Accelerating PyG with Intel CPUs

Ke Ding, Principal Engineer
Intel – SATG AIA
Sep 2022
Outline

• Intel AI Software Stack
• PyG Optimization for Intel CPUs
  ▪ Message Passing Analysis
  ▪ Kernel Optimization: scatter_add
  ▪ Kernel Optimization: spmm_reduce
  ▪ Node Sampling Optimization
• What’s Next
AI Software Ecosystem and Intel Tools

Engineer Data > Create Machine Learning & Deep Learning Models > Deploy

- Container Repository: oneContainer
- MLOps: Cnvr.io
- Developer Sandbox: DevCloud

Accelerate End to End Data Science and AI

Connect AI to Big Data

- Domain Toolkit: NLP, RecSys, TLT, Time Series, PPML
- AI Analytics Toolkit: BigDL (Analytics Zoo)

Data Analytics at Scale

- MODIN
- NumPy
- pandas
- SciPy
- omnisci
- Numba

Optimized Frameworks and Middleware

- TensorFlow
- PyTorch
- PyG
- ONNX
- PaddlePaddle
- mxnet
- ONNX
- scikit-learn
- dmlc
- XGBoost
- DeepSpeed

Optimize and Deploy Models

- Automate Model Tuning AutoML: SigOpt
- Write Once Deploy Anywhere: OpenVINO Toolkit
- Automate Low-Precision Optimization: Neural Compressor (INC)

- oneDAL
- oneDNN
- oneCCL
- oneMKL

* Other names and brands may be claimed as the property of others.
PyG Overview on Intel Platforms

• Open source upstream first
• Inference and training
• Abstract perf primitives into oneDNN

PyG: Models, Operators, Storage

PyTorch
• scatter_add
• index_select
• spmm_reduce
• sort
• index
• …

torch-scatter
torch-sparse
torch-cluster
pyg-lib

oneDNN
oneCCL
Message Passing Paradigm - GAS

- Graph Conv (GCN)
- Graph Attention (GAT)
- SAGEConv
- GINConv
- EdgeConv
- PNACConv
- RGCN
Message Passing Profiling

**Case I: EdgeIndex in COO.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Self CPU %</th>
<th>Self CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>aten::scatter_add</td>
<td>49.00%</td>
<td>37.797s</td>
</tr>
<tr>
<td>aten::index_select</td>
<td>19.74%</td>
<td>15.223s</td>
</tr>
<tr>
<td>aten::linear</td>
<td>0.01%</td>
<td>5.706ms</td>
</tr>
<tr>
<td>aten::addmm</td>
<td>6.62%</td>
<td>5.108s</td>
</tr>
<tr>
<td>aten::matmul</td>
<td>0.00%</td>
<td>2.339ms</td>
</tr>
<tr>
<td>aten::mm</td>
<td>7.09%</td>
<td>5.472s</td>
</tr>
<tr>
<td>aten::index</td>
<td>5.89%</td>
<td>4.544s</td>
</tr>
<tr>
<td>aten::fill_</td>
<td>3.59%</td>
<td>2.768s</td>
</tr>
<tr>
<td>aten::zeros</td>
<td>0.01%</td>
<td>7.616ms</td>
</tr>
<tr>
<td>aten::zero_</td>
<td>0.00%</td>
<td>2.728ms</td>
</tr>
<tr>
<td>aten::true_divide_</td>
<td>0.00%</td>
<td>1.158ms</td>
</tr>
<tr>
<td>aten::div_</td>
<td>2.74%</td>
<td>2.116s</td>
</tr>
<tr>
<td>aten::add_</td>
<td>1.36%</td>
<td>1.046s</td>
</tr>
<tr>
<td>aten::copy_</td>
<td>1.30%</td>
<td>1.005s</td>
</tr>
</tbody>
</table>

**Profiling of SAGE+Reddit**

**Case II: EdgeIndex in CSR.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Self CPU %</th>
<th>Self CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch_sparse::spmm_sum</td>
<td>97.09%</td>
<td>56.086s</td>
</tr>
<tr>
<td>aten::linear</td>
<td>0.00%</td>
<td>85.000us</td>
</tr>
<tr>
<td>aten::matmul</td>
<td>0.00%</td>
<td>57.000us</td>
</tr>
<tr>
<td>aten::mm</td>
<td>1.38%</td>
<td>795.201ms</td>
</tr>
<tr>
<td>aten::relu</td>
<td>0.00%</td>
<td>50.000us</td>
</tr>
<tr>
<td>aten::clamp_min</td>
<td>0.76%</td>
<td>440.384ms</td>
</tr>
<tr>
<td>aten::add_</td>
<td>0.57%</td>
<td>327.801ms</td>
</tr>
<tr>
<td>aten::log_softmax</td>
<td>0.00%</td>
<td>23.000us</td>
</tr>
<tr>
<td>aten::_log_softmax</td>
<td>0.10%</td>
<td>55.480ms</td>
</tr>
<tr>
<td>aten::argmax</td>
<td>0.09%</td>
<td>53.149ms</td>
</tr>
<tr>
<td>aten::index</td>
<td>0.01%</td>
<td>5.771ms</td>
</tr>
<tr>
<td>aten::empty</td>
<td>0.00%</td>
<td>1.088ms</td>
</tr>
<tr>
<td>aten::t</td>
<td>0.00%</td>
<td>68.000us</td>
</tr>
<tr>
<td>aten::detach</td>
<td>0.00%</td>
<td>65.000us</td>
</tr>
</tbody>
</table>

**Profiling of GCN+ogbn-products**

* Single batch inference (training hotspot slightly different).
Kernel Optimization I: scatter_add

- `scatter_add` is hotspot when `EdgeIndex` stored in COO.
- It adds all values from the tensor `src` into `self` at the indices specified in the `index` tensor.

**Scatter_add Analysis:**
- Memory bandwidth bound.
- M refers to `num_nodes`, N refers to `num_edges`, K refers to `num_features`.
- Input shape might be very large: for example, in `SAGE+Reddit`, M = 135K, N = 447K, K = 256.
- Possible *write conflicts* since multiple threads may attempt to write the same address simultaneously.
Kernel Optimization I: scatter_add

- `scatter_add` is hotspot when `EdgeIndex` stored in COO.
- It adds all values from the tensor `src` into `self` at the indices specified in the `index` tensor.

**Scatter_add Optimization:**

- **Good performance**: parallel on outer dimension (M or N) and vectorize on inner dimension (K).
- Solve write conflicts though index sorting via radix sort.
- **SAGE+Reddit**: single socket inference:
  i. `scatter_add` time: 5.9x speedup.
  ii. end to end time: 1.7x speedup.
- The algorithm is equivalent to:
  i. convert COO to CSR (sorted indices encoded on CSR format);
  ii. do `spmm_reduce`. 

![Diagram](image)
Kernel Optimization II: spmm_reduce

- *spmm_reduce* is hotspot when *EdgeIndex* stored in CSR.
- API definition similar to SpMM, except that more reduction type required “max”, “mean”.

**Spmm Reduce Optimization:**
- Memory bandwidth bound.
- Reduce type: “sum”, “max”, “mean”.
- M and N refers to *num_nodes*, nnz refers to *num_edges*, K refers to *num_features*.
- Input shape might be very large: for example, in *GCN+ogbn-products*:
  1. *num_nodes*: 2.4M
  2. *num_edges*: 126M
  3. *num_features*: 256
Kernel Optimization II: spmm_reduce

- **GCN+ogbn-products** single socket inference got 4.3x speedup (spmm_sum improved by 4.7x)
Kernel Optimization II: spmm_reduce

- **GCN+ogbn-products** single socket inference got 4.3x speedup (spmm_sum improved by 4.7x)

![Diagram showing performance improvements](image)

**spmm_sum in GCN+ogbn-products**
Kernel Optimization II: spmm_reduce

- **GCN+ogbn-products** single socket inference got **4.3x** speedup (spmm_sum improved by **4.7x**)

![Diagram showing time in seconds for different optimizations]

- **OOB**
  - Time: 56.09 s
- vectorization
  - Time: 29.31 s
- unrolling
  - Time: 25.66 s
- **blocking**
  - Time: 21.95 s (2.6x improvement)
- balanced partition
  - Time: 11.83 s

**spmm_sum in GCN+ogbn-products**
Kernel Optimization II: spmm_reduce

- **GCN+ogbn-products** single socket inference got 4.3x speedup (spmm_sum improved by 4.7x)

---

**Balanced Thread Partition**
- Length of each row refers to number of connections for each node.
- Directly parallel on rows would lead to thread payload unbalance.
- Can’t assume each node has the same number of connections.

---

![Graph showing speedup improvements.](image-url)
Node Sampling

Larger graphs require node sampling and graph partitioning.

1. Sampling neighborhood \( (k=1,2) \)
2. Aggregate features from neighbors
Node Sampling

- PyG has implemented commonly used samplers at `torch_geometric.loader` such as NeighborLoader, HGTLoader, RandomNodeSampler, etc.

- Concept of Loader is a combination of PyTorch's DataLoader and a specific sampler, which handles data sampling and transformation. On some workloads, loader itself might be major performance hotspot.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Self CPU %</th>
<th>Self CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MultiProcessingDataLoaderIter...</td>
<td>71.27%</td>
<td>608.842s</td>
</tr>
<tr>
<td>torch_sparse::spmm_mean</td>
<td>14.91%</td>
<td>127.390s</td>
</tr>
<tr>
<td>aten::addmm</td>
<td>3.77%</td>
<td>32.166s</td>
</tr>
<tr>
<td>aten::copy_</td>
<td>3.60%</td>
<td>30.766s</td>
</tr>
<tr>
<td>aten::mm</td>
<td>2.29%</td>
<td>19.588s</td>
</tr>
<tr>
<td>aten::native_batch_norm</td>
<td>0.94%</td>
<td>7.989s</td>
</tr>
<tr>
<td>aten::add_</td>
<td>0.57%</td>
<td>4.877s</td>
</tr>
<tr>
<td>aten::clamp_min</td>
<td>0.28%</td>
<td>2.429s</td>
</tr>
<tr>
<td>aten::empty</td>
<td>0.23%</td>
<td>1.935s</td>
</tr>
<tr>
<td>torch_sparse::ptr2ind</td>
<td>0.19%</td>
<td>1.661s</td>
</tr>
<tr>
<td>aten::to</td>
<td>0.18%</td>
<td>1.550s</td>
</tr>
</tbody>
</table>

Profiling of GraphSAGE+mag240m

- In the example of GraphSAGE + mag240m, the DataLoader is major performance hotspot.
- Multi-process data loading is enabled if user set `num_workers > 0`, and the sampling from each worker will be sequential.
- Multi-process worker + sequential sampler be not optimal.
Node Sampling

- PyG has implemented commonly used samplers at `torch_geometric.loader` such as NeighborLoader, HGTLoader, RandomNodeSampler, etc.

- Concept of Loader is a combination of PyTorch's DataLoader and a specific sampler, which handles data sampling and transformation. On some workloads, loader itself might be major performance hotspot.

```
def train_dataloader(self):
    return NeighborSampler(self.adj_t, node_idx=self.train_idx,
                            sizes=self.sizes, return_e_id=False,
                            transform=self.convert_batch,
                            batch_size=self.batch_size, shuffle=True,
                            num_workers=4)
```

```
def convert_batch(self, batch_size, n_id, adj):
    if self.in_memory:
        x = self.x[n_id].to(torch.float)
    else:
        x = self.x[n_id].to(torch.float)
        y = self.y[n_id][:batch_size].to(torch.long)
    return Batch(x=x, y=y, adj_s_t=[adj_t for adj_t in adj]
```

Profiling of GraphSAGE+mag240m

48.6% time spent on this circled line since only 4 cores used!
Node Sampling

- PyG has implemented commonly used samplers at `torch_geometric.loader` such as NeighborLoader, HGTLoader, RandomNodeSampler, etc.

- Concept of Loader is a combination of PyTorch's DataLoader and a specific sampler, which handles data sampling and transformation. On some workloads, loader itself might be a major performance hotspot.

### Optimize data loader on CPU:

- Make sure sampling and transformation can be properly paralleled.
- Set CPU affinity if the multi-process data loader is used.
- Fuse multiple operators from `convert_batch` in C++ kernel.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Self CPU %</th>
<th>Self CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MultiProcessingDataLoaderIter...</td>
<td>71.27%</td>
<td>608.842s</td>
</tr>
<tr>
<td>torch_sparse::spmm_mean</td>
<td>14.91%</td>
<td>127.390s</td>
</tr>
<tr>
<td>aten::addmm</td>
<td>3.77%</td>
<td>32.166s</td>
</tr>
<tr>
<td>aten::copy_</td>
<td>3.60%</td>
<td>30.766s</td>
</tr>
<tr>
<td>aten::mm</td>
<td>2.29%</td>
<td>19.588s</td>
</tr>
<tr>
<td>aten::native_batch_norm</td>
<td>0.94%</td>
<td>7.989s</td>
</tr>
<tr>
<td>aten::add</td>
<td>0.57%</td>
<td>4.877s</td>
</tr>
<tr>
<td>aten::clamp_min</td>
<td>0.28%</td>
<td>2.429s</td>
</tr>
<tr>
<td>aten::empty</td>
<td>0.23%</td>
<td>1.935s</td>
</tr>
<tr>
<td>torch_sparse::ptr2ind</td>
<td>0.19%</td>
<td>1.661s</td>
</tr>
<tr>
<td>aten::to</td>
<td>0.18%</td>
<td>1.550s</td>
</tr>
</tbody>
</table>

*Profiling of GraphSAGE+mag240m*
What’s Next

• Fully Optimize PyG for both inference and training for Intel platforms
• Large scale distributed GNNs for top use cases
• Unified Graph Platform and workflows supporting query, analytics and GNNs
Questions?