A THEORY

ROAST is a generalized model compression which performs operation specific system-friendly lookup and global memory sharing. This raises some interesting theoretical questions

A.1 BACKWARD PASS FOR MODEL SHARING WEIGHTS ACROSS DIFFERENT COMPONENTS

A general function sharing a weight, say x across different components can be written as , $f(x, g(x))$ The interpretation is that x was used in g(.) and then again used ahead in f. (In case of MLP, we can think of x being used in multiple layers)

Let $f(g_1, g_2)$ where both g_1 and g_2 are functions of x.

$$\frac{\partial f(g_1, g_2)}{\partial x} = \frac{\partial f(g_1, g_2)}{\partial g_1} * \frac{\partial g_1}{\partial x} + \frac{\partial f(g_1, g_2)}{\partial g_2} * \frac{\partial g_2}{\partial x} \tag{10}$$

$g_1 = x$ and $g_2 = g(x)$

$$\frac{\partial f(g_1, g_2)}{\partial x} = \frac{\partial f(x, g(y))}{\partial x}|_{y=x} + \frac{\partial f(y, g(x))}{\partial g(x)} * \frac{\partial g(x)}{\partial x}|_{y=x} \tag{11}$$

$$\frac{\partial f(g_1, g_2)}{\partial x} = \frac{\partial f(x, g(y))}{\partial x}|_{y=x} + \frac{\partial f(y, g(x))}{\partial x}|_{y=x} \tag{12}$$

Renaming,

$$\frac{\partial f(x, g(x))}{\partial x} = \frac{\partial f(z, g(y))}{\partial z}|_{y=x,z=x} + \frac{\partial f(z, g(y))}{\partial y}|_{y=x,z=x} \tag{13}$$

Thus, we can essentially consider each place where x appears as new variables and then gradient w.r.t x is just summation of partial derivatives of the function w.r.t these new variables. Thus, it is easy to implement this in the backward pass. In order to make sure that the memory utilization in backward pass is not of the order of the recovered model size, we do not use the auto-differentiation of tensorflow/pytorch. We implement our own backward pass and it can be found in the code.

A.2 GLOBAL FEATURE HASHING VS LOCAL FEATURE HASHING.

We can consider model compression techniques as dimensionality reduction of the parameter vector (a one dimensional vector of all parameters in a model) of size n into a vector of size $|\mathcal{M}| = m$. Quality of inner-product preservation is used as a metric to measure the quality of dimensionality reduction. In terms of dimensionality reduction, ROAST uses ROBE hashing Desai et al. (2022), which showed that chunk based hashing is theoretically better than hashing individual elements. In this section, we analyse GMS proposal of ROAST against LMS of HashedNet. For the purpose of this comparison we assume a chunk size of 1. Consider two parameter vectors $x, y \in R^n$. We are interested in how inner product between these parameter vectors are preserved under hashing. Let $x = [x_1 x_2 ... x_k]$ and $y = [y_1 y_2 ... y_k]$ be composed of k pieces of sizes $n_1, n_2, ... n_k$. In LMS, let each piece be mapped into memory of size $f_i m$ where $\sum_i f_i = 1$.

The estimators of inner product in the GMS case can be written as ,

$$\widehat{\langle x, y \rangle}_{G,m} = \sum_{j=1}^{m} (\sum_{i=1}^{n} \mathbb{I}(h(i)=j) g(i) x[i]) (\sum_{i=1}^{n} \mathbb{I}(h(i)=j) g(i) y[i]) \tag{14}$$

The estimate of inner product with LMS can be written as,

$$\widehat{\langle x, y \rangle}_{L,m,\vec{f}} = \sum_{l=1}^{k} \sum_{j=1}^{f_l m} (\sum_{i=1}^{n_l} \mathbb{I}(h(i)=j) g(i) x_l[i]) (\sum_{j=1}^{n_l} \mathbb{I}(h(i)=j) g(i) y_l[i]) = \sum_{l=1}^{k} \widehat{\langle x_l, y_l \rangle}_{G,(f_i m)} \tag{15}$$

Note that

$$\widehat{\langle x, y\rangle}_{L,m,\vec{f}} = \sum_{l=1}^{k} \widehat{\langle x_l, y_l\rangle}_{G,(f_l m)} \tag{16}$$

The GMS estimator is the standard feature hashing estimator and the LMS is essentially sum of GMS estimators for each of the piece. as $E[g(i)] = 0$, it is easy to check by linearity of expectations that **Expectation** The suffix L refers to local hashing and G refers to global hashing.

$$E_G = \mathbb{E}(\widehat{\langle x, y\rangle}_{G,m}) = \langle x, y\rangle \tag{17}$$

$$E_L = \mathbb{E}(\widehat{\langle x, y\rangle}_{L,m,\vec{f}}) = \langle x, y\rangle \tag{18}$$

Let us now look at the variance. Let us follow the following notation,

- $V_G = \mathbb{V}(\widehat{\langle x, y\rangle}_{G,m})$. GMS variance of entire vectors
- $V_L = \mathbb{V}(\widehat{\langle x, y\rangle}_{L,m,\vec{f}})$. LMS variance of entire vectors
- $V_l = \mathbb{V}(\widehat{\langle x_l, y_l\rangle}_{G,f_l m})$. variance of each piece

we can write V_l as follows. The following equation is easy to derive and it can be found the lemma 2 of Weinberger et al. (2009)

$$V_l = \frac{1}{f_l}\frac{1}{m}(\sum_{i\neq j} a_i^2 b_j^2 + \sum_{i\neq j} a_i b_i a_j b_j) \text{ where } x_l = (a_1, a_2 ... a_{n_l}) \text{ and } y_l = (b_1, b_2 ... b_{n_l}) \tag{19}$$

As, each of the piece is independently hashed in LSM, we can see

$$V_L = \sum_{l=1}^{k} V_l \tag{20}$$

Let us now look at V_G. Again, using lemma 2 from Weinberger et al. (2009)

$$V_G = \frac{1}{m}(\sum_{i\neq j} x_i^2 y_j^2 + \sum_{i\neq j} x_i y_i x_j y_j) \tag{21}$$

The expression can be split into terms that belong to same pieces and those across pieces

$$V_G = \frac{1}{m}\sum_{l=1}^{k}(\sum_{i\neq j\in\text{piece-l}} x_i^2 y_j^2 + \sum_{i\neq j\in\text{piece-l}} x_i y_i x_j y_j)$$
$$+ \frac{1}{m}\sum_{l1=1}^{k}\sum_{l2=1, l1\neq l2}^{k}(\sum_{i\in\text{piece-l1}, j\in\text{pieces-l2}} (x_i^2 y_j^2) + \sum_{i\in\text{piece-l1}, j\in\text{pieces-l2}} x_i y_i x_j y_j))$$

$$V_G = \sum_{l=1}^{k} f_l V_l + \frac{1}{m}\sum_{l1=1}^{l}\sum_{l2=1, l1\neq l2}^{l} ||x_{l1}||_2^2 ||y_{l2}||_2^2 + \langle x_{l1}, y_{l2}\rangle\langle x_{l2}, y_{l2}\rangle \tag{22}$$

Observation 1: In V_L we can see that there are terms with $\frac{1}{f_l}$ which makes it unbounded. It makes sense as if number of pieces increase a lot a lot of compressions will not work for example if number of peices $> |\mathcal{M}|$. Also, it will affect V_L a lot when some f_l is very small which can often be the case. For example, generally embedding tables in DLRM model are much larger than that of matrix multiplciation modules (MLP) . which can make $f \approx 0.001$ for MLP components.

Observation 2: Practically we can assume each piece, no matter the size of the vector, to be of same norm. The reason lies in initialization. According to Xavier's initialization the weights of a particular node are initialized with norm 1. So for now lets assume a more practical case of all norms being equal to $\sqrt{\alpha}$. Also, in order to make the comparisons we need to consider some average case over the

data. So let us assume that under independent randomized data assumption, the expected value of all inner products are β. With this , in expectation over randomized data, we have

$$V_G = \sum f_l V_l + \frac{k(k-1)}{m}(\alpha^2 + \beta^2) \tag{23}$$

Now note that,

$$V_l = \frac{1}{f_l}\frac{1}{m}(\sum_{i \neq j} a_i^2 b_j^2 + \sum_{i \neq j} a_i b_i a_j b_j) \text{ where } x_l = (a_1, a_2 ... a_{n_l}) \text{ and } y_l = (b_1, b_2 ... b_{n_l}) \tag{24}$$

(dropping the subscript "l" below)

$$V_l = \frac{1}{f_l}\frac{1}{m}((||x||_2^2 ||y||_2^2 + \langle x, y \rangle^2) - 2\sum_i x_i^2 y_i^2) \tag{25}$$

$$V_l = \frac{1}{f_l}\frac{1}{m}((\alpha^2 + \beta^2) - 2\sum_i x_i^2 y_i^2) \tag{26}$$

Note that for each negative term, there are n_l positive terms. To simplify we disregard this term in the equation above. This is an approximation which is practical and only made to get a sense of V_L and V_G relation.

$$\begin{aligned}
V_L - V_G &= \sum V_l - \sum f_l V_l - \frac{k(k-1)}{m}(\alpha^2 + \beta^2) \\
V_L - V_G &= \sum_l \frac{1}{m}(\frac{1}{f_l} - 1)((\alpha^2 + \beta^2)) - \frac{k(k-1)}{m}(\alpha^2 + \beta^2) \\
V_L - V_G &= \sum_l \frac{1}{m}(\frac{1}{f_l} - 1)((\alpha^2 + \beta^2) - \frac{k(k-1)}{m}(\alpha^2 + \beta^2) \\
V_L - V_G &\geq \frac{k(k-1)}{m}((\alpha^2 + \beta^2) - \frac{k(k-1)}{m}(\alpha^2 + \beta^2) \\
V_L - V_G &\geq 0
\end{aligned}$$

Note that we ignored a term which reduces the V_L a bit, Let the error be ϵ

$$V_L - V_G \geq -\epsilon \tag{27}$$

The above equation shows even for the best case, V_G might be slightly more than V_L. However for general case where harmonic mean is much worse than arithmetic mean, V_L will be much larger depending on exact f_l s

Table 6: Inference times of different square weight matrices using an input batch of 512. For ROAST, the tile parameters of each matrix multiplication are autotuned. The measurements were taken using TF32 on a NVIDIA A100 GPU (48GB). We used PyTorch's matmul function (MM) for the full uncompressed matrix multiplication. ■:bad ■: good

Inference time (ms)									
		Weight matrix dimensions (Dim × Dim)							
Model	$\mathcal{M}$ size ↓	512	1024	2048	4096	8096	10240	20480	Average
Full size →		1MB	4MB	16MB	64MB	128MB	420MB	1.6GB	
PyTorch-MM		0.10	0.11	0.12	0.22	0.69	1.18	3.91	0.91
HashedNet	4MB	0.31	0.34	0.63	2.02	6.20	9.67	35.22	7.77
	32MB	0.31	0.41	0.86	3.64	13.66	22.11	92.40	19.06
	64MB	0.31	0.46	1.09	6.47	31.21	42.45	178.07	37.15
	128MB	0.31	0.60	1.62	9.10	34.62	56.03	229.31	47.37
	256MB	0.32	0.62	1.82	10.25	38.28	62.67	256.22	52.88
	512MB	0.33	0.68	2.05	10.59	40.55	65.74	272.23	56.03
ROAST	4MB	0.28	0.30	0.27	0.48	0.99	1.36	4.83	1.22
	32MB	0.28	0.29	0.27	0.44	1.01	1.38	4.88	1.22
	64MB	0.28	0.29	0.27	0.44	1.00	1.40	4.93	1.23
	128MB	0.30	0.27	0.27	0.45	1.01	1.39	4.91	1.23
	256MB	0.30	0.27	0.27	0.44	1.01	1.40	4.90	1.23
	512MB	0.30	0.30	0.27	0.45	1.02	1.39	4.95	1.24

B ROAST-MM LATENCY MEASUREMENTS

B.1 INFERENCE OPTIMIZATION

B.2 TRAINING OPTIMIZATION

See tables 7, 8, 9, 10

		forward(ms) (optimized for forward + backward)							
		dim (Matrix dimension = dim x dim)							
	Memory (mb)	512	1024	2048	4096	8096	10240	20480	Average
Full (uncompressed)		0.16	0.12	0.12	0.24	0.66	0.91	3.03	0.75
HashedNet	4	0.37	0.35	0.65	2.04	6.23	9.62	35.64	7.84
	32	0.39	0.42	0.90	3.67	13.73	22.06	92.83	19.14
	64	0.33	0.47	1.11	6.45	25.78	42.51	178.20	36.41
	128	0.28	0.56	1.61	9.07	34.21	56.07	229.34	47.31
	256	0.20	0.54	1.72	9.95	38.17	62.47	258.11	53.02
	512	0.14	0.50	1.88	10.37	40.40	65.43	272.19	55.84
ROAST	4	0.30	0.31	0.31	0.50	1.43	2.01	7.54	1.77
	32	0.30	0.33	0.35	0.55	1.44	2.09	7.59	1.81
	64	0.29	0.31	0.33	0.56	1.45	2.08	7.80	1.83
	128	0.25	0.27	0.28	0.54	1.41	2.09	7.84	1.81
	256	0.16	0.18	0.19	0.46	1.33	2.02	7.82	1.74
	512	0.21	0.06	0.13	0.41	1.29	1.97	4.98	1.29

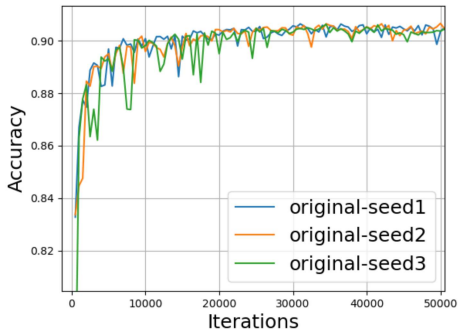
Table 7: Inference (forward pass time) for different shapes of square weight matrix with input batch of 512. The tile-parameters of multiplication are optimized for each function over "forward + backward" pass .The measurements are taken with tf32 on A100 (48GB)

C VARIANCE IN QUALITY OVER DIFFERENT RUNS

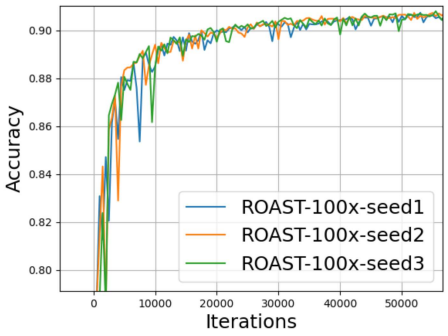
The figure 4 shows three runs of ROASTed BERT and BERT models

		backward(ms) (optimized for forward + backward)							
		dim (Matrix dimension = dim x dim)							
	Memory (mb)	512	1024	2048	4096	8096	10240	20480	Average
Full (uncompressed)		0.35	0.22	0.24	0.48	1.35	2.01	7.65	1.76
HashedNet	4	0.65	0.53	0.95	2.60	8.51	13.21	56.59	11.86
	32	0.68	0.69	1.80	6.36	24.13	38.95	160.54	33.31
	64	0.74	1.06	2.81	10.78	41.35	67.02	271.86	56.52
	128	0.91	1.34	3.40	12.41	51.00	81.25	337.31	69.66
	256	1.29	1.84	4.02	14.57	58.03	91.18	376.83	78.25
	512	2.08	2.62	4.90	16.24	62.45	98.46	391.46	82.60
ROAST	4	0.54	0.54	0.60	1.20	2.54	3.72	13.99	3.30
	32	0.57	0.61	0.69	1.06	2.71	4.04	15.07	3.54
	64	0.64	0.73	0.77	1.17	2.82	4.18	15.50	3.69
	128	0.79	0.81	0.89	1.38	3.17	4.73	18.30	4.30
	256	1.19	1.17	1.27	1.77	3.56	5.17	18.33	4.64
	512	2.11	1.92	2.12	2.53	4.33	5.98	22.71	5.96

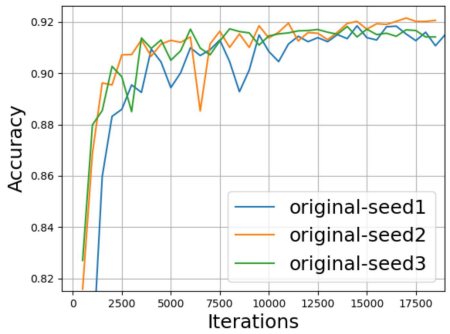
Table 8: Backward pass for different shapes of square weight matrix with input batch of 512. The tile-parameters of multiplication are optimized for each function over "forward + backward" pass .The measurements are taken with tf32 on A100 (48GB)



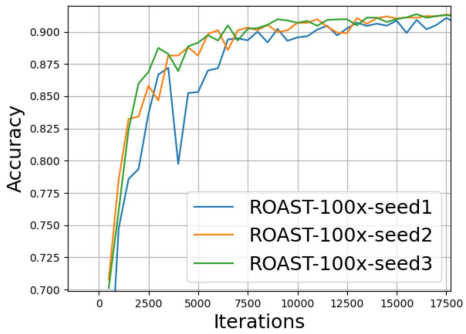
(a) yelp - original



(b) yelp - ROAST100x



(c) ag_news - original



(d) ag_news - ROAST100x

Figure 4: Three runs of original and ROAST-100x runs

		update weights (optim.step())(ms) (optimized for forward + backward)								
		dim (Matrix dimension = dim x dim)								
optim	*Model*	*msize*	512	1024	2048	4096	8096	10240	20480	Average
adagrad	Full		0.14	0.11	0.15	0.60	2.16	3.41	13.45	2.86
	HashedNet	4	0.14	0.11	0.11	0.11	0.11	0.12	0.54	0.18
		32	0.35	0.33	0.33	0.33	0.33	0.34	0.36	0.34
		64	0.61	0.61	0.61	0.61	0.61	0.62	0.61	0.61
		128	1.15	1.14	1.14	1.14	1.15	1.19	1.18	1.15
		256	2.22	2.21	2.21	2.21	2.22	2.26	3.87	2.46
		512	4.36	4.36	4.35	4.35	4.37	4.40	4.47	4.38
	ROAST	4	0.11	0.11	0.11	0.11	0.12	0.11	0.11	0.11
		32	0.33	0.34	0.34	0.33	0.33	0.33	0.33	0.33
		64	0.60	0.61	0.61	0.60	0.61	0.61	0.61	0.61
		128	1.14	1.14	1.14	1.14	1.14	1.14	1.14	1.14
		256	2.21	2.21	2.21	2.21	2.21	2.21	2.21	2.21
		512	4.38	4.35	4.36	4.35	4.35	4.36	4.35	4.36
adam	Full		0.15	0.15	0.23	1.06	3.89	6.18	24.47	5.16
	HashedNet	4	0.15	0.23	0.16	0.16	0.16	0.16	0.16	0.17
		32	0.57	0.57	0.57	0.57	0.57	0.57	0.59	0.57
		64	1.06	1.06	1.06	1.06	1.06	1.06	1.16	1.08
		128	2.03	2.05	2.04	2.04	2.05	2.04	2.23	2.07
		256	3.98	3.99	3.98	3.99	4.00	4.00	4.22	4.02
		512	7.89	7.89	7.89	7.89	7.91	7.90	8.13	7.93
	ROAST	4	0.15	0.23	0.15	0.16	0.16	0.15	0.16	0.17
		32	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57
		64	1.07	1.06	1.06	1.06	1.06	1.07	1.06	1.06
		128	2.05	2.03	2.04	2.04	2.03	2.04	2.04	2.04
		256	4.01	3.98	3.99	3.99	3.99	3.99	3.99	3.99
		512	7.89	7.89	7.89	7.89	7.89	7.89	7.89	7.89
sgd	Full		0.08	0.07	0.08	0.20	0.62	0.97	3.92	0.85
	HashedNet	4	0.08	0.07	0.08	0.07	0.07	0.08	0.08	0.08
		32	0.12	0.12	0.12	0.12	0.12	0.12	0.17	0.13
		64	0.19	0.20	0.20	0.20	0.20	0.21	0.31	0.22
		128	0.35	0.34	0.34	0.35	0.35	0.37	0.48	0.37
		256	0.64	0.64	0.64	0.64	0.65	0.67	0.83	0.67
		512	1.23	1.23	1.23	1.23	1.25	1.24	1.25	1.24
	ROAST	4	0.07	0.07	0.07	0.08	0.07	0.07	0.23	0.10
		32	0.12	0.12	0.13	0.12	0.12	0.12	0.12	0.12
		64	0.22	0.19	0.20	0.19	0.19	0.20	0.29	0.21
		128	0.34	0.35	0.34	0.34	0.34	0.35	0.40	0.35
		256	0.64	0.65	0.64	0.64	0.64	0.65	0.64	0.64
		512	1.27	1.23	1.23	1.23	1.28	1.23	1.62	1.30

Table 9: Weight update operation (optimizer.step()) for different shapes of square weight matrix with input batch of 512. The tile-parameters of multiplication are optimized for each function over "forward + backward" pass .The measurements are taken with tf32 on A100 (48GB)

		total = fwd + bkwd + optimize (ms) (optimized for forward + backward)								
		dim (Matrix dimension = dim x dim)								
optim	*Model*	*msize*	512	1024	2048	4096	8096	10240	20480	Average
adagrad	Full		0.65	0.46	0.51	1.32	4.17	6.33	24.13	5.37
	HashedNet	4	1.16	0.99	1.71	4.74	14.86	22.95	92.78	19.88
		32	1.43	1.44	3.03	10.37	38.19	61.35	253.72	52.79
		64	1.68	2.14	4.53	17.83	67.74	110.15	450.66	93.53
		128	2.34	3.04	6.15	22.62	86.36	138.51	567.83	118.12
		256	3.71	4.59	7.95	26.73	98.42	155.92	638.80	133.73
		512	6.58	7.47	11.13	30.96	107.21	168.30	668.12	142.83
	ROAST	4	0.95	0.95	1.02	1.81	4.09	5.84	21.64	5.19
		32	1.21	1.27	1.38	1.94	4.49	6.46	23.00	5.68
		64	1.54	1.64	1.70	2.34	4.87	6.86	23.90	6.12
		128	2.18	2.22	2.31	3.06	5.72	7.97	27.28	7.25
		256	3.57	3.56	3.67	4.43	7.10	9.40	28.35	8.58
		512	6.70	6.32	6.62	7.29	9.97	12.31	32.04	11.61
adam	Full		0.50	0.48	0.60	1.78	5.89	9.11	35.01	7.62
	HashedNet	4	1.00	1.56	1.76	4.81	14.94	23.07	86.76	19.13
		32	1.43	1.78	3.29	10.60	38.45	61.64	253.20	52.91
		64	2.03	2.63	4.97	18.35	68.28	110.63	450.86	93.96
		128	3.18	4.27	7.02	23.54	87.47	139.30	568.72	119.07
		256	5.45	6.30	9.71	28.66	100.19	157.55	633.80	134.52
		512	10.08	10.94	14.64	34.56	110.71	171.67	672.24	146.41
	ROAST	4	1.00	1.27	1.05	1.86	4.06	5.89	21.71	5.26
		32	1.45	1.56	1.52	2.21	4.72	6.69	23.28	5.92
		64	2.13	2.02	2.18	2.80	5.34	7.39	24.35	6.60
		128	3.26	3.11	3.23	3.95	6.62	8.85	28.22	8.18
		256	5.82	5.33	5.45	6.21	8.97	11.15	30.19	10.45
		512	9.82	9.87	10.14	10.90	13.52	15.82	35.59	15.09
sgd	Full		0.44	0.43	0.46	0.90	2.62	3.90	14.68	3.35
	HashedNet	4	1.25	0.95	1.70	4.72	14.76	22.96	86.70	19.01
		32	0.99	1.23	2.86	10.17	38.10	61.16	252.99	52.50
		64	1.16	1.84	4.11	17.51	67.28	109.78	450.34	93.15
		128	1.59	2.24	5.28	21.84	85.46	137.54	566.88	117.26
		256	2.21	3.00	6.35	25.19	96.91	154.43	630.75	131.26
		512	3.42	4.28	8.06	27.91	104.03	164.94	665.29	139.70
	ROAST	4	0.92	0.92	0.98	1.79	3.94	5.82	22.39	5.25
		32	0.95	1.00	1.17	1.75	4.28	6.25	22.77	5.45
		64	1.62	1.15	1.26	1.92	4.45	6.45	24.01	5.84
		128	1.38	1.44	1.52	2.26	4.90	7.25	27.18	6.56
		256	2.04	2.10	2.14	2.85	5.53	7.91	26.98	7.08
		512	3.56	3.20	3.36	4.18	7.10	9.17	31.20	8.82

Table 10: Total training step time for different shapes of square weight matrix with input batch of 512. The tile-parameters of multiplication are optimized for each function over "forward + backward" pass .The measurements are taken with tf32 on A100 (48GB)