MIHash: Online Hashing with Mutual Information Supplementary Material

1. Implementation Details of MIHash

We discuss the implementation details of MIHash. In the online hashing experiments, for simplicity we model MIHash using linear hash functions, in the form of $\phi_i(\mathbf{x}) = \operatorname{sgn}(w_i^\top \mathbf{x}) \in \{-1, +1\}, i = 1, \ldots, b$. The learning capacity of such a model is lower than the kernel-based OKH, and is the same as OSH, AdaptHash, and SketchHash, which use linear hash functions as well.

For the batch hashing experiments, as mentioned in the paper, we similarly model MIHash using linear hash functions in the first setting, but perform end-to-end learning with the VGG-F network in the second setting. In this case, the hash functions become $\phi_i(\mathbf{x}) = \operatorname{sgn}(f_i(x; w)) \in \{-1, +1\}, i = 1, \ldots, b$, where f_i are the logits produced by the previous layer in the network.

We train MIHash using stochastic gradient descent. In Eq. 11 in the paper, we gave the gradients of the mutual information objective $\mathcal{I}$ with respect to the *outputs* of the hash mapping, $\Phi(\mathbf{x})$. Both $\mathcal{I}$ and $\partial\mathcal{I}/\partial\Phi(\mathbf{x})$ are parameter-free. In order to further back-propagate gradients to the *inputs* of $\Phi(\mathbf{x})$ and model parameters $\{w_i\}$, we approximate the sgn function using the sigmoid function σ:

$$\phi_i(\mathbf{x}) \approx 2\sigma(A w_i^\top \mathbf{x}) - 1, \tag{1}$$

where $A > 1$ is a scaling parameter, used to increase the "sharpness" of the approximation. We find A from the set $\{10, 20, 30, 40, 50\}$ in our experiments.

We note that A is not a tuning parameter of the mutual information objective, but rather a parameter of the underlying hash functions. The design of the hash functions is not coupled with the mutual information objective, thus can be separated. It will be an interesting topic to explore other methods of constructing hash functions, potentially in ways that are free of tuning parameters.

2. Experimental Details

2.1. The streaming scenario

We set up a streaming scenario in our online hashing experiments. We run three randomized trials for each experiment. In each trial, we first randomly split the dataset into a retrieval set and a test set as described in Sec. 4.1 in the paper, and randomly sample the training subset from the retrieval set. The ordering of the training set is also randomly permuted. The random seeds are fixed, so the baselines and methods with the Trigger Update module observe the same training sequences.

In a streaming setting, we also measure the *cumulative* retrieval performance during online hashing, as opposed to only the final results. To mimic real retrieval systems where queries arrive randomly, we set 50 randomized checkpoints during the online process. We first place the checkpoints with equal spacing, then add small random perturbations to their locations. We measure the instantaneous retrieval mAP at these checkpoints to get mAP vs. time curves (*e.g.* curves shown in Fig. 5 in the paper), and compute the area under curve (AUC). AUC gives a summary of the entire online learning process, which cannot be reflected by the final performance at the end.

2.2. Parameters for online hashing methods

We describe parameters used for online hashing methods in the online experiments. Some of the competing methods require parameter tuning, therefore we sample a validation set from the training data and find the best performing parameters for each method. The size of the validation sets are 2K, 2K and 10K for CIFAR-10, LabelMe and Places205, respectively. Please refer to the respective papers for the descriptions of the parameters.

- **OSH**: η is set to 0.1 for all datasets. The ECOC codebook C is populated the same way as in OSH.
- **AdaptHash**: the tuple (α, λ, η) is set to $(0.9, 0.01, 0.1)$, $(0.1, 0.01, 0.001)$ and $(0.9, 0.01, 0.1)$ for CIFAR-10, LabelMe and Places205, respectively.
- **OKH**: the tuple (C, α) is set to $(0.001, 0.3)$, $(0.001, 0.3)$ and $(0.0001, 0.7)$ for CIFAR-10, LabelMe and Places205, respectively.
- **SketchHash**: the pair (sketch size, batch size) is set to $(200, 50)$, $(100, 50)$ and $(100, 50)$ for CIFAR-10, LabelMe and Places205, respectively.

2.3. Parameters for batch hashing methods

We use the publicly available implementations for the compared methods, and exhaustively search parameter settings, including the default parameters as provided by the

Method	Training Time (s)
OKH	10.8
OKH + TU	23.6
OSH	97.6
OSH + TU	175.8
AdaptHash	47.8
AdaptHash + TU	94.8
SketchHash	68.8
SketchHash + TU	80.0

Table 1. Online hashing: running times on the CIFAR-10 20k training set, with 32-bit hash codes. For methods with the TU plugin, the added time is due to maintaining the reservoir set and computing the mutual information update criterion, and is dominated by the maintaining of the reservoir set.

Method	mAP	Training Time (s)
SHK	0.682	180
SDH	0.592	4.8
FastHash	0.738	140
VDSH*	0.554	206
DPSH	0.553	450
DTSH	0.702	1728
MIHash, 1ep	0.609	1.9
MIHash	0.746	190

Table 2. Batch hashing: test performance and training time for 48-bit codes on the CIFAR-10, using the 5k training set. *VDSH is trained with the full model as detailed in 2.3. 1ep stands for training for one epoch only.

authors. For DPSH and DTSH we found a combination that worked well for the first setting: the mini-batch size is set to the default value of 128, and the learning rate is initialized to 1 and decayed by a factor of 0.9 after every 20 epochs. Additionally, for DTSH, the margin parameter is set to $b/4$ where b is the hash code length. VDSH uses a heavily customized architecture with only fully-connected layers, and it is unclear how to adapt it to work with standard CNN architectures. In this sense, VDSH is more akin to nonlinear hashing methods such as FastHash and SHK. We used the full VDSH model with 16 layers and 1024 nodes per layer, and found the default parameters to perform the best, except that we increased the number of training iterations by an order of magnitude during finetuning.

For MIHash, in the first setting we use a batch size of 100, and run SGD with initial learning rate of 0.1 and a decay factor of 0.5 every 10 epochs, for 100 epochs. For the second setting where we finetune the pretrained VGG-F network, batch size is 250, learning rate is initially set to 0.001 and decayed by half every 50 epochs.

3. Running Time

3.1. Online Setting: Trigger Update Module

In Table 1 we report running time for all methods on the CIFAR-10 dataset with 20k training examples, including time spent in learning hash functions and the added processing time for maintaining the reservoir set and computing TU. Numbers are recorded on a 2.3GHz Intel Xeon E5-2650 CPU workstation with 128GB of DDR3 RAM. Most of the added time is due to maintaining the reservoir set, which is invoked in each training iteration; the mutual information update criterion is only checked after processing every $U = 100$ examples. Methods with small batch sizes (*e.g.* OSH, batch size 1) therefore incur more overhead than methods with larger batches (*e.g.* SketchHash, batch size 50). Results for other datasets are similar.

We note that in a real retrieval system with large-scale data, the bottleneck likely lies in recomputing the hash tables for indexed data, due to various factors such as scheduling and disk I/O. We reduce this bottleneck significantly by using TU. Compared to this bottleneck, the increase in training time is not significant.

3.2. Batch Setting

Table 2 reports CPU times for learning 48-bit hash mappings in the first experimental setting on CIFAR-10 (5K training set). Retrieval mAP are replicated from Table 1 in the paper. For learning a single layer, our Matlab implementation of MIHash achieves 1.9 seconds per epoch on CPU. MIHash achieves competitive performance with a single epoch, and has a total training time on par with FastHash, while yielding superior performance.

4. Additional Experimental Results

4.1. Online Hashing: Other Code Lengths

In the online hashing experiments we reported in the paper, all online hashing methods are compared in the same setup with 32-bit hash codes. Additionally, we also present results using 64-bit hash codes on all three datasets. The parameters for all methods are found through validation as described in 2.2.

Similar to Sec 4.2 in the paper, we show the comparisons with and without TU for existing online hashing methods in Fig. 1, and plot the mAP curves for all methods, including MIHash, in Fig. 2. The 64-bit results are uniformly better than 32-bit results for all methods in terms of mAP, but still follow the same patterns. Again, we can see that MIHash clearly outperforms all competing online hashing methods, and shows potential for improvement given more training data.

4.2. Parameter Study: θ

We present a parameter study on the parameter θ, the improvement threshold on the mutual information criterion in TU. In our previous experiments, we found the default $\theta = 0$

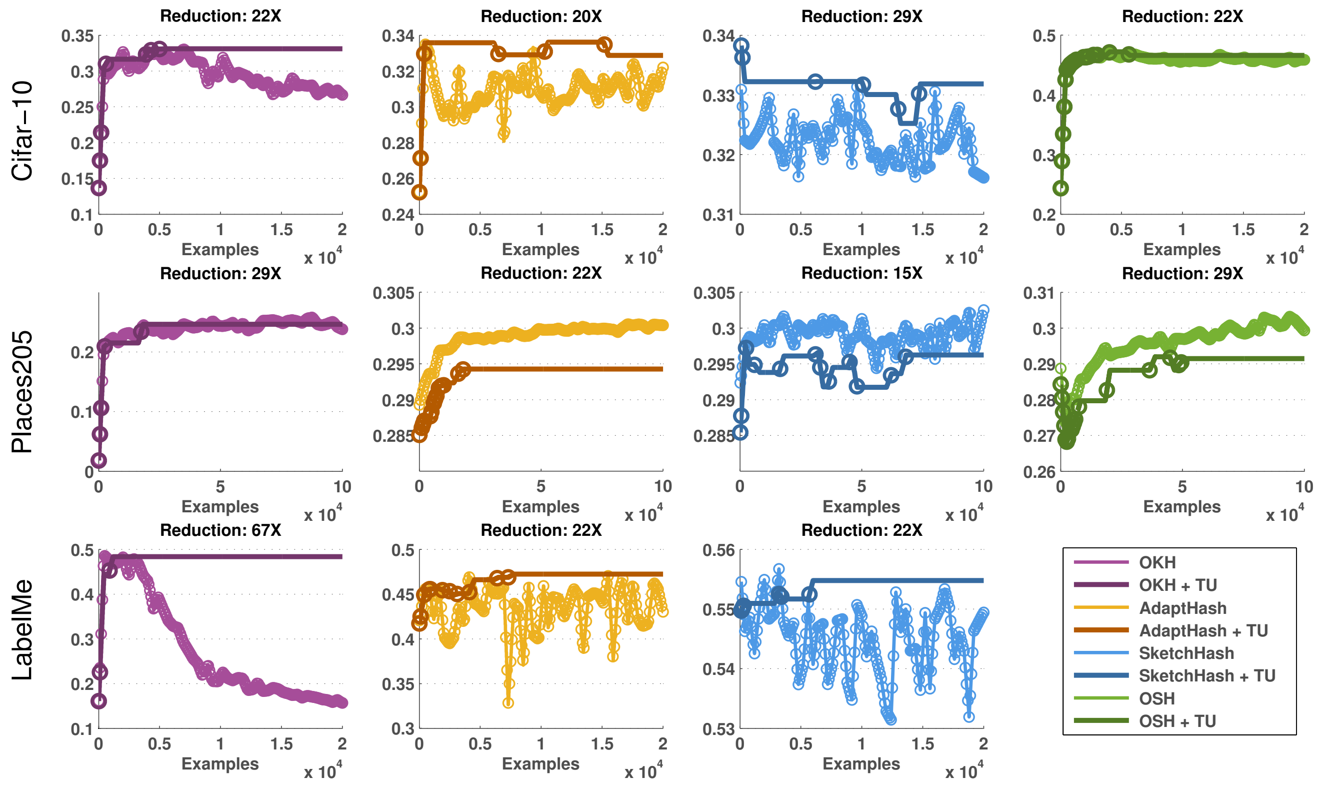

Figure 1. 64-bit experiments: Retrieval mAP *vs.* number of training examples for four existing online hashing methods on the three datasets, with and without Trigger Update (TU). We use default threshold $\theta = 0$ for TU. Circles indicate hash table updates, and the ratio of reduction in the number of updates is marked for each graph. TU substantially reduces the number of updates while having a stabilizing effect on the retrieval performance. Note: since the OSH method assumes supervision in terms of class labels, it is not applicable to the unsupervised LabelMe dataset.

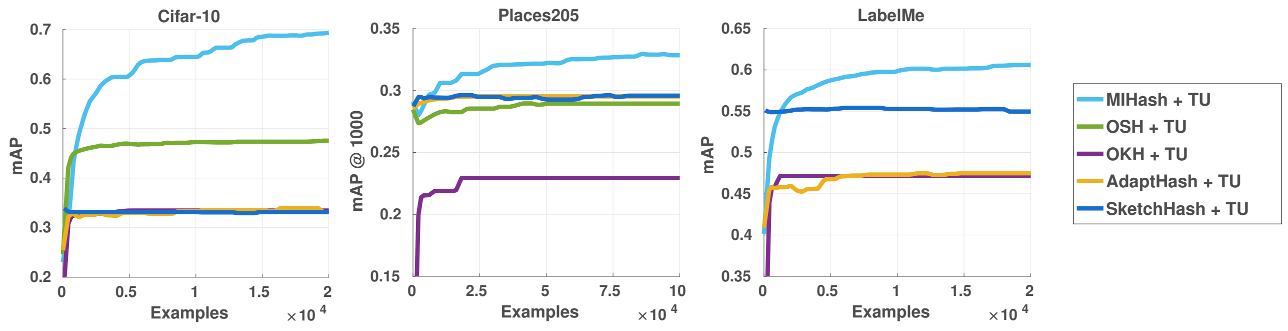

Figure 2. 64-bit experiments: Online hashing performance (mAP) comparison on three datasets, where all methods use the Trigger Update module (TU) with $\theta = 0$. Using the mutual information objective, MIHash clearly outperforms other methods. OKH, AdaptHash, ad SketchHash perform very similarly on CIFAR-10. OSH, AdaptHash, ad SketchHash perform very similarly on Places205. Again, the OSH method is not applicable to the unsupervised LabelMe dataset.

to work well, and did not specifically tune θ. However, tuning for a larger θ could lead to better trade-offs, since small improvements in the quality of the hash mapping may not justify the cost of a full hash table update.

For this study, we vary parameter θ from $-\infty$ to ∞ for all methods (with 32-bit hash codes). $\theta = -\infty$ reduces to the baseline. On the other hand, $\theta = \infty$ prevents any updates to the initial hash mapping and hash table, and results in only one hash table update (for the initial mapping) and typically low performance. The performance metric we focus on in this study is the cumulative metric, AUC, since it better summarizes the entire online learning process than the final performance alone.

We use a custom update schedule for SketchHash: we enforce hash table updates in the early iterations regardless of other criteria, until the number of observed examples reaches the specified size of the "data sketch", which SketchHash uses to perform a batch hashing algorithm. This was observed to be critical for the performance of SketchHash. Therefore, the number of hash table updates for SketchHash can be greater than 1 even for $\theta = \infty$.

We present full results in Tables 3, 4, 5. In all cases, we observe a substantial decrease in the number of hash table updates as θ increases. With reasonable θ values (typically around 0), the number of hash table updates can be reduced by over an order of magnitude with no loss in AUC. Note that the computation-performance trade-off achieved by the default $\theta = 0$ is always among the best, thereby in practice it can be used without tuning.

4.3. Parameter Study: U

We simulate a data-agnostic baseline that updates hash tables at a constant rate, using the update interval parameter U. In the paper, U is set such that the baseline updates a total of 201 times for all datasets. This ensures that the baseline is never too outdated (compared to 50 checkpoints at which performance is evaluated), but is still fairly infrequent: the smallest U in this case is 100, which means the baselines process at least 100 training examples before recomputing the hash table. For completeness, here we present the results using different values of U, where all methods again use 32-bit hash codes and the default $\theta = 0$.

We used a simple rule that avoids unnecessary hash table updates if the hash mapping itself does not change. Specifically, we do not update if $\|\Phi_t - \Phi^s\| < 10^{-6}$, where Φ^s is the current snapshot and Φ_t is the new candidate. Some baseline entries have fewer updates because of this rule (*e.g.* AdaptHash on Places205). And as explained before, due to the custom update schedule, SketchHash may have more hash table updates than what is suggested by U.

Please see Tables 6, 7, 8 for the full results. In all experiments, we run three random trials and average the results as mentioned before, and the standard deviation of mAP and

CIFAR-10, 32 bits			
OKH	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.1	201	0.259	–
-0.01	190 (5.8x)	0.260	$+0.4\%$
-10^{-4}	8.0 (25.1x)	0.287	$+10.8\%$
0	8.0 (25.1x)	0.287	$+10.8\%$
10^{-4}	7.7 (26.1x)	0.287	$+10.8\%$
0.01	3.3 (91.2x)	0.280	$+8.1\%$
≥ 0.2	1.0 (201x)	0.134	-48.3%
OSH	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.01	201	0.463	–
-10^{-4}	39.0 (5.2x)	0.466	$+0.6\%$
0	36.7 (5.5x)	0.466	$+0.6\%$
10^{-4}	35.7 (5.6x)	0.466	$+0.6\%$
0.01	6.7 (30x)	0.453	-2.1%
0.1	2.0 (100x)	0.386	-16%
≥ 0.3	1.0 (201x)	0.207	-55%
AdaptHash	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.1	201	0.218	–
-0.01	68.3 (2.9x)	0.238	$+9.2\%$
-10^{-4}	10.3 (19.5x)	0.250	$+14.7\%$
0	10.0 (20.1x)	0.250	$+14.7\%$
10^{-4}	10.0 (20.1x)	0.250	$+14.7\%$
0.01	3.3 (60.9x)	0.244	$+11.9\%$
≥ 0.1	1.0 (201x)	0.211	-3.3%
SketchHash	**HT updates**	**AUC**	**ΔAUC**
≤ -0.01	201	0.304	–
-10^{-4}	9.0 (22.3x)	0.318	$+4.6\%$
-10^{-6}	7.3 (27.5x)	0.319	$+4.9\%$
0	7.3 (27.5x)	0.319	$+4.9\%$
10^{-4}	7.3 (27.5x)	0.319	$+4.9\%$
0.01	4.3 (46.7x)	0.318	$+4.6\%$
≥ 0.1	4.0 (50.3x)	0.314	$+3.3\%$

Table 3. Parameter study on the threshold value θ for online hashing methods on **CIFAR-10** (32 bits). We report the number of hash table updates, where 100x indicates a 100 times reduction with respect to the baseline. We also report the area under the mAP curve (AUC) and compare to baseline.

AUC scores are less than 0.01. Generally, using smaller U leads to more updates by both the baselines and methods with `TU`; recall that U is also a parameter of `TU` which specifies the frequency of checking the update criterion. However, methods with the `TU` module appear to be quite insensitive to the choice of U, *e.g.* the number of updates for SketchHash with `TU` on CIFAR-10 only increases by 2x while U is reduced by 20x, from 1000 to 50. We attribute this to the ability of `TU` to filter out unnecessary updates. Across different values of U, `TU` consistently brings computational savings while preserving/improving online hashing performance, as indicated by final mAP and AUC.

Places205, 32 bits			
OKH	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.01	201	0.163	–
-10^{-4}	8.3 (24.2x)	0.161	-1.2%
-10^{-6}	7.0 (28.7x)	0.161	-1.2%
0	7.0 (28.7x)	0.161	-1.2%
10^{-6}	7.0 (28.7x)	0.161	-1.2%
10^{-4}	5.7 (35.3x)	0.161	-1.2%
0.01	2.0 (100x)	0.123	-25%
≥ 0.1	1.0 (201x)	0.014	-91%
OSH	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.001	201	0.246	–
-20^{-4}	101 (2.0x)	0.246	0%
-10^{-4}	9.3 (21.6x)	0.236	-4.1%
0	7.0 (28.7x)	0.236	-4.1%
10^{-4}	5.7 (35.3x)	0.230	-6.5%
10^{-3}	2.7 (74.4x)	0.224	-8.9%
≥ 0.1	1.0 (201x)	0.226	-8.1%
AdaptHash	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.01	199.7	0.237	–
-10^{-4}	199 (1.0x)	0.237	0%
-10^{-6}	9.7 (20.6x)	0.236	-0.4%
0	8.7 (23.0x)	0.236	-0.4%
10^{-6}	8.7 (23.0x)	0.235	-0.8%
10^{-4}	3.0 (66.6x)	0.235	-0.8%
≥ 0.01	1.0 (201x)	0.227	-3.4%
SketchHash	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.01	201	0.237	–
-10^{-4}	52.3 (3.8x)	0.238	$+0.4\%$
-10^{-6}	15.3 (12.6x)	0.238	$+0.4\%$
0	12.7 (15.8x)	0.236	-0.4%
10^{-6}	15.3 (13.1x)	0.238	$+0.4\%$
10^{-4}	7.0 (28.7x)	0.239	$+0.8\%$
≥ 0.01	2.0 (101x)	0.223	-5.9%

Table 4. Parameter study on the threshold value θ for online hashing methods on **Places205** (32 bits). We report the number of hash table updates, where 100x indicates a 100 times reduction with respect to the baseline. We also report the area under the mAP curve (AUC) and compare to baseline.

LabelMe, 32 bits			
OKH	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.2	201	0.198	–
-0.1	196 (1.0x)	0.199	$+0.5\%$
-0.01	2.7 (74.4x)	0.373	$+88\%$
-10^{-6}	2.3 (87.4x)	0.374	$+89\%$
0	2.3 (87.4x)	0.374	$+89\%$
10^{-6}	2.3 (87.4x)	0.374	$+89\%$
0.01	2.0 (101x)	0.372	$+88\%$
≥ 0.6	1.0 (201x)	0.111	-44%
AdaptHash	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.1	201	0.333	–
-10^{-6}	149 (1.3x)	0.330	-0.9%
-10^{-4}	9.3 (21.6x)	0.365	$+9.6\%$
-10^{-2}	8.7 (23.1x)	0.365	$+9.6\%$
0	5.3 (37.9x)	0.369	$+11\%$
10^{-6}	8.7 (23.1x)	0.365	$+9.6\%$
10^{-4}	8.3 (24.2x)	0.358	$+7.5\%$
10^{-2}	2.7 (74.4x)	0.351	$+5.4\%$
≥ 0.1	1 (201x)	0.296	-11%
SketchHash	**HT Updates**	**AUC**	**ΔAUC**
≤ -0.1	201	0.446	–
-10^{-2}	195 (1.0x)	0.446	0%
-10^{-4}	9.3 (21.6x)	0.460	$+3.1\%$
0	8.7 (23.1x)	0.460	$+3.1\%$
10^{-4}	10 (20.1x)	0.459	$+2.9\%$
10^{-2}	4.7 (42.8x)	0.446	0%
≥ 0.1	4.0 (50.3x)	0.439	-1.6%

Table 5. Parameter study on the threshold value θ for online hashing methods on **LabelMe** (32 bits). We report the number of hash table updates, where 100x indicates a 100 times reduction with respect to the baseline. We also report the area under the mAP curve (AUC) and compare to baseline. Note: OSH is not applicable to this unlabeled dataset since it needs supervision in terms of class labels.

CIFAR-10, 32 bits				
Method	**TU**	**HT Updates**	**Final mAP**	**AUC (mAP)**
OKH, $U = 10$	×	1870	0.238	0.259
	✓	15.6 (119.3x)	0.297	0.293 (+13%)
OKH, $U = 100$	×	201	0.238	0.259
	✓	8 (25.1x)	0.291	0.287 (+10.8%)
OKH, $U = 1000$	×	21	0.238	0.255
	✓	2.6 (8x)	0.282	0.273 (+7%)
OSH, $U = 10$	×	2001	0.480	0.463
	✓	110.7 (18x)	0.483	0.466 (+0.6%)
OSH, $U = 100$	×	201	0.480	0.463
	✓	36.7 (5.4x)	0.483	0.466 (+0.6%)
OSH, $U = 1000$	×	21	0.480	0.454
	✓	11.3 (1.9x)	0.479	0.454
AdaptHash, $U = 10$	×	2001	0.244	0.224
	✓	19.6 (101.7x)	0.267	0.261 (+16%)
AdaptHash, $U = 100$	×	201	0.244	0.224
	✓	10.0 (10.1x)	0.255	0.250 (+11.6%)
AdaptHash, $U = 1000$	×	21	0.244	0.222
	✓	5 (4.2x)	0.252	0.234 (+5%)
SketchHash, $U = 50$	×	400	0.306	0.303
	✓	9 (44.4x)	0.318	0.318 (+5%)
SketchHash, $U = 100$	×	202	0.306	0.304
	✓	7.3 (27.5x)	0.320	0.319 (+4.9%)
SketchHash, $U = 1000$	×	24	0.306	0.305
	✓	4.6 (5.2x)	0.317	0.314 (+2.9%)

Table 6. Online hashing results (32 bits) with different update interval parameters (U) on the **CIFAR-10** dataset. All results are averaged from 3 random trials. For the number of hash table updates, we report the reduction ratio (*e.g.* 8x) for TU. For AUC, we report the relative change compared to baseline. Note: SketchHash uses a batch size of 50, therefore the smallest U is set to 50.

LabelMe, 32 bits				
Method	**TU**	**HT Updates**	**Final mAP**	**AUC (mAP)**
OKH, $U = 10$	×	2001	0.119	0.200
	✓	8 (250x)	0.382	0.377 (+88.5%)
OKH, $U = 100$	×	201	0.119	0.200
	✓	2.3 (86.2x)	0.380	0.374 (+87%)
OKH, $U = 1000$	×	21	0.119	0.193
	✓	2 (10.5x)	0.373	0.357 (+85%)
AdaptHash, $U = 10$	×	2001	0.318	0.319
	✓	12.6 (157.9x)	0.380	0.371 (+16.3%)
AdaptHash, $U = 100$	×	201	0.318	0.318
	✓	8.6 (23.1x)	0.379	0.365 (+14.7%)
AdaptHash, $U = 1000$	×	21	0.318	0.317
	✓	5 (4.2x)	0.343	0.337 (+6.3%)
SketchHash, $U = 50$	×	400	0.445	0.447
	✓	9.6 (41.6x)	0.461	0.460 (+2%)
SketchHash, $U = 100$	×	202	0.445	0.446
	✓	8.67 (23.2x)	0.462	0.460 (+3.1%)
SketchHash, $U = 1000$	×	24	0.445	0.445
	✓	8.3 (2.8x)	0.456	0.455 (+2%)

Table 7. Online hashing results (32 bits) with different update interval parameters (U) on the **LabelMe** dataset. All results are averaged from 3 random trials. For the number of hash table updates, we report the reduction ratio (*e.g.* 8x) for TU. For AUC, we report the relative change compared to baseline. Note: since LabelMe is an unsupervised dataset, the OSH method is not applicable since it requires supervision in the form of class labels.

Places205, 32 bits				
Method	**TU**	**HT Updates**	**Final mAP**	**AUC (mAP)**
OKH, $U = 50$	×	2001	0.182	0.163
	✓	8 (250.1x)	0.173	0.169 (+3.7%)
OKH, $U = 500$	×	201	0.182	0.163
	✓	7 (28.7x)	0.165	0.161 (-1.2%)
OKH, $U = 5000$	×	21	0.182	0.156
	✓	2 (10.5x)	0.157	0.148 (-5.1%)
OSH, $U = 50$	×	2001	0.248	0.246
	✓	25 (80x)	0.239	0.238 (-3%)
OSH, $U = 500$	×	201	0.248	0.246
	✓	7 (28.7x)	0.236	0.236 (-4.0%)
OSH, $U = 5000$	×	21	0.248	0.245
	✓	2 (10.5x)	0.234	0.233 (-4%)
AdaptHash, $U = 50$	×	823.7	0.238	0.237
	✓	26.6 (30.8x)	0.236	0.236 (-0.4%)
AdaptHash, $U = 500$	×	200	0.238	0.237
	✓	8.6 (23.0x)	0.236	0.236 (-0.4%)
AdaptHash, $U = 5000$	×	21	0.238	0.237
	✓	3 (7x)	0.236	0.236 (-0.4%)
SketchHash, $U = 50$	×	2000	0.238	0.235
	✓	19.3 (103.4x)	0.236	0.235 (0%)
SketchHash, $U = 500$	×	202	0.237	0.235
	✓	15.3 (13.1x)	0.240	0.238 (+1.2%)
SketchHash, $U = 5000$	×	22	0.235	0.235
	✓	6.6 (3.2x)	0.239	0.238 (+1.2%)

Table 8. Online hashing results (32 bits) with different update interval parameters (U) on the **Places205** dataset. All results are averaged from 3 random trials. For the number of hash table updates, we report the reduction ratio (*e.g.* 8x) for TU. For and AUC, we report the relative change compared to baseline.