Accelerating Sparse DNN Models without Hardware-Support via Tile-Wise Sparsity

2020/11 • Shanghai
Accelerating Sparse DNN Models without Hardware-Support via Tile-Wise Sparsity

Cong Guo¹, Bo Yang Hsueh², Jingwen Leng¹, Yuxian Qiu¹, Yue Guan¹, Zehuan Wang², Xiaoying Jia², Xipeng Li², Yuhao Zhu³ and Minyi Guo¹

¹Shanghai Jiao Tong University, Emerging Parallel Computing Center, REArch (Resilient and Efficient Architecture) group
²NVIDIA, ³University of Rochester
Outline

- Background & Motivation
- Tile-Wise Sparsity
- Efficient GPU Implementation
- Evaluation
**Dense GEMM Accelerator**

GEMM-based accelerators are dominant owing to their wide applicability.

Convolution operations that dominate computer vision models are converted to the GEMM. NLP models are naturally equivalent to the GEMM operation.
DNN Models and Pruning

The DNN models are sparse! Pruning is an effective and promising approach to reduce the DNN latency.

Enormous computation cost and memory usage.

Pruning

Fewer parameters and less computation cost

[Song Han, etc. NIPS’15]
**Sparsity Pattern**

- **Irregular, Random**
  - High Accuracy
  - Low efficiency

- **Balance**

- **Regular, Structured**
  - Low Accuracy
  - High efficiency

**No constraint (1*1 block)**

- **Software:** MKL, cuSparse
- **Hardware:** OuterSPACE, [HPCA’18]
- **SpArch, [HPCA’20]**

**Fixed sparsity of each vector**

- **Software:** Balanced Sparsity [AAAI’19]
- **Hardware:** Bank-Balanced Sparsity [FPGA’19]
- **Sparse Tensor Core [MICRO’19]
- **Tesla A100 [GTC’20]**

**n*n block**

- **Software:** Block-sparse [arXiv’17]
  - 8x8, 16x16(CUDA) 32x32, 64x64(Tensor)

**BW is friendly to dense GEMM accelerator.**
Sparsity Pattern Efficiency

GPU: Tesla V100 32GB
Workload: BERT(MNLI)
Software: TensorFlow 1.15(Fine-tune)
   cuBlas, cuSparse and Block-sparse (Inference)
Accuracy loss < 1%

1. BW achieves the best performance.
2. BW is still $3 \times$ slower than the dense model on the tensor core.
3. They are all Inefficient on the existing dense GEMM hardware.

A new sparsity pattern that can match the existing hardware features while maintaining the fine granularity, which is critical for achieving the high model accuracy.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Core</th>
<th>Library</th>
<th>Speedup</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element Wise</td>
<td>CUDA</td>
<td>cuSparse</td>
<td>0.06x</td>
<td>63%</td>
</tr>
<tr>
<td>Vector Wise</td>
<td>CUDA</td>
<td>cuSparse</td>
<td>0.07x</td>
<td>56%</td>
</tr>
<tr>
<td>Block Wise</td>
<td>Tensor</td>
<td>Block-sparse</td>
<td>0.33x</td>
<td>50%</td>
</tr>
</tbody>
</table>

[Song Han, etc. NIPS’15]  [Block-Sparse, arXiv’17]
[Sparse Tensor Core, Micro’19]  [Balanced Sparsity, AAAI’19]
Outline

1. Background & Motivation
   - Tile-Wise Sparsity

2. Efficient GPU Implementation

3. Evaluation
Algorithm-software Co-designed Tile-Wise Sparsity

**Tile-Wise Sparsity.** An algorithm-software co-designed pruning method that reduces the DNN latency on existing dense architectures while maintaining high accuracy without special hardware support.

**key insight:** a tiling-friendly sparsity pattern

Software: CUTLASS – Tiling GEMM

Hardware: Tesla V100
Tile-Wise Sparsity

GEMM: M, N, K

- $T_x = G = \text{Granularity}$
- $T_y = \text{Tile Length (y)}$
- $C = \text{pruned column}$

The key idea of our tile-wise pattern is to prune each $B_{tile}$ with the regular row and column pruning.

The tiling based GEMM is widely used in the dense GEMM accelerators, such as TPU, not only GPU.
Pruning algorithm

Importance Score

Gradually Pruning

Global Weight Pruning

Apriori Tuning

More details on the paper...

\[ \Delta L(w) = \sqrt{(L(w = w_i) - L(w = 0))^2} \]
\[ L(w = 0) = L(w_i) + \frac{\partial L(w_i)}{\partial w} \cdot w_i + R_i(w = 0) \]
\[ \Delta L(w) \approx \sqrt{\left(\frac{\partial L(w_i)}{\partial w} \cdot w_i\right)^2} \]

Importance Score

[P. Molchanov, etc. CVPR 2019]

Uneven distribution of EW

Global Weight Pruning

Apriori Tuning

Gradually Pruning

[Song Han, etc. NIPS’15]
Outline

Background & Motivation

Tile-Wise Sparsity

- Efficient GPU Implementation

Evaluation
Goal:

Execute TW sparsity on **GPU** (including CUDA core and Tensor core) efficiently.

Three optimizations leveraging GPU’s programming features:

1. Memory accesses coalesce (via memory layout transpose)
2. Kernel reduction (via fusion)
3. Load imbalance mitigation (via concurrent kernel)
Baseline GEMM Tiling

Re-organize

Run-time

Preprocessed

CUTLASS

Run-time

Sparsity in the Global Density in the Core Execute Efficiently!

A[\text{offset}_A + \text{offset}_k[k]]
B[\text{offset}_B]
C[\text{offset}_C + \text{offset}_n[n]]
Memory Accesses Coalesce

Optimization 1

- Column skipping
- Transpose to eliminate Memory uncoalescing
- Row skipping

Efficiency

Performance degradation

Transposed Memory Uncoalescing

Efficiency
Kernel Fusion

Optimization 2

Fused with img2col on CNN
Transpose is free to GPU and TPU
Kernel Fusion

Optimization 2

Fused with Transpose on BERT
Based on NVIDIA Faster Transformer.

Transpose fusion
Optimization 3

**Condensed Tile**

**Multi-Stream**

*Concurrent kernel execution*

Overlap the computation of different tiles by assigning to different streams, and rely on the underlying scheduler to maximize the resource utilization.
Outline

Background & Motivation

Tile-Wise Sparsity

Efficient GPU Implementation

- Evaluation
Methodology

**Hardware:** NVIDIA Tesla V100 32GB GPU

**Sparsity pattern:**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Core</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile Wise (TW)</td>
<td>Tensor-fp16</td>
<td>Tile Sparsity*</td>
</tr>
<tr>
<td></td>
<td>CUDA-fp32</td>
<td></td>
</tr>
<tr>
<td>Block Wise (BW)</td>
<td>Tensor-fp16</td>
<td>Block-sparse</td>
</tr>
<tr>
<td>Element Wise (EW)</td>
<td>CUDA-fp32</td>
<td>cuSparse</td>
</tr>
<tr>
<td>Vector Wise (VW)</td>
<td>CUDA-fp32</td>
<td>cuSparse**</td>
</tr>
</tbody>
</table>

*Based on CUTLASS 1.3
**V100 can not support sparse tensor core.

**DNN models and datasets:**

<table>
<thead>
<tr>
<th>Models</th>
<th>Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Base</td>
<td>GLUE dataset: MNLI, MRPC, SST, CoLA, RTE, QNLI, SQuAD</td>
</tr>
<tr>
<td>VGG-16 (CNN)</td>
<td>ImageNet</td>
</tr>
<tr>
<td>NMT (LSTM)</td>
<td>IWSLT English-Vietnamese dataset</td>
</tr>
</tbody>
</table>

In the rest of this section, we focus on the GEMM execution time unless explicitly mentioned.
Impact of TW Granularity

**Workload:**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Granularity</th>
<th>Critical Sparsity*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>128</td>
<td>0%</td>
</tr>
<tr>
<td>EW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BW</td>
<td>32</td>
<td>~85%</td>
</tr>
<tr>
<td>BW</td>
<td>64</td>
<td>~85%</td>
</tr>
<tr>
<td>TW</td>
<td>64</td>
<td>75%</td>
</tr>
<tr>
<td>TW</td>
<td>128</td>
<td>40%</td>
</tr>
</tbody>
</table>

* With the sparsity, the pruning method starts to outperform the dense model latency. The lower the better.

At the sparsity of 75%, TW-128 has accuracy loss of about 0.9% and 2.4% compared to EW and the baseline dense model at 75% sparsity, respectively. With only 40% sparsity, TW-128 starts to outperform the dense model latency.

BW-64 experiences the most drastic accuracy drop of 4% at 75% sparsity. BW-64 is faster than the dense model only when the sparsity is greater than 85%, which leads to an accuracy loss as high as 10%.

**TW exceeds BW in both of speedup and model accuracy. G=128 is sufficient to maintain the model accuracy while providing significant latency reduction for TW.**
Accuracy

Workload:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>EW</td>
<td>-</td>
</tr>
<tr>
<td>BW</td>
<td>32 * 32</td>
</tr>
<tr>
<td>TW</td>
<td>128</td>
</tr>
<tr>
<td>VW</td>
<td>16</td>
</tr>
</tbody>
</table>

EW the best.
BW the worst.

The accuracy of TW and VW are similar when the sparsity is below 70%. With high sparsity (> 70%), TW generally outperforms the VW with the exception of NMT.
Sparsity Pattern

BERT-base Layer-0 $W_Q$

Irregularity: $EW > TW > VW, BW$
Speedup on GEMM

BERT accuracy loss < 3%
VGG accuracy loss < 1%
NMT BLEU loss < 1

Tensor cores: TW 1.95× speedup
CUDA cores: TW 2.86× speedup

TW achieves the meaningful latency reduction on both tensor cores and CUDA cores owing to its compatibility with dense GEMM, while all other sparsity patterns cause the actual slowdown.

(a) TW vs BW on tensor core.

(b) TW vs VW and EW on CUDA core.
# End-to-end Latency and Impact of Optimizations

Without transpose: **Performance degradation.**

With explicit transpose: **10% overhead. -- Optimization 1**

With fusion transpose: **2% overhead. -- Optimization 2**

End-to-end speedup: **1.61x.**

---

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Dense</th>
<th>W/o Transpose</th>
<th>Explicit Transpose</th>
<th>Fused Transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>32.38 (71%)</td>
<td>37.6</td>
<td>14.29</td>
<td>14.29 (51%)</td>
</tr>
<tr>
<td>Transpose</td>
<td>0</td>
<td>0</td>
<td>5.18</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>45.37</td>
<td>50.59</td>
<td>32.46</td>
<td>28.22</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>0.9</td>
<td>1.4</td>
<td>1.61</td>
</tr>
</tbody>
</table>

GEMM: **2.26x** than Dense and **2.63x** than w/o Transpose

Fusion Transpose: **2%** overhead

Without Fusion Transpose: **~10%** overhead

---

BERT-Base model with 75% sparsity on tensor core

---

Without transpose: **Performance degradation.**
TW achieves the meaningful speedup on both tensor cores (1.95×) and CUDA cores (2.86×) with a high model accuracy, while all other sparsity patterns cause the actual slowdown.

The tiling GEMM algorithm is widely used in the dense GEMM-based accelerators. In other words, supporting TW on other platforms like TPU is feasible.

Proposed a new DNN model sparsity design insight based on the Tile-Wise algorithm-software optimization.

Tile Sparsity is open source!

[https://github.com/clevercool/TileSparsity](https://github.com/clevercool/TileSparsity)
Questions?