternating power iteration:

$$A_{t+1} \propto \mathbb{E}[\mathrm{tr}(b^T B_t b) \cdot aa^T] \qquad B_{t+1} \propto \mathbb{E}[\mathrm{tr}(a^T A_{t+1} a) \cdot bb^T].$$

This Kronecker approximation method has been used by Tseng et al. (2026) (YAQA). A third method ("FlipFlop") was recently proposed by Birnick & Saab (2026). It targets $\Phi(H, A \otimes B) := \frac{1}{mn} \mathrm{tr}(H(A \otimes B)^{-1})/(|H|^{1/mn} \cdot |A \otimes B|^{-1/mn})$, which is an objective specifically to minimize quantization distortion measured in the full curvature H, and it uses a similar alternating iteration:

$$A_{t+1} \propto \mathbb{E}[\mathrm{tr}(b^T B_t^{-1} b) \cdot aa^T] \qquad B_{t+1} \propto \mathbb{E}[\mathrm{tr}(a^T A_{t+1}^{-1} a) \cdot bb^T].$$

4 Computing Backpropagated Hessians Efficiently

In this section, we explain an improved quantization flow to compute Kronecker approximations of the "global" Hessian H_{FIM} from Section 3.1 efficiently. Note that for any of the approximation methods from Section 3.2, all we need is to be able to sample the module input $a \in \mathbb{R}^n$ and the module output gradient $b \in \mathbb{R}^m$ according to the particular distribution discussed in Section 3.1.

In practice, the distribution of network inputs x is taken from a calibration dataset. Accordingly, a is computed by forward-propagating x to the module W, and b is computed by forward-propagating a further till the end of the network, then sampling a label y, computing the negative log-likelihood, and back-propagating the gradient to the module W.

However, the concrete schedule of doing this, the *quantization flow*, determines the compute and memory requirements of this endeavor. We first review the classic forward flow and two naive flows

for our problem. Then we propose our new flow. We discuss compute and memory requirements for each flow under a simplified network with ℓ layers, each of which contains one $n \times n$ linear module, and with a calibration dataset of k inputs. We only consider the cost of forward/backward-propagating activations/gradients and executing the Hessian approximation method, since this dominates the cost of running the actual FastKron rounding algorithm when $k \gg n$, as discussed in Section 2.6. Concretely, we assume $k \geq n\ell$. Practical numbers are usually $k \geq 256n$, $\ell \leq 256$, and $2^9 \leq n \leq 2^{14}$; in particular, this simplifying assumption is satisfied. "Fast memory" in practice is VRAM of the GPU. "Slow memory" can be RAM of the host machine, or might even be (partially) offloaded to much slower storage. All costs are summarized in Table 3.

Forward Flow. We first recall the classic GPTQ flow, which also works for the "local" Hessian variants of FastKron. We proceed layer by layer, and only load one layer at a time into memory. The computational cost is $\mathcal{O}(kn^2)$ in each layer to forward the activations, so $\mathcal{O}(kn^2\ell)$ in total. For each layer, we need to keep W, A, B in fast memory, costing $\mathcal{O}(n^2)$ fast memory. We also need to store the activations a for all k calibration inputs, costing $\mathcal{O}(kn)$, but this can reside in slow memory.

Fresh Backward Passes Flow. Recall from above that for the "global" Hessian, we additionally need to send activations through the whole network and back-propagate the gradient. The naive way of doing this would be to keep proceeding layer by layer, and do a fresh backward pass for all k calibration samples each time. The computational cost of this, however, is quadratic in the number of layers, namely $\mathcal{O}(kn^2\ell^2)$. The fast and slow memory requirements are $\mathcal{O}(n^2)$ and $\mathcal{O}(kn)$, respectively, as in the forward flow (provided batch sizes for backward passes are kept small enough).

Simultaneous Hessian Flow. To bypass that computational burden, one can just propagate each of the k calibration samples only *once* forward and backward through the network, and compute/accumulate the Hessian approximations for all modules at the same time. This is the flow used by Tseng et al. (2026) for YAQA. However, while it brings the computational cost back to $\mathcal{O}(kn^2\ell)$, it introduces massive requirements for fast memory: We need to have all $\mathcal{O}(\ell)$ Hessian Kronecker factors readily available, necessitating $\mathcal{O}(n^2\ell)$ fast memory. This is at least the size of the whole model, but $\mathcal{O}$-notation and model simplification hide large constants: in practice, it usually requires at least 5 times the model size of fast memory.

Recursive Halving Flow. This leads us to propose a the recursive divide-and-conquer flow which aims to find a sweet spot: low fast memory requirements and reasonable compute requirements. It does so by employing recursive checkpointing. Given a range of layers, initially the full list of layers, the recursive algorithm requires the input to the first layer and the gradient of the output of the last layer. Next, a forward and backward pass (for all k calibration samples) is required to compute the activations as well as their gradient at exactly the middle of the layer list ("mid-layer activations"). Then one can split the layer list into the first half and second half. For each of those lists one has the input as well as the gradient of the output, so one can call the algorithm recursively on the two layer lists. When the list consists of just a single layer, one has reached the base case, where one can use one of the Kronecker approximation methods and call FastKron. This divide-and-conquer approach requires $\mathcal{O}(kn^2\ell \log \ell)$ compute, $\mathcal{O}(n^2)$ fast memory, and $\mathcal{O}(kn \log \ell)$ slow memory. Thus, compared to the forward flow, it only adds a factor of $\log \ell$ to compute and slow memory, while providing each module with additional global back-propagated curvature information.

Without this flow improvement it can be impractical to quantize larger models in environments with limited compute and fast memory capabilities. This (affordability) is also the reason why we do not provide comparative empirical benchmarks for the flows. For example, on a single GPU, the "Fresh Backward Passes" and "Simultaneous Hessian" flows become impractical for models beyond a few billion parameters.

Table 3: Asymptotic compute and memory costs of the different quantization flows under k calibration samples, ℓ layers, modules of size $n \times n$, and the realistic simplifying assumption $k \geq n\ell$.

Flow	Hessian	Compute	Fast mem.	Slow mem.
Forward	local	$kn^2\ell$	n^2	kn
Fresh Backward Passes	global	$kn^2\ell^2$	n^2	kn
Simultaneous Hessian (YAQA)	global	$kn^2\ell$	$n^2\ell$	–
Recursive Halving	global	$kn^2\ell \log \ell$	n^2	$kn \log \ell$

AI USE STATEMENT

All new algorithms in this paper were found by humans.

We have used generative AI tools to implement the methods into software code, in conversation for literature search and to help develop theoretical models or conceptual frameworks, and for polishing tasks for the paper, such as formatting tables and finding typos.

We have not used AI to formulate mathematical claims, provide critical ingredients for proving mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, support qualitative and thematic data analysis, assist in the writing of proofs, or interpret results. We have also not used AI to perform tasks outside the scope of this paper, such as to generate synthetic datasets, assist with translation, or clean and reformat datasets.

We have reviewed all AI-assisted work and take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

The source code to reproduce the benchmarks will be made public with the publication of the paper.

REFERENCES

Johann Birnick. The lattice geometry of neural network quantization: A short equivalence proof of GPTQ and Babai's algorithm. In *International Conference on Learning Representations*, volume 2026, pp. 101779–101787, 2026.

Johann Birnick and Rayan Saab. WaterKron and FlipFlop Hessian: Information-theoretically grounded quantization with Kronecker-factored Hessians. *arXiv preprint arXiv:2609.14706*, 2026.

Jiale Chen, Torsten Hoefler, and Dan Alistarh. GPTQ-2D: Cubic-time two-sided adaptive rounding. *arXiv preprint arXiv:2607.27042*, 2026a.

Jiale Chen, Yalda Shabanzadeh, Elvir Crnčević, Torsten Hoefler, and Dan Alistarh. The geometry of LLM quantization: GPTQ as Babai's nearest plane algorithm. In *International Conference on Learning Representations*, volume 2026, pp. 122653–122695, 2026b.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

Junhan Kim, Ho-young Kim, Eulrang Cho, Chungman Lee, Joonyoung Kim, and Yongkweon Jeon. BoA: Attention-aware post-training quantization without backpropagation. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Uvj6XcSJ5d.

James Martens. New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21(146):1–76, 2020.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Albert Tseng, Zhaofeng Sun, and Christopher De Sa. Model-preserving adaptive rounding. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=PKFilPWjMI.

Charles F. Van Loan and Nikos Pitsianis. Approximation with Kronecker products. In *Linear Algebra for Large Scale and Real-Time Applications*, pp. 293–314. Springer, 1993.

Haoyu Zhang, Shihao Zhang, Ian Colbert, and Rayan Saab. Provable post-training quantization: Theoretical analysis of OPTQ and Qronos. *arXiv preprint arXiv:2508.04853*, 2025.

A Equivalence Proof

Proof of Theorem 2.1. Recall that $L^{(A)} = \mathrm{LDL}(A^{-1})$ and $L^{(B)} = \mathrm{LDL}(B^{-1})$ are lower triangular and normalized to satisfy $L^{(A)}_{i,i} = L^{(B)}_{j,j} = 1$. Define the following partial order:

$$(i,j) \preceq (p,q) \quad :\Longleftrightarrow \quad i \le p \text{ and } j \le q$$

A rounding error $\Delta_{i,j}$ adds $L^{(B)}_{p,i} \cdot \Delta_{i,j} \cdot L^{(A)}_{q,j}$ to entry $W_{p,q}$. This vanishes unless $(i,j) \preceq (p,q)$. Hence an entry can be rounded as soon as it has received all updates from the entries with smaller indices than itself with respect to the order $\preceq$.

Therefore, one may reorder the rounding steps, provided that the visited sequence of entries does not decrease with respect to $\preceq$. One may also delay the updates caused by rounding, provided that before rounding an entry, the contributions from all smaller indices with respect to $\preceq$ have been accumulated. Lastly, because of the simple algebraic fact that a sum of outer products corresponds to a matrix multiplication, one may batch updates using matrix multiplication. Concretely, for any matrix Δ of rounding errors we have:

$$L^{(B)} \cdot \Delta \cdot (L^{(A)})^T = \sum_{i,j} L^{(B)}_{:,i} \cdot \Delta_{i,j} \cdot (L^{(A)}_{:,j})^T$$

FastKron-naive visits entries in column-major order, which respects $\preceq$, and does not delay or batch any updates. FastKron-antidiagonal batches updates from each anti-diagonal, but does not delay them, and the anti-diagonal schedule respects $\preceq$ because if the anti-diagonal indices of entries (i,j) and (p,q) compare as $i + j \ge p + q$, then the only way for $(i,j) \preceq (p,q)$ to hold is $(i,j) = (p,q)$. The divide-and-conquer algorithms FastKron-recursive and FastKron split the indices into two sets S_1 and S_2, and it holds $s_1 \preceq s_2$ for all $s_1 \in S_1, s_2 \in S_2$, and all the updates to entries in S_2 caused by entries in S_1 are accumulated before any entry of S_2 is rounded. □

B Error Bound Guarantees

We recall the error bound from GPTQ as proved recently in the works by Zhang et al. (2025); Birnick (2026); Chen et al. (2026b). For the statements, we use the reverse LDL decomposition: For a positive definite matrix A, when we write $A = UDU^T$, then U is the unique upper-triangular matrix with unit diagonal $U_{i,i} = 1$ that satisfies this equation with a diagonal matrix D.

Theorem B.1. *Suppose Algorithm 1 outputs $V \in \mathbb{Z}^{m\times n}$ on inputs $W \in \mathbb{R}^{m\times n}$ and $L = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, and write $A = UDU^T$. Then*

$$D_A = \|(V - W)A^{1/2}\|_F^2 \le \frac{m}{4}\sum_{j=1}^{n} D_{j,j} \le \frac{m}{4}\sum_{j=1}^{n} A_{j,j} = \frac{m}{4}\,\mathrm{tr}(A).$$

From this we can deduce an error bound for FastKron.

Corollary B.2. *Suppose Algorithm 6 outputs $V \in \mathbb{Z}^{m\times n}$ on inputs $W \in \mathbb{R}^{m\times n}$, $L^{(A)} = \mathrm{LDL}(A^{-1}) \in \mathbb{R}^{n\times n}$, and $L^{(B)} = \mathrm{LDL}(B^{-1}) \in \mathbb{R}^{m\times m}$, and write $A = U^{(A)}D^{(A)}(U^{(A)})^T$ and $B = U^{(B)}D^{(B)}(U^{(B)})^T$. Then*

$$D_{A,B} = \|B^{1/2}(V - W)A^{1/2}\|_F^2 \le \frac{1}{4}\sum_{i=1}^{m}\sum_{j=1}^{n} D^{(B)}_{i,i} D^{(A)}_{j,j} \le \frac{1}{4}\,\mathrm{tr}(A)\cdot\mathrm{tr}(B).$$

Proof. By Theorem 2.1, it suffices to prove this for the output of Algorithm 2. But the output of Algorithm 2 is just the output of Algorithm 1 when called with inputs $\mathrm{vec}(W)^T \in \mathbb{R}^{1\times mn}$ and $\mathrm{LDL}((A \otimes B)^{-1}) = L^{(A)} \otimes L^{(B)}$. Now Theorem B.1 with a single row, using $(A \otimes B)^{1/2} = A^{1/2} \otimes B^{1/2}$ and $A \otimes B = (U^{(A)} \otimes U^{(B)})(D^{(A)} \otimes D^{(B)})(U^{(A)} \otimes U^{(B)})^T$, guarantees that

$$\begin{aligned}\|(A^{1/2} \otimes B^{1/2}) \cdot \mathrm{vec}(V - W)\|_2^2 &\le \frac{1}{4}\sum_{k=1}^{m\cdot n}(D^{(A)} \otimes D^{(B)})_{k,k} = \frac{1}{4}\sum_{i=1}^{m}\sum_{j=1}^{n} D^{(B)}_{i,i} D^{(A)}_{j,j} \\ &\le \frac{1}{4}\sum_{k=1}^{m\cdot n}(A \otimes B)_{k,k} = \frac{1}{4}\,\mathrm{tr}(A)\cdot\mathrm{tr}(B).\end{aligned}$$

We finish by applying Equation (1). □

These guarantees are stated for simplicity in our convention that $V \in \mathbb{Z}^{m\times n}$, but they also apply (with appropriate scaling modification) when rows and columns of V are additionally scaled by real-valued scaling factors. There are also expected error guarantees for the case when the entries $W_{i,j}$ are iid Gaussian, and also measured directly with respect to the full Hessian H (instead of the approximation $A \otimes B$); see Birnick & Saab (2026).

C Complexity Proofs

In this section, we prove the bounds on sequential steps and total cost reported in Table 1.

We consider only the core algorithms, excluding the matrix inversion or Cholesky factorization. Total cost counts scalar arithmetic operations. Sequential steps count parallel scalar operations as a single step, and also count a full matrix multiplication as a single step (which is a simplified complexity model since technically a matrix multiplication needs log-many sequential steps).

Sequential Steps. GPTQ, FastKron-naive, and FastKron-antidiagonal have n, mn, and $m+n-1$ iterations, respectively, where each iteration only requires a constant number of sequential steps. The divide-and-conquer algorithms FastKron-recursive and FastKron process their two children one after another, and otherwise only use a constant number of sequential steps, thus we need to count the number of nodes in the binary recursion tree, or equivalently the number of leaves. FastKron-recursive has one leaf for each of the mn matrix entries, while FastKron has one leaf for each of the $m+n-1$ anti-diagonals.

GPTQ Total Cost. The dominant cost in each of the n iterations is to update $\mathcal{O}(mn)$ entries of W, giving $\mathcal{O}(mn^2)$ total cost.

FastKron-naive Total Cost. Similarly, each iteration costs $\mathcal{O}(mn)$ for the outer product update of W, but there are mn iterations, giving $\mathcal{O}(m^2n^2)$.

FastKron-antidiagonal Total Cost. Let d_k be the number of entries of the kth anti-diagonal. Its update is a sum of d_k outer products, which can be evaluated as the product of an $m \times d_k$ matrix with a $d_k \times n$ matrix, costing $\mathcal{O}(mnd_k)$. Since every entry is on exactly one anti-diagonal we have $\sum_{k=2}^{m+n} d_k = mn$, hence the total work is:

$$\sum_{k=2}^{m+n} \mathcal{O}(mnd_k) = \mathcal{O}\left(mn \sum_{k=2}^{m+n} d_k\right) = \mathcal{O}(m^2n^2)$$

FastKron-recursive Total Cost. In each node of the recursion tree, operating on a matrix of size $r\times s$, the cost of error propagation is $\mathcal{O}(r^2s+rs^2) = \mathcal{O}(rs(r+s))$ due to two matrix multiplications. (Left-multiplication with $L^{(B)}$ and right-multiplication with $(L^{(A)})^T$.) Distribute this cost equally among the rs entries of the $r \times s$ block, so that each entry pays $\mathcal{O}(r+s)$ cost in this recursion node. Now compute the cost that each entry of the whole $m \times n$ matrix pays over all the recursion blocks that it is contained in: Each entry appears once in the root node, paying $\mathcal{O}(m+n)$. Then it recursively appears in one of the child blocks, whose number of entries is halved, so that in sum over all those nodes the entry pays $\mathcal{O}(m+n) + \mathcal{O}((m+n)/2) + \mathcal{O}((m+n)/4) + \ldots + \mathcal{O}(1)$, which is $\mathcal{O}(m+n)$. Summing this cost over all mn entries, we have a total cost of $\mathcal{O}(mn(m+n))$.

FastKron Total Cost. Similarly, consider a recursion node which operates on the anti-diagonal range $k : l$, and denote the set of entries that lie on these anti-diagonals by S. We claim that the cost of error propagation in this node is $\mathcal{O}(|S|\cdot(l-k))$, which when distributed equally among all the $|S|$ entries of this anti-diagonal range gives a cost of $\mathcal{O}(l-k)$ per entry. Now, again, compute the cost that each entry of the whole $m \times n$ matrix pays over all the recursion nodes that it is contained in: Each entry pays $\mathcal{O}(m+n)$ in the root node with range $1 : m+n$, and then recursively the number of anti-diagonals is halved for each child, giving also $\mathcal{O}(m+n)$ cost for the entry over all recursion nodes. Summing over all mn entries gives a total cost of $\mathcal{O}(mn(m+n))$.
It only remains to prove our claim that error propagation in the $k : l$ node costs $\mathcal{O}(|S| \cdot (l-k))$. We have to compute the matrix product $L^{(B)}\Delta(L^{(A)})^T$, but only on the anti-diagonal range $s : l$, and Δ is supported only on the anti-diagonal range $k : s$, where $s = \lfloor (k+l)/2 \rfloor$ is the range midpoint. For

this, it suffices to compute $E = L^{(B)}\Delta$ only on the entries in S, yielding E', and then to compute $E'(L^{(A)})^T$ only on the entries of anti-diagonal range $s : l$. Indeed, suppose we want to compute the (i, j) entry of $E(L^{(A)})^T$, where $i + j \leq l$. This is $\sum_p E_{i,p} L^{(A)}_{j,p}$. Since $L^{(A)}_{j,p} = 0$ for $j < p$, we only need $E_{i,p}$ for $p \leq j$. Since then $i + p \leq i + j \leq l$, we only need entries of E on the anti-diagonals up to l. Moreover, $E_{i,p} = \sum_{i'} L^{(B)}_{i,i'} \Delta_{i',p}$ vanishes for $i + p \leq k$ since $L^{(B)}_{i,i'} = 0$ for $i < i'$ and $\Delta_{i',p} = 0$ for $i' + p \leq k$. Hence all needed entries of E lie in S. Both the computation of E' and of the relevant entries of $E'(L^{(A)})^T$ require at most $|S|$ inner products of cost $\mathcal{O}(l - k)$ each, landing at $\mathcal{O}(|S| \cdot (l - k))$ cost for the error propagation.

D Implementation

This section describes the implementations whose runtimes were reported in Section 2.7. The goal of the implementations is not to be the fastest possible, but to be reasonably fast while being comparable for the benchmark results.

All algorithms are implemented using Triton (Tillet et al., 2019). They use comparable base kernels on 16×16 blocks, comparable error propagation, and the same basic row-major memory layout. Everything runs in full IEEE float32 precision, without use of TF32 tensor cores.

Base Kernels. Each algorithm processes a 16×16 tile of W with a base kernel, so that on the order of 16 sequential steps are processed within a single kernel. The reason for this "tiled" schedule choice is that the cost would be dominated by kernel launch overhead if a separate kernel were launched for each sequential step. The algorithm is executed for each tile in registers, propagating errors within the tile without delaying any updates. For example, the GPTQ base kernel rounds a tile column by column (16 in total), and FastKron-antidiagonal and FastKron round the 31 anti-diagonals of the tile sequentially. For GPTQ, all tiles in a band of 16 columns are run in parallel, while for FastKron-antidiagonal and FastKron, all tiles lying on an "anti-diagonal of tiles" are run in parallel. For FastKron-recursive, the base kernel rounds single entries of the tile at a time, in order of the recursion, and runs no tiles in parallel.

Error Propagation. Between the (potentially parallel) runs of the base kernels, the errors are propagated to other tiles, following the algorithmic descriptions of the algorithms. For example, GPTQ propagates to all later tiles, while GPTQ-recursive has a recursive error propagation schedule which only propagates errors to the "next few" tiles, where "few" can also be as much as half of the matrix in the root of the schedule. The update matrix $L^{(B)}\Delta(L^{(A)})^T$ is computed with two masked matrix multiplications, as in the proof of Appendix C: first $E = L^{(B)}\Delta$ is computed on the relevant entries, yielding E', and then $E'(L^{(A)})^T$ is computed on the relevant entries (the ones that are ultimately updated). For GPTQ we have $B = I_m$, so only the second product is computed. The matrix multiplications use 128×128 output blocks if the updated region spans at least 128 rows, columns or anti-diagonals, and 16×16 blocks otherwise. (The reason for using larger blocks if possible is to match the complexity model from Appendix C where one matrix multiplication counts as a single sequential step.)

E Curvature Approximations in Detail

Let us first explain the way we use the term *curvature* matrix or *Hessian*. We use the term curvature in an optimization sense, as in Martens (2020), which should not be confused with the curvature tensor from Riemannian geometry; instead, what we call curvature is called the *metric* in Riemannian geometry. Concretely, for us, a curvature matrix is any positive definite matrix which gives a local quadratic approximation of some loss function, which is why we also refer to curvature matrices purely as Hessians (implicitly assuming that they are positive definite). Thus a curvature matrix captures the local geometry of the loss function; it can also be interpreted as the Gram matrix of an inner product. The loss function, for the purpose of this paper, measures the distortion between the original and the quantized model.

$H \in \mathbb{R}^{mn \times mn}$ denotes the induced curvature / Gram matrix / Hessian for the (vectorized) weight space $\mathrm{vec}(W) \in \mathbb{R}^{mn}$. Since it is computationally infeasible to deal with this directly for the purpose of this paper, we seek to approximate it with a Kronecker product $H \approx A \otimes B$ with

$A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{m\times m}$. Then, in a certain sense, A captures the geometry of the input activations, while B captures the geometry of the output gradients. More concretely, A defines an inner product on row vectors $\delta w \in \mathbb{R}^{1\times n}$, i.e., on input activation *dual* space, while B defines an inner product on column vectors $\delta z \in \mathbb{R}^{m\times 1}$, i.e., on output activation space. The former measures the change in loss that δw causes, while the latter measures the change in loss that δz causes.

We now describe the Hessian constructions from Section 3 in detail. Recall that we first have to choose a loss, which determines H, and then we can choose a Kronecker approximation method to find A, B.

E.1 Choosing the Loss: Local vs. Global

The most desirable way to measure the distortion is to use the loss function $\mathcal{L}(W)$ which was also used to train the network. Since the network is given to us in trained state, it is natural to assume the network parameters are at (or near) a local minimizer of $\mathcal{L}$, so that $\nabla_{\mathrm{vec}(W)}\mathcal{L} \approx 0$ and $\nabla^2_{\mathrm{vec}(W)}\mathcal{L}$ is positive (semi-)definite. Keeping only the quadratic term in the Taylor expansion of $\mathcal{L}$ around W then yields a quadratic loss for $\mathrm{vec}(V) - \mathrm{vec}(W)$ with Hessian $H = \nabla^2_{\mathrm{vec}(W)}\mathcal{L}$.

Now suppose the output $f_W(x)$ of the neural network represents a probability distribution for the label y. Since the quantized model should approximate the original model, a natural choice is the cross entropy loss with respect to the model's own predictive distribution. That is, the expectation below is taken over the calibration input data and over labels sampled from the predictive distribution of the original model:

$$\mathcal{L}(V) = \mathbb{E}_{x,y\sim f_W(x)}[-\log p_{f_V}(y \mid x)]$$

In this case, it is well known (Martens, 2020) that the Hessian at $V = W$ is equal to an expectation over gradient outer products, namely it is equal to the Fisher information matrix (FIM):

$$\nabla^2_{\mathrm{vec}(W)}\mathcal{L} = \mathbb{E}_{x,y\sim f_W(x)}[\nabla_{\mathrm{vec}(W)} \log p_{f_W}(y \mid x) \nabla_{\mathrm{vec}(W)} \log p_{f_W}(y \mid x)^T]$$

Since W is a matrix, this expression has additional structure. Denote the input to the linear unit W by a and the output by z, so that $z = Wa$. If we write $\mathcal{L}_{x,y}(W) := -\log p_{f_W}(y \mid x)$ and $b := \nabla_z \mathcal{L}_{x,y}$, then $\nabla_{\mathrm{vec}(W)}\mathcal{L}_{x,y} = a \otimes b$. Thus the Fisher information matrix associated with W is:

$$H_{\mathrm{FIM}} = \mathbb{E}_{x,y\sim f_W(x)}\left[(a \otimes b)(a \otimes b)^T\right] = \mathbb{E}_{x,y\sim f_W(x)}\left[aa^T \otimes bb^T\right]$$

One could instead take the label y from the calibration dataset, rather than sampling it from the model's predictive distribution. In this case, the corresponding matrix is known as the *empirical* Fisher information matrix. For FastKron, this choice would make the quantization more strongly adapted ("overfitted") to the calibration dataset, which may or may not be desired. More generally, the Gauss-Newton component of the Hessian for *any* (convex) network loss has the same form as H_{FIM} above.

Another option is to make a local curvature approximation, which only requires data from surrounding modules; in particular, it does not require a full backward pass through the network. This approach is used by the BoA algorithm (Kim et al., 2025). BoA considers the self-attention module in Transformer-based architectures, where the attention matrix is computed as $a_{:,i}^T W_Q^T W_K a_{:,j}$. Here, only for this paragraph, a denotes not just a single token vector but a matrix that holds a full sequence of tokens. If one uses a Frobenius loss for the attention matrix alone, then the resulting local Hessians are

$$H_{W_Q} = \mathbb{E}[aa^T \otimes kk^T] \qquad \text{where } k = W_K a$$
$$H_{W_K} = \mathbb{E}[aa^T \otimes qq^T] \qquad \text{where } q = W_Q a$$

where the expectation is over sequences. The BoA algorithm forms these local Hessians separately for each attention head, which allows the attention heads to be processed in parallel. This parallelism is key to making their method practical. Indeed, BoA applies GPTQ-style updates in the vectorized weight domain, but does not use any of the algorithmic performance improvements that we discussed in this paper, i.e., they use FastKron-naive. FastKron therefore provides a direct acceleration of BoA.

The approach of Kim et al. (2025) does not consider the MLP module in the Transformer architecture. Matrices in this module do not naturally split into a collection of smaller matrices like the

attention heads in the attention module. Thus, applying FastKron-naive would require $m \cdot n$ sequential steps in the MLP module, making the approach impractical for large weight matrices. FastKron makes this computation feasible, so we derive local Hessians for the MLP module. Concretely, we focus on the case of a Gated Linear Unit (GLU) followed by another linear unit, as employed by many modern Transformer-based architectures. For simplicity of exposition we ignore biases. Fix an activation function σ. Then the MLP is given by:

$$\mathrm{MLP}(a) = W_{\mathrm{down}}(W_{\mathrm{up}}a \odot \sigma(W_{\mathrm{gate}}a))$$

If we use D_s to denote a diagonal matrix with the vector s on the diagonal, this can be written as:

$$\mathrm{MLP}(a) = W_{\mathrm{down}}D_{\sigma(W_{\mathrm{gate}}a)}W_{\mathrm{up}}a$$

So if we use an ℓ_2-loss for the output of the MLP module alone, the corresponding local Hessians are:

$$H_{W_{\mathrm{up}}} = \mathbb{E}[aa^T \otimes (c_{\mathrm{up}}c_{\mathrm{up}}^T \odot W_{\mathrm{down}}^T W_{\mathrm{down}})] \qquad \text{where } c_{\mathrm{up}} = \sigma(W_{\mathrm{gate}}a)$$
$$H_{W_{\mathrm{gate}}} = \mathbb{E}[aa^T \otimes (c_{\mathrm{gate}}c_{\mathrm{gate}}^T \odot W_{\mathrm{down}}^T W_{\mathrm{down}})] \qquad \text{where } c_{\mathrm{gate}} = \sigma'(W_{\mathrm{gate}}a) \odot W_{\mathrm{up}}a$$

Note that the Schur product theorem guarantees that the Hadamard product of two positive (semi-)definite matrices is itself positive (semi-)definite.

E.2 Choosing a Kronecker Approximation

In the previous subsection, we considered both a global and a local approach for computing Hessians for the linear unit $W \in \mathbb{R}^{m\times n}$. In both cases the resulting Hessian has the form

$$H = \mathbb{E}[aa^T \otimes bb^T]$$

where a, b may be vectors or matrices, and the expectation is taken over the calibration data.

In both the global and local settings, a is the input to the linear unit. In the global setting, b is the gradient of the global loss with respect to the output of the linear unit, and in the local MLP setting $b = D_{c_{\mathrm{up}}}W_{\mathrm{down}}^T$ or $b = D_{c_{\mathrm{gate}}}W_{\mathrm{down}}^T$.

However, this Hessian is an *expectation* over Kronecker-factored matrices, and therefore is itself *not* a Kronecker-factored matrix, in general. We therefore seek a Kronecker product approximation of the form $H \approx A \otimes B$.

One simple approach to obtain such an approximation is to assume that a, b are independent, so that the Kronecker product can be pulled out of the expectation. This gives

$$H \approx \mathbb{E}[aa^T] \otimes \mathbb{E}[bb^T].$$

We refer to this as the *Marginal* approximation.

Another approach is based on a power iteration scheme that aims to minimize $\|H - A \otimes B\|_F$. We refer to this as the *Frobenius* approximation. This technique has been employed by Tseng et al. (2026) for YAQA, and we explain it in detail below.

This problem of finding the best Kronecker-factored approximation of a matrix in Frobenius norm was studied by Van Loan & Pitsianis (1993). They define a rearrangement operator $\mathcal{R}$ which maps an $mp \times nq$ matrix to an $mn \times pq$ matrix. This operator $\mathcal{R}$ is linear, invertible, and satisfies:

$$\|\mathcal{R}(M)\|_F = \|M\|_F \qquad \langle \mathcal{R}(M), \mathcal{R}(N)\rangle = \langle M, N\rangle \qquad \mathcal{R}(M \otimes N) = \mathrm{vec}(M)\,\mathrm{vec}(N)^T$$

Since $\mathcal{R}$ preserves the Frobenius norm, minimizing $\|H - A \otimes B\|_F$ is equivalent to minimizing $\|\mathcal{R}(H) - \mathrm{vec}(A)\,\mathrm{vec}(B)^T\|_F$. Thus, the best Kronecker approximation of H is obtained from the best rank-1 approximation of $\mathcal{R}(H)$. In our setting,

$$\mathcal{R}(H) = \mathbb{E}[\mathcal{R}(aa^T \otimes bb^T)] = \mathbb{E}[\mathrm{vec}(aa^T)\,\mathrm{vec}(bb^T)^T]$$

The best rank-1 approximation to a matrix is given by the outer product of the top left and top right singular vectors, scaled by the top singular value. Since the algorithms in this paper are invariant under multiplying the Hessian by a scalar, we only need the top left and top right singular vectors of

$\mathcal{R}(H)$ up to scaling. These singular vectors can be found using the power iteration. Given a matrix M, and initial vectors u_0 and v_0, the iterations (without normalization) take the form:

$$u_{t+1} = Mv_t \qquad v_{t+1} = M^T u_{t+1}$$

Substituting $M = \mathcal{R}(H)$ and identifying $u = \mathrm{vec}(A)$ and $v = \mathrm{vec}(B)$ gives:

$$\mathrm{vec}(A_{t+1}) = \mathbb{E}[\mathrm{vec}(aa^T)\,\mathrm{vec}(bb^T)^T\,\mathrm{vec}(B_t)] = \mathbb{E}[\mathrm{vec}(aa^T)\,\mathrm{tr}(bb^T B_t)]$$

Applying vec^{-1} and using the cyclic property of the trace yields:

$$A_{t+1} = \mathbb{E}[\mathrm{tr}(b^T B_t b) \cdot aa^T] \qquad B_{t+1} = \mathbb{E}[\mathrm{tr}(a^T A_{t+1} a) \cdot bb^T]$$

Each iteration requires a separate pass over the calibration data. Compared to the Marginal approximation above, this approach weights the outer products aa^T by $\mathrm{tr}(b^T B_t b)$. If a and b are both vectors, as in our global Hessian case, the arguments of the trace operators above become scalars, and one can drop the "tr" from the expressions. For the local MLP Hessians, we have $b = D_{c_{\mathrm{up}}} W_{\mathrm{down}}^T$ or $b = D_{c_{\mathrm{gate}}} W_{\mathrm{down}}^T$, and we find (in the former case):

$$\mathrm{tr}(b^T B_t b) = \mathrm{tr}(W_{\mathrm{down}} D_{c_{\mathrm{up}}} B_t D_{c_{\mathrm{up}}} W_{\mathrm{down}}^T) = c_{\mathrm{up}}^T (B_t \odot W_{\mathrm{down}}^T W_{\mathrm{down}}) c_{\mathrm{up}}$$

A third approximation method called *FlipFlop* was introduced by Birnick & Saab (2026). Algorithmically, it utilizes the same alternating iterations as the Frobenius approximation method, but it is "reciprocal" in the sense that, inside the expectation, B_t is replaced by B_t^{-1}, and A_{t+1} is replaced by A_{t+1}^{-1}. While the Frobenius iteration minimizes $\|H - A \otimes B\|_F$, the FlipFlop iteration minimizes $\Phi(H, A \otimes B)$, which they call the Kronecker-Hessian mismatch factor:

$$\Phi(H, A \otimes B) := \frac{\mathrm{tr}(H(A \otimes B)^{-1})/mn}{|H|^{1/mn} \cdot |A \otimes B|^{-1/mn}}$$

It is specifically derived for the purpose of minimizing distortion of the two-sided GPTQ algorithm: Birnick & Saab (2026) show theoretically and empirically that FlipFlop outperforms the other Kronecker approximation methods in this context.