WACO with Meta-Learning

Cecilia Nyberg

December 2024

1 Problem Statement and Motivation

Sparse matrix operations are critical components in scientific computing, machine learning, and engineering applications. The optimization of these operations can make a big difference in program performance. An important factor for optimizing these operations is choosing the right storage format for the specific matrix [4]. The optimal storage format depends on the characteristics of the matrix, and over the past few years, machine-learning approaches have been used to optimize these storage selections for space matrices. One such approach is WACO, a method for optimizing both the storage and schedule in a sparse tensor program for specific sparse matrix characteristics.

The WACO model has shown good performance in speeding up runtime for space matrix operations [4], but it also comes with limitations. The WACO model is higly hardware dependent and performs significantly better on the machine it has been trained on. This stems from the fact that labels regarding storage and schedule optimization of sparse matrices is hardware dependent. Different storage formats excel for different machines. Since the data collection is time-consuming (it took two weeks to collect data for the WACO model) this is an important shortage since it highly limits the practical usability of the WACO model. This report aims to analyze how integrating meta-learning into the training loop can improve the generalizability of the WACO model across different hardware.

1.1 Aim

From the problem statement the following aim can be specified for this report:

1. Achieve a more hardware-independent optimization of sparse matrix operations by using meta-learning.

2 Existing Work Focused on Meta-Learning

Meta-learning, often described as ”learning to learn,” aims to train models capable of quickly adapting to new tasks by using experience from prior tasks [2]. Instead of training for a single problem, meta-learning involves a task distribution where the model iteratively learns generalizable features and initialization parameters that can quickly adapt to new tasks. A key challenge arises when these tasks need to be executed on hardware with limited computational resources, such as edge devices. [2] propose a hardware-aware meta-learning framework where hardware constraints, such as quantized training precision, are incorporated upfront during the meta-learning process. By performing meta-learning in a hardware-constrained environment, the resulting model initialization is optimized for low-resource adaptation, enabling more efficient fine-tuning for new tasks. This technique reduces the computational overhead of fine-tuning while maintaining high task performance.

[3] frames few-shot learning as an optimization problem where models are trained to adapt quickly to new tasks using limited data. This is achieved by optimizing not just for task-specific performance but for generalization across multiple tasks. The authors emphasize a two-level learning process: a task-specific inner loop, where models learn quickly from few examples, and a task-general outer loop, which optimizes the initialization or update rules for broad task adaptability. By treating task-specific updates as an optimization procedure, the model becomes capable of learning to learn. The paper draws parallels between few-shot learning and optimization-based approaches, where prior knowledge is encoded into model parameters to ensure efficient adaptation.

The MAML algorithm presents a model-agnostic meta-learning approach designed to enable neural networks to rapidly adapt to new tasks with minimal fine-tuning [1]. MAML operates by learning a general initialization of model parameters such that a few gradient steps on a new task result in good performance. This is achieved through a two-level optimization process:

1. Task-specific inner loop: The model adapts to a specific task by performing gradient descent updates using a small amount of task-specific data.
2. Meta-level outer loop: The model parameters are optimized across a distribution of tasks to ensure that the inner-loop adaptation is efficient and generalizable.

3 High-Level Overview

Figure 1 and figure 2 represent flow charts for different phases of the implemented algorithm. Figure 1 show the steps for training the base meta-model. The three first steps involve data collection and is performed on the school server specific CPU architecture, as specified in the upper part of the figure. After the three first steps, the leftmost flow shows the meta-model being trained on collected data, builds the KNN graph and finds top 20 candidate SuperSchedules for each matrix. Then the models performance is tested by measuring runtimes for model selected formats and fixed CSR formats. The right flow shows the same steps but using the pre-trained WACO model instead of training a meta-model. The purpose of the right flow is to provide comparison of the performances.

Figure 2 shows the flow for testing the meta-models ability to adapt to a new task. The pink left part of the figure represents steps being performed on a different machine, with CPU configurations as specified in the top of the figure. The collected data from the first three steps is transferred to the first machine and the same approach is taken for testing the model as in figure 1. A difference is though that instead of training the meta-model, the already trained base meta-model is fine-tuned on only 100 matrices collected on the new machine. For the new task, only the inner loop is used to adapt the model parameters.

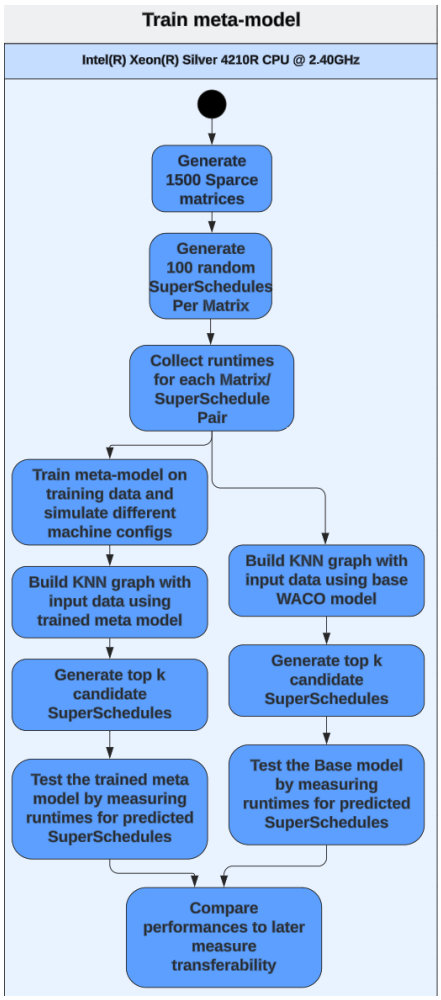

Figure 1: Flow chart of training meta model

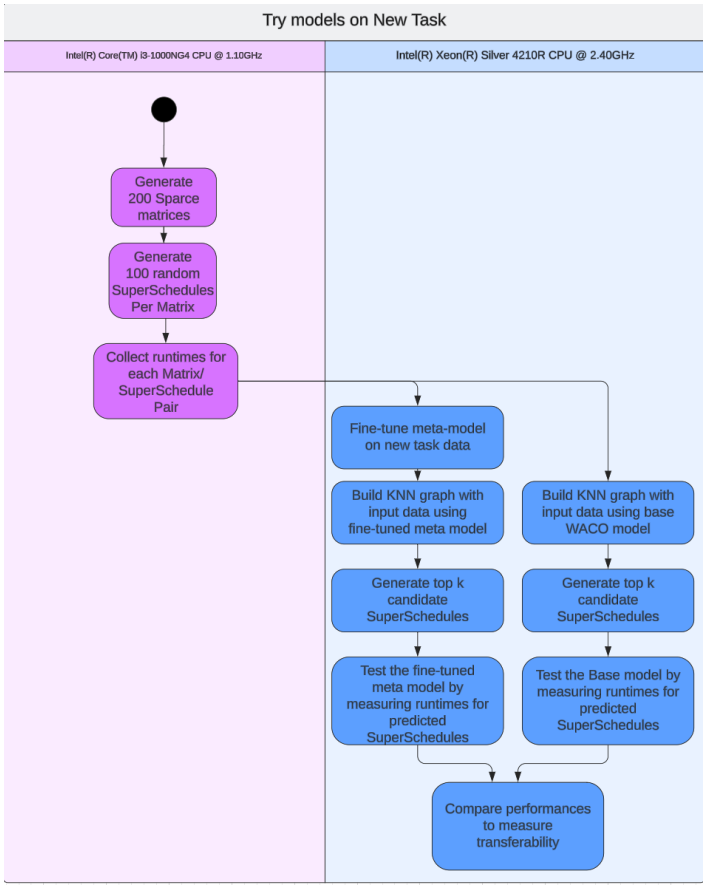

Figure 2: Flow chart of fine-tuning and performing generalizability test

Figrue 3 shows the overview of the project structure. In this project, only SpMM was used. The gen.txt in the training data represents the data collected on a different CPU.

Figure 3: Overview on Project

Limitations

- **Data:** Simulated Sparce matrices were used. These had limited characteristics and do not completly reflect real worl examples.
- **Simulation Accuracy:** The simulated runtimes are deeply simplied hardware behaviors and do not reflect real-world behaviors. Because of time limitation, it was not possible to collect data from different tasks. TO still get a multitask training, this simplificaiton was made manly to show how the training is intended to look.
- **Task specific updates** The model performs only a single gradient step in the task-specific inner loop. This is a simple version of meta learning, and the result could potentially be improved if several gradient steps were performed.

4 Implementation Details

In the context of sparse matrix operations, meta-learning can be employed to achieve higher hardware independence: each hardware platform represents a task with different data characteristics. Hardware platforms have distinct constraints, such as memory bandwidth and latency. By training across multiple hardware-dependent tasks, the model learns an initialization that generalizes well, which minimizes the need for extensive retraining when adapting to a new hardware environment.

Figure 4 shows an overview of the meta-learning training loop implemented in this project. The implementation focuses on meta-learning for optimizing sparse matrix format and scheduling, where the goal is to predict the best matrix-SuperSchedule pair for minimizing runtime on hardware-dependent tasks. The system relies on a pre-trained ResNet14-based model with MinkowskiEngine to process sparse matrices, while a meta-learning integration is integrated to enable task-specific adaptation through an inner loop and generalization across tasks via an outer loop. The goal with this approach is that the model should be able to adapt to different machine configurations with minimal task-specific data. The reason why the pre-trained ResNet14-based model is used as a starting point is because the data is similar to the one used in WACO, both respresents sparce matrices, and it can therefore be benefical to initalize the weights to these pre-trained values instead of random ones.

The tasks for meta-learning are defined as different hardware configurations. The initial idea was to use only one hardware as a task to train the base meta-learning model on, but since meta-learning is primarly benecial for multiple task-training, it was decided to try to integrate several tasks. The best solution would have been to include collected data from different hardware and divide these into different tasks to train the model on. However,

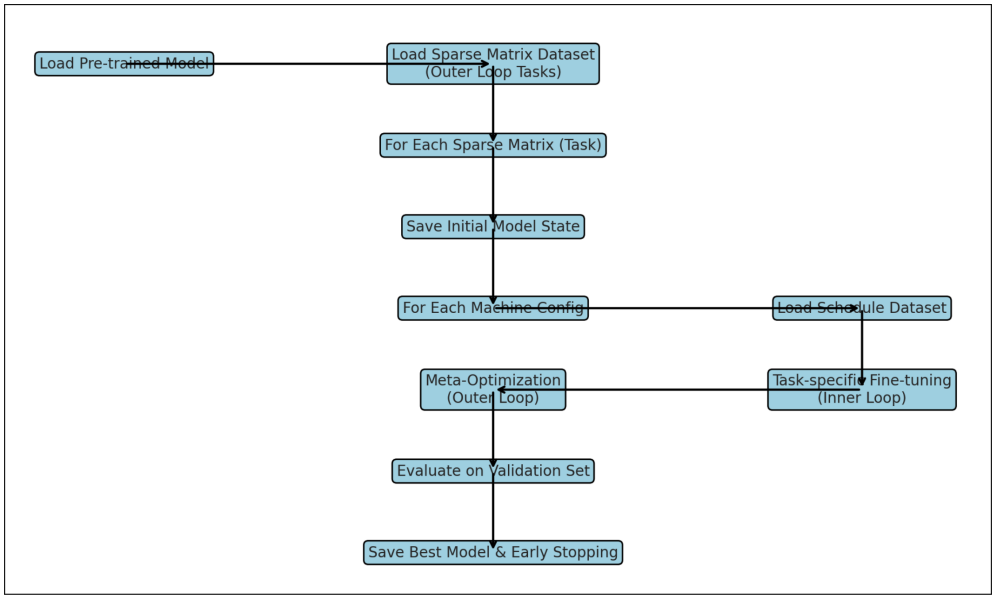

Figure 4: Overview of meta-learning training loop

because of time-limitations this was not possible. Instead different hardware configurations were simulated and the runtimes were slightly changed depending on simulated hardware specifics. This was done in a highly simplified way. The purpose was mostly to include this logic in the training loop, and for future research it could be advanced and improved by using real tasks or better informed simulations. The simulated machines are parameterized by the number of cores, memory bandwidth, and latency. The simulated runtimes for task-specific SuperSchedules are computed using a simplified formula that incorporates these machine parameters:

$$\text{Simulated Runtime} = \frac{\text{Actual Runtime}}{\text{Cores} \times \text{Bandwidth}} \times \text{Latency}.$$

The goal with this runtime approximation is to introduce variability to help the model learn relationships between matrix structure, schedules, and hardware configurations while avoiding the need for real runtime measurements, although it is highly simplified. The actual runtime used for the simulated runtimes calculations is collected on the school server CPU. The CPU specifics can be seen in figure 1.

Sparse matrices are represented as `MinkowskiEngine.SparseTensor`, which handles sparse data using coordinates and features of the non-zero elements.

The core functionality of the meta-training revolves around two loops: an inner loop for task-specific fine-tuning and an outer loop for meta-optimization. In the inner loop, the model is fine-tuned for a specific sparse matrix and machine configuration. Before fine-tuning, the model parameters are reset to a saved initial state to keep task independence. The fine-tuning process involves a single gradient step using the pairwise ranking loss:

$$\mathcal{L}_{\text{rank}} = \max(0, 1 - \text{sign}(y_i - y_j) \cdot (f(x_i) - f(x_j))),$$

where y are the true runtimes and $f(x)$ are the predicted runtimes. The ranking loss ensures that schedules with shorter simulated runtimes are assigned lower predictions, improving the model's ability to identify the best-performing schedules.

The outer loop performs meta-optimization by updating the model parameters across multiple tasks. Once task-specific fine-tuning is completed for all machine configurations associated with a sparse matrix, the outer loop aggregates the learning from all tasks and updates the model using a meta-optimizer. This step ensures that the model generalizes well across tasks, capturing shared patterns across sparse matrices and machine configurations.

At the end of the meta-training process, the model can be fine-tuned on new tasks with limited data. The meta-trained model should provide a beneficial initialization that requires minimal data and gradient updates to adapt to new hardware configurations.

The overall workflow can be summarized as follows: the outer loop iterates over sparse matrices, where each matrix represents a task. For each task, the inner loop fine-tunes the model on simulated schedules for different machine configurations. The task-specific losses are then used to update the model in the outer loop. Validation is performed at the end of each epoch, and early stopping is used to save the best-performing model.

The implementation has certain limitations. The use of simulated runtimes does not capture the nuances of real hardware behavior. Additionally, it was not possible to integrate the existing learn2learns MAML library because of miss-matches in prefix usage for MinkowskiEngine and learn2learns maml implementation. Therefore, the meta-logic had to be manually implemented. By using a pre-build library, a more complex solution could have been possible, such as using mutliple gradient steps.

In summary, the meta-learning framework provides an solution to more hardware-independent sparse matrix optimization by combining task-specific fine-tuning and generalization across tasks.

The figure below showcases an example flow through the meta-training loop.

Example Flow: Sparse Matrix Scheduling on Hardware-Dependent Formats

Input:

- **Sparse Matrix:** A matrix $matrix1$ with 1000 non-zero elements representing a specific computational task.
- **Machine Configurations:**
  - **Machine A:** 8 cores, 50 GB/s memory bandwidth, latency 1.2.
  - **Machine B:** 16 cores, 80 GB/s memory bandwidth, latency 1.0.
- **Schedules:**
  - **Schedule 1:** Runtime ~ 120 ms on Machine A.
  - **Schedule 2:** Runtime ~ 95 ms on Machine A.

Workflow:

1. **Sparse Matrix Embedding:** A `ResNet14`-based model, incorporating `MinkowskiEngine` layers, embeds the sparse matrix into a learned feature representation.
2. **Simulated Runtimes:** Runtimes are computed for each schedule and machine configuration using:

$$\text{Simulated Runtime} = \frac{\text{Actual Runtime}}{\text{Cores} \times \text{Bandwidth}} \times \text{Latency}.$$

   Example: For Schedule 1 on Machine A:

$$\frac{120}{8 \times 50} \times 1.2 \approx 0.36\,\text{ms}.$$

3. **Task-Specific Inner-Loop:** For each machine configuration:
   - Re-set the model to its initial state.
   - Fine-tune using a pairwise ranking loss:

$$\mathcal{L}_{\text{rank}} = \max(0, 1 - \text{sign}(y_i - y_j) \cdot (f(x_i) - f(x_j))).$$

   - Perform a single gradient update to improve predictions specific to that machine.
4. **Meta-Learning (Outer Loop):** After fine-tuning, the meta-model is updated to improve generalization across all tasks and hardware configurations.

Output:

- The model predicts runtimes for each schedule and ranks them to select the optimal configuration.
- The meta-model should learn to adapt to new tasks with minimal re-training.

Capabilities:

- **Hardware-Aware Optimization:** Selects the best matrix-schedule pair for each hardware configuration.
- **Efficient Fine-Tuning:** Adapts quickly to task-specific needs with only a single gradient step.
- **Generalization:** The goal is to increase performance on unseen tasks and configurations.

Figure 5: Detailed Workflow for Sparse Matrix Scheduling using Meta-Learning.

5 Evaluation

To evaluate the performance of the meta-trained model, the model was fine-tuned on a small amount of data for a new task (labels from a different hardware). The different hardware CPU was also from intel, but with much lower capacity. The CPU specifics can be seen in table 1, where the left CPU is on the school server and the right one is on a local computer.

CPU for collecting training labels	**Task specific CPU**
Intel(R) Xeon(R) Silver 4210R CPU 2.40 GHz	Intel(R) Core(TM) i3-1000NG4 CPU 1.10 GHz

Table 1: Training CPU and new task CPU

The testing is conducted by using the top 20 SuperSchedules for each test matrix found by the model that is the test subject. Runtimes are collected for operations on matrices with the SuperSchedules the model has predicted, and with a fixed CSR for comparison.

Four different test were conducted:

1. The pre-trained WACO model on data collected on school server CPU
2. The meta-trained model on data collected on school server CPU
3. The pre-trained WACO model on data collected on local computer CPU
4. The fine-tuned meta-trained model on data collected on local computer CPU

The speedups of the first two tests were compared, as well as the speedups of the third and fourth test. These two relative speedups were then compared, to measure the gained transferability. The reason for this was that the simulated matrices differ from the real-world matrices that WACO is trained on. Therefore, it is reasonable that the meta-trained model performs better than WACO on the new CPU since the data is more similar to the original training data. To still see the effect of the learnt generalization, focus was on comparing the relative speedup of the meta-model compared to original WACO for the first task (the school server collected data) and second task (locally collected data).

For testing the models, 100 test matrices were used. It took about 1.5 hour to test each model on 100 matrices. In total, running the tests took 4*1.5 hours = 6 hours.

5.1 Results

Table 2 shows the mean relative speedup of the model-selected SuperSchedule and a fixed CSR format. It can be seen that the Meta model achieves notably higher speedups than the WACO model. However, an influencing reason for this is that the data that the original WACO model was trained on consists of real-world matrices with a bigger range of sizes and some different characteristics than the simulated ones. These changes in the data characteristics makes it expected that the meta model should perform better than the WACO model. Additionally, when tested on the school server collected data, the meta model should perform better since the labels for the test data were collected on the same machine that the meta model had been trained on. The results show that this is indeed the case, having the meta model achieving mean percentage speedup of 5.54% while the WACO model only sees a speedup of 0.52%.

The two last columns in table 2 shows the speedups for data collected on a different machine for the base WACO model and the fine-tuned meta model. The meta-model had been fine-tuned on only 100 task specific data samples and for 5 epochs. Still, the fine-tuned meta model performed much better than the Base WACO model. Both the WACO model and the fine-tuned meta model achieved higher speedups on this new task than for the first task. A possible reason for this was that the local CPU was much slower than the one on the school server. Therefore, choosing the right matrix-SuperSchedule pair can have a higher impact on the runtime. Still, there is a noticeable improvement in runtime for the fine-tuned meta model.

Figure 6 and figure 7 show the numerical speedups for each model. Figure 6 show speedups for school server collected data (task 1) and figure 7 show the speedups for the local machine collected data (task 2). For both types of data, the speedups follow similar patterns. The Meta-trained model seem to more consistently achieve speedups for all matrices, while the WACO base model some times finds very unefficent SuperSchedules which results in runtimes much slower than the fixed CSR. Overall, the WACO base model has a higher number of negative speedups than the meta based models. These results suggests that the meta-model might be better at successfully adapting to the new task, meaning the data collected on the new machine.

Model	Data	Mean relative Speedup over fixedCSR
Base WACO	School server data	0.52%
Base Meta	School server data	5.54%
Base WACO	Local computer data	6.93%
Fine-tuned Meta	Local computer data	19.6%

Table 2: Training CPU and new task CPU

For future research, the experiments should be conducted on real-world matrices instead of smaller simulated ones. Additionally, instead of simulating hardware configurations as different tasks, data should be collected on a few different CPU architectures which would represent the different tasks, enabling real multitask meta training.

Furthermore, the transferability of the model was measured on the same CPU architecture type (Intel), although the machine configurations differed. Testing the model on a different architecture type could give a more realistic indication of the models transferability ability.

It would also be interesting to analyze the matrices were the WACO model choose very disadvantageous SuperSchedules, to get an idea of the shortcomings of WACO. These disadvantageous ones assumably highly impacted the resulting speedup for the WACO base model.

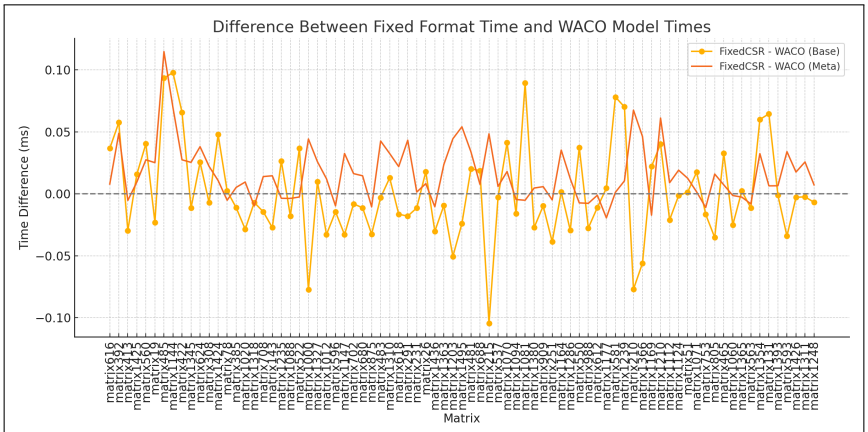

Figure 6: Speedups for Base WACO and Base Meta on school server data

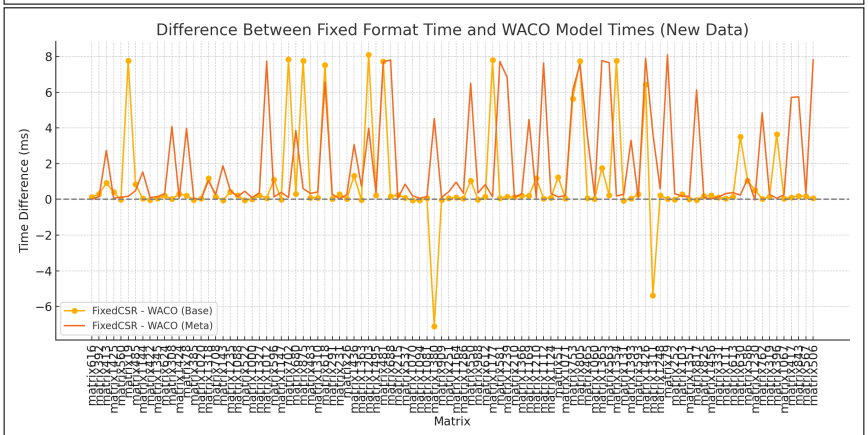

Figure 7: Speedups for Base WACO and FT-Meta on local machine data

6 Appendix

The code can be found here: GitHub Repository
Or by copying this link: https://github.com/cecnyb/WACO_Meta_Learning_SpMM.git

6.1 Instructions for Running the Code

6.1.1 Installment instructions

1. **Clone the Git Repository:**
   - Clone the repository and set the environment variable `WACO_HOME`:

```
git clone https://github.com/cecnyb/WACO_Meta_Learning_SpMM.git
cd WACO_Meta_Learning_SpMM
export WACO_HOME=`pwd`
```

2. **Install Minkowski Engine:**
   - This can be tricky; the following steps worked in a clean Conda environment:

```
conda create -n WACO python=3.10
conda activate WACO
conda install openblas-devel -c anaconda
conda install pytorch=1.12.0 torchvision torchaudio cudatoolkit=11.6 -c
    pytorch -c nvidia
conda install "setuptools <65"
pip install -v --no-cache-dir git+https://github.com/NVIDIA/MinkowskiEngine
```

   - *Note:* Using `conda install "setuptools <65"` solved installation issues.

3. **Install HNSWLib with Python Binding:**
   - Navigate to the `hnswlib` directory and install:

```
cd $WACO_HOME/hnswlib
pip install .
```

4. **Install and Build TACO:**
   - Build TACO by running the following commands:

```
cd $WACO_HOME/code_generator/taco
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j8
```

5. **Build the Code Generator:**
   - Navigate to the `code_generator` directory and compile:

```
cd $WACO_HOME/code_generator
make gcc  # ICC would be better
```

6.1.2 Data collection

1. **Run the Data Collection Pipeline:**
   - Execute the following script to collect the data:

```
$WACO_HOME/dataset/simulated_data/code/data_collection_pipeline.py
```

2. **Specify Matrix Parameters:**

   - To set parameters such as maximum rows, maximum columns, and the number of matrices, modify the following file:

```
$WACO_HOME/dataset/simulated_data/code/simulate_matrices.py
```

3. **Matrix Simulation and Storage:**

   - The script will simulate sparse matrices and store them in the following directory:

```
$WACO_HOME/dataset/simulated_data/simulated_matrices
```

4. **SuperSchedule Generation:**

   - The script will generate 100 SuperSchedule candidates for each matrix and store them in this directory:

```
$WACO_HOME/WACO/training_data_generator/config/
```

   - Additionally, it will write the names of the matrices into the following list:

```
/home/s0/ml4sys16/project1/Workload-Aware-Co-Optimization/WACO/SpMM/
    TrainingData/total.txt
```

5. **Generate Runtimes for Matrices and SuperSchedules:**

   - The script will compute runtimes for matrix and SuperSchedule pairs to determine the best one. The results will be stored here:

```
$WACO_HOME/WACO/SpMM/TrainingData/CollectedData/
```

6.1.3 Training the model

To train the meta model, execute the following script:

```
$WACO_HOME/WACO/SpMM/train_meta_model.py
```

6.1.4 Building the KNN graph and testing the trained cost mdoel

1. **Modify Configuration File:**

   - Open the file:

```
$WACO_HOME/WACO/SpMM/build_hnswindex.py
```

   - Change the file to specify the matrices you want to use (e.g., `test` or `total` to include all matrices).
   - Update the loaded model to point to the desired pre-trained model. We used restnet.phd and resnet_best_model.phd (the meta one).

2. **Build the HNSW Index:**

   - Navigate to the appropriate directory:

```
cd $WACO_HOME/WACO/SpMM
```

   - Run the script:

```
python $WACO_HOME/WACO/SpMM/build_hnswindex.py
```

   - *Note:* There is an error in the official WACO GitHub instructions stating to run build_hnsw.py. Additionally, you need to manually install `hnswi` using pip:

```
pip install hnswi
```

3. **Top-K Search for Sparse Matrices:**

   - Run the following commands:

```
cd $WACO_HOME/WACO/SpMM
python topk_search.py
cd topk
```

   - This searches for the Top-K matrices for a given sparse matrix. You can specify which matrices to use (e.g., `test` or `total`) in `topk_search.py`.

4. **Test Performance:**

   - To test the performance on a single matrix and compare it with a fixed CSR, run:

```
./spmm ../dataset/simulated_data/simulated_matrices/{matrix}.csr
$WACO_HOME/WACO/SpMM/topk/{matrix}.txt
```

   - For example:

```
./spmm ../dataset/simulated_data/simulated_matrices/matrix27.csr
$WACO_HOME/WACO/SpMM/topk/matrix27.txt
```

   - To test performance on all matrices in the `test` file, execute the script:

```
cd $WACO_HOME/code_generator
$WACO_HOME/WACO/SpMM/test_script.sh
```

   - *Output:* The results will be written to an output file named `results_meta`. It took about 1 h for 100 matrices.

References

[1] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017.

[2] Nitish Satya Murthy, Peter Vrancx, Nathan Laubeuf, Peter Debacker, Francky Catthoor, and Marian Verhelst. Learn to learn on chip: Hardware-aware meta-learning for quantized few-shot learning at the edge. *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, pages 14–25, 2022.

[3] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[4] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. Waco: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 920–934, New York, NY, USA, 2023. Association for Computing Machinery.