Supplementary Material

In this document, we include the full object detection evaluation table reporting accuracy and speedup in all configurations (see Figure 2). Furthermore, we explain in more detail how the convolutions are implemented to achieve high efficiency (Section 1) and perform ablation studies on threshold tuning (Section 2).

1. Efficient convolution on GPUs

Designing an efficient convolutional layer that is on par with today's leading framework, *cuDNN*, requires extensive profiling, analysis and optimizations. Minimization of memory transfers and logic operations, as well as optimizing local memory usage (registers and shared memory) are key in achieving a high performance for compute-heavy operations like convolutions. In this section, we discuss our design decisions and perform ablation studies.

1.1. Tiling for memory reuse

Like cuDNN, we perform convolutions in rectangular tiles. Each tile consists of a few output pixels (4-48, depending on local memory requirements) and is processed by a single cooperative thread array (CTA). Neighboring output pixels share most of their input pixels - in the case of a 3x3 filter, 2 neighboring output pixels share 6 out of each one's 9 input pixels. Larger tiles lead to better memory reuse and can thereby lower the number of global memory transfers significantly. For example, we use a tile size of 6x6 output pixels for a 3x3 convolutional kernel with a stride of 1. Compared to processing each pixel individually, this reduces the memory reads to less than a fourth, effectively reading less than 2 inputs per output instead of 9. Even more important is the reuse of filter parameters which are often larger than input feature maps. Yet, smaller tile sizes require less registers and allow for more parallelization. And more importantly, smaller tiles are more likely to be skipped as less inputs can require updates. As always, the key is to find the right balance (see Table 1).

1.2. Hybrid dense/sparse tile inference

The convolutional kernel starts with calculating indices, loading the update mask of the input, writing the output mask and, in case any of the inputs was updated, performing the actual convolution. The convolution is performed in three steps: a) load updated input pixels and store them in CTA shared memory and store zero values for inputs which were not updated, b) load filter values and multiply them with the input stored in shared memory, c) write outputs. Steps a) and c) use runtime conditionals against input and output boundaries, update mask and dilation. Step b) uses a highly optimized static code that, independent of the input mask and tile position, always performs all multiply-accumulate operations. Very sparse tiles with four or less active input pixels, making up about 20% of non-empty tiles in our tests, are accelerated in a special mode which can process sparse tiles up to 2x faster. This mode loads only pixels of the filter weights that are required and iterates over an array of active pixels contrary to iterating over all pixels and checking the update flag. The performance impact of adding a runtime conditional to allow for per-pixel sparsity (check each pixel if processing is required) and the performance of our hybrid approach compared to a per-tile sparsity (process all pixels or none) approach are reported in Table 1.

Tile Mode	Tile Size	s=0%	s=50%	s=90%	s=99%
p.t. Sparse	6x6	17.0	16.9	16.5	7.7
	5x5	23.7	23.6	22.9	8.5
Hybrid	6x6	17.0	16.9	15.8	5.0
	5x5	23.5	23.2	18.3	4.5
p.p. Sparse	6x6	60.4	51.9	44.3	20.4
	5x5	49.6	43.6	38.0	14.1

Table 1. GTX 1050 runtime comparison between different tile sizes and sparsity levels (s) reported in milliseconds for a 3x3 convolution with 128 input and output channels and a 256x256 pixels input. Since sparsity is generated uniformly, even with a 90% sparse input is very likely have at least one updated pixel per tile. *Per-tile sparse* mode always processes either all pixels of a tile or none of them. *Per-pixel sparse* mode decides per input pixel whether it needs to be processed or can be skipped. *Hybrid* mode uses dense mode when more than four input pixels are updated and uses a special implementation for very sparse tiles. Larger tile sizes cannot be used due to local memory resource limitations.

It should be noted that depth-wise convolutions require a special implementation to stay competitive against cuDNN. In depth-wise convolutions, every output channel only depends on input values of the corresponding channel in the input feature map. Because of that, input values are used less often and (in our implementation) only by a single thread, removing the necessity of keeping data local to a CTA. When loading input data, the update flag of a pixel must always be checked before reading the values because non-updated pixels contain invalid data. Since the loaded value is only used for up to 9 multiplications in the case of a 3x3 convolution, we fuse loading and multiplication steps described above. Thus, we decide per pixel if we load the value and perform the multiplication – resulting in a per-pixel sparse operation compared to per-tile for standard convolutions. Still, skipping a tile entirely is much faster than processing a single input pixel.

1.3. Memory layout for bandwidth reductions

Using a per-pixel update mask, we can reduce the memory bandwidth during inference greatly in many ways. Compared to previous work that had to compare the inputs of each layer against previous values, we only load values that were marked as updated. Furthermore, using the update mask, we do not have to set unchanged values in the feature map to zero. This allows us to use uninitialized memory for the greater part of the feature maps - only writing valid values for updated pixels. Compared to setting all unchanged output features to zero, this improved the performance of convolutions in HRNet by up to 68%.

PyTorch's default memory layout is $NCHW$, storing one image per pixel channel. Accessing pixels individually, however, is very inefficient with this layout, as the minimum memory access size – a cache line – would always load multiple neighboring pixels per instruction. For efficient use of per-pixel sparsity, we store feature maps in $NHWC$ format. This way, all channels of a pixel are stored coalesced and can be accessed efficiently.

1.4. Floating point number inaccuracies

While delta updates can in theory be applied indefinitely, floating point number operations result in slightly different outcomes when taking large accumulated values or small deltas as input. With HRNet and Human3.6M, we did not experience any problems even with sequences thousands of frames long. However, we did notice small errors accumulating over time in EfficientDet, even with dense inference, *i.e.* using negative thresholds. We recommend to reset the buffers every few hundred frames, or when the network input switches between different videos, to flush all accumulated errors due to floating point inaccuracies.

2. Threshold tuning ablation study

In our evaluations, we used a maximum loss increase target of 3% for the task of threshold tuning. To ensure that we do not exceed this limit, every truncation threshold is only allowed to increase the loss by a maximum of $\frac{3\%}{\#_{layers}}$. Due to a predefined step size in the threshold tuning process, however, the actual loss increase is typically much lower. We evaluated different maximum loss increase targets to compare the resulting accuracy and speedup (see Figure 1). To better show the impact of different parameters, we do not use a high threshold and update mask dilation on the first layer as this already increases sparsity greatly, and thereby reduces the impact of the chosen parameters. Instead, we set the first threshold to a low value of 0.15 manually.

2.1. Training truncation thresholds

We experimented with training truncation thresholds in parallel instead of auto-tuning them one-by-one. Using the update mask density as an additional loss, the trade-off between accuracy and pixel update density could be optimized more accurately and the accuracy reduction per layer could be better balanced. *E.g.* early layers of the CNNs can be al-

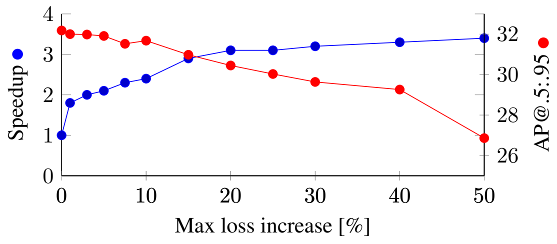

Figure 1. Accuracy and speedup achieved with EfficientDet-d1 on the MOT16 dataset with thresholds tuned on different maximum loss increase targets. Even when using a 30% total loss increase target, the accuracy stays close to dense inference. With a smaller step size for threshold tuning, the accuracy could be kept closer to the maximum loss increase target.

lowed a larger accuracy decrease, because they have a larger impact on the overall update density.

Instead of boolean update masks, we used soft truncation with a sigmoid activation function to allow for gradients to propagate better. With the same intention, we switched to using norm truncation instead of maximum truncation, *i.e.* we compare the L^2 norm of a pixel's *deltas* against a threshold. This way, all values contribute to the update mask and the training process is more stable than with maximum truncation. While we managed to train feasible thresholds with a pre-defined accuracy vs. sparsity trade-off, the resulting thresholds did not outperform tuned thresholds. Furthermore, threshold training takes many times longer to process and hyper parameters are more difficult to tune than the single parameter used for auto tuning.

Dataset	Backend	AP@0.5	AP@.5:.95	GFLOPs	Jetson Nano b=1		GTX 1050 b=1		GTX 1050 b=8/3		RTX 3090 b=1		RTX 3090 b=48/24	
					FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup
MOT16-d0	cuDNN	55.6%	27.8%	5.0	1.4	1.0	26.0	1.0	26.4	1.0	29.3	1.0	368	1.0
	ours dense				4.0	2.9	29.3	1.1	28.6	1.1	57.2	2.0	369	1.0
	ours $\epsilon = \infty$	17.9%	6.6%	-	8.8	6.3	61.0	2.4	340	12.9	57.0	2.0	2389	6.5
	CBInfer	54.3%	27.0%	1.7	2.0	1.4	9.5	0.4	24.3	0.9	15.6	0.5	273	0.7
	ours sparse	55.8%	27.8%	2.8	7.9	5.6	48.2	1.9	76.6	2.9	57.2	2.0	896	2.4
MOT16-d1	cuDNN	63.9%	32.2%	12.2	0.7	1.0	10.5	1.0	11.3	1.0	23.3	1.0	169	1.0
	ours dense				1.8	2.6	11.6	1.1	11.8	1.0	45.0	1.9	167	1.0
	ours $\epsilon = \infty$	21.7%	7.2%	-	6.6	9.4	51.1	4.9	151.6	13.4	44.0	1.9	1032	6.1
	CBInfer	63.5%	31.9%	4.6	1.3	1.9	6.3	0.6	9.4	0.8	11.4	0.5	106	0.6
	ours sparse	64.0%	32.0%	6.8	3.9	5.6	27.4	2.6	30.9	2.7	45.2	1.9	389	2.3
WildTrack-d0	cuDNN	67.1%	35.4%	5.0	1.4	1.0	26.0	1.0	26.4	1.0	29.3	1.0	383	1.0
	ours dense				4.0	2.9	29.3	1.1	28.6	1.1	57.2	2.0	384	1.0
	ours $\epsilon = \infty$	29.5%	14.3%	-	8.8	6.3	61.0	2.4	340	12.9	57.0	2.0	2389	6.5
	CBInfer	65.4%	33.9%	1.1	2.2	1.6	10.4	0.4	25.0	0.9	16.8	0.6	274	0.7
	ours sparse	66.4%	34.7%	2.3	8.1	5.8	56.1	2.2	100	3.8	57.2	2.0	973	2.5
WildTrack-d1	cuDNN	72.1%	41.6%	12.2	0.7	1.0	10.5	1.0	11.3	1.0	23.3	1.0	169	1.0
	ours dense				1.8	2.6	11.6	1.1	11.8	1.0	45.0	1.9	167	1.0
	ours $\epsilon = \infty$	28.3%	15.0%	-	6.6	9.4	51.1	4.9	151	13.4	45.9	2.0	1032	6.1
	CBInfer	71.5%	40.5%	2.4	1.4	2.0	6.8	0.6	10.6	0.9	13.1	0.6	120	0.7
	ours sparse	71.6%	40.8%	5.7	4.8	6.9	32.1	3.1	38.6	3.4	44.8	1.9	415	2.5

Table 2. Speed and accuracy comparisons of different CNN backends used for the task of object detection with batch size b. MOT16 is evaluated only on fixed camera videos. Our evaluations show that DeltaCNN achieves higher frame rate and accuracy using the 12.2 GFLOPs *d1* configuration of EfficientDet than cuDNN achieves with the 5.0 GFLOPs d0 configuration.