Final Year Project Report

Tianhao Liu (20205784)

April 26, 2024

Contents

1 Introduction **5**

2 Background **6**

2.1 Multi-Processor System 6

2.2 Hybrid Computing 6

2.3 Hybrid Computing Challenge 7

2.4 OpenH 8

2.5 Ensemble Learning Method 8

2.6 Random Forest Model 9

3 Project Design **10**

3.1 Overview 10

3.2 Sytem Architecture 10

3.3 Metrics 11

4 Experimental Setup **12**

4.1 Dataset 12

4.2 Library 12

4.3 Profiling Tool 12

4.4 Compiler 12

4.5 Operating System and Hardware 13

5 Implementation **14**

5.1 Profiling and Analysis 14

5.1.1 Decision Tree Training Process 14

5.1.2 Profiling Results 15

5.1.3 Profiling Analysis 15

5.2 Parallelization 16

5.2.1 Decision Tree Level Parallelization: 17

5.2.2 Random Forest Parallelization: 18

5.3 Aggregation and Model Output 19

5.4 Performance Evaluation 19

6 Results **20**

6.1 Training Time vs. Dataset Size between OpenMP and OpenACC 20

6.2 Speed up of Hybrid Training Approach 21

6.3 Summary 21

6.4 Further Optimization 22

7 Additional Contribution to OpenH **24**

7.1 Issue Encontered 24

7.2 Possible Solutions 24

7.3 Proposed Solution 25

7.4 Further Improvements 25

8 Future Work **26**

8.1 Expansion to Other Ensemble Models: 26

8.2 Dynamic Load Balancing: 27

8.3 Cross-Platform Scalability: 27

8.4 Memory Management: 27

8.5 Supporting More Parallel Programming Models: 27

9 Conclusion **28**

10 Acknowledgements **28**

List of Figures

1 OpenCL/OpenMP running time ratios vs. innerIters (Kamran Karimi et al.[15]) 8
2 OpenCL/OpenMP running time ratios vs. outerIters (Kamran Karimi et al.[15]) 9
3 OpenCL vs OpenMP on CPUs 10
4 Ensemble Learning (Mohammed et al. [17]) 11
5 Trend in Ensemble Learning Research (Scopus [18]) 12
6 System Architecture 13
7 Dataset (Clanuwat et al.)[20] 14
8 Hardware Configuration 15
9 Parallelization of Find Best Split Function 17
10 Parallelization of Find Best Split Function 19
11 Training Time vs. Dataset Size 21
12 Training Time vs. Dataset Size (Hybrid Training) 22
13 hwloc topology data structure 26

List of Tables

1 Compiler support for OpenACC and OpenMP 14

1 Introduction

In the competitive landscape of machine learning and high-performance computing, the efficiency of model training is a paramount concern that drives the adoption of innovative computational strategies. This paper presents a pioneering approach by introducing the first machine learning model, random forest, that is trained using a hybrid infrastructure employing both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). This novel training method capitalizes on the combined strengths of these processing units to optimize computation times and resource utilization.

The integration of CPUs and GPUs for parallel computing has historically presented notable challenges, primarily due to the complexities of effective workload assignment and processor affinity. Traditional methods often failed to exploit the full potential of these heterogeneous systems due to several limitations inherent in existing tools such as improper thread placement, and suboptimal utilization of the differing architectures of CPUs and GPUs.

To overcome these hurdles, our approach incorporates OpenH, an advanced hybrid programming model that leverages an integrated use of Pthreads, OpenMP, and OpenACC to facilitate the seamless integration of CPUs and GPUs[1]. The results from the deployment of this training methodology demonstrate the robust capability of the OpenH framework in managing and executing a balanced load across the hybrid system, maximizing the utilization efficiencies and markedly reducing the computational overhead associated with machine learning model training.

In the following sections, we will provide an overview of the background, design, implementation, and results of this project, along with a discussion of potential future work and concluding remarks.

2 Background

The evolution of machine learning models and their increasing complexity has led to a significant surge in computational demands. Traditional single-processor systems often fall short in effectively managing the computational burdens associated with advanced algorithms, particularly in the realm of data-intensive tasks such as training deep neural networks. As such, the exploration of more capable and efficient systems has become a crucial research endeavor in this field.

2.1 Multi-Processor System

Historically, the transition from single-processor to multi-processor systems marked a significant leap in computational capabilities. Multi-processor systems, utilizing either multiple CPUs or a combination of CPUs and GPUs, offer parallel processing capabilities that significantly accelerate computational tasks. CPUs, with their general-purpose architectures, are adept at handling complex instructions, while GPUs, originally designed for image processing, excel at executing simpler, concurrent tasks over large blocks of data. This architectural difference inherently positions GPUs as favorable for the parallel execution of numerous operations, which is a common requirement in machine learning tasks.

The advantages and underlying principles of multi-processor systems have been elaborated in key studies such as those by Flynn (1995) and Hennessy et al. (2011), providing foundational knowledge in processor architectures[2] [3]. Despite the advantages, exploiting the full potential of these heterogeneous systems has been a challenging feat due to several technical and architectural hurdles. The parallelism inherent in machine learning tasks requires intricate coordination and efficient data handling capabilities between the different types of processors. Additionally, inherent differences in memory architecture and data processing paradigms between CPUs and GPUs often lead to bottlenecks, which can negate the benefits of parallel processing.

2.2 Hybrid Computing

Hybrid systems that synergize CPU and GPU capabilities aim to combine the strengths of both processor types to enhance performance and energy efficiency. Nodes and servers with diverse components, including multicore CPUs and various accelerators like GPUs and FPGAs, are prevalent in computing for their exceptional performance and energy efficiency. These configurations form the backbone of supercomputers leading both the TOP500 [4] and Green500 [5] rankings, enabling exascale computing. The tools and programming models used for developing parallel programs on heterogeneous servers can be categorized into two main types: vendor-specific tools at a lower level and vendor-agnostic tools at a higher level [1].

Specialized programming tools like CUDA [6] offer programmers low-level APIs, allowing them to optimize parallel programs for performance on Nvidia and AMD GPUs, respectively. These tools are often paired with OpenMP for developing heterogeneous parallel programs. However, their use can lead to vendor lock-in, limiting the portability of the programs.

Numerous high-level programming tools have been suggested to streamline heterogeneous programming. These tools offer elevated abstractions that automate the management of intricate details and optimization specific to architecture, all while minimizing performance trade-offs. The prominent research endeavors in this domain employ directive-based, C++-centered, and skeleton-based methodologies [1].

The C++-oriented methods (including OpenCL[7], SYCL[8], Kokkos[9], and OCCA[10]) enable the utilization of different heterogeneous devices within a single application, all through the standard C++ programming language.

On the other hand, the skeleton-based approaches (such as SkelCL[11] and SkePU[12]) rely on the notion of skeletons, which are higher-order functions embodying common computational patterns and data dependencies like map, reduce, scan, farm, and pipeline. SkelCL and SkePU backends convert codes utilizing these skeletons into OpenCL kernels and C++ library calls, facilitating execution on multicore CPUs and GPUs.

However, while C++-based tools offer code portability by conforming to modern C++ standards and supporting various devices (CPU, GPU, and FPGA), they lack predominant mainstream support as they compete with widely adopted solutions like OpenMP[13] for multicore CPUs and CUDA for Nvidia GPUs.

2.3 Hybrid Computing Challenge

The existing tools and programming models for heterogeneous computing often face a trade-off between performance and portability. For example, OpenCL offers a comprehensive API that enables the programming of both CPU and accelerator program components, facilitating their integration into a unified hybrid application. Consequently, OpenCL competes with OpenMP in programming the CPU components of such hybrid programs and with vendor-specific APIs like CUDA when it comes to programming the accelerator components. According to the study by Kamran Karimi et al.[15], the performance of OpenCL and OpenMP on CPUs is comparable, with OpenCL shows a slight better performance when there are more outer iterations, as shown in figure 2, while OpenMP is faster when there are more inner iterations, as shown in figure 1.

However, in terms of ease of use, OpenCL does entail more programming overhead. The higher learning curve and the necessity for additional setup are inherent in a system primarily designed for GPU programming.

From another research conducted by Kamran Karimi et al.[16], OpenCL is not comparable to CUDA in terms of performance on GPUs. As is shown in graph 3, OpenCL is slower than CUDA on GPUs. This performance gap is attributed to the fact that OpenCL is designed to be a general-purpose API, while CUDA is optimized for Nvidia GPUs.

In summary, the state-of-art tools that provide a single API for both CPUs and GPUs, such as OpenCL, face challenges in terms of performance and ease of use. This performance gap hinders the widespread adoption of these tools for hybrid computing.

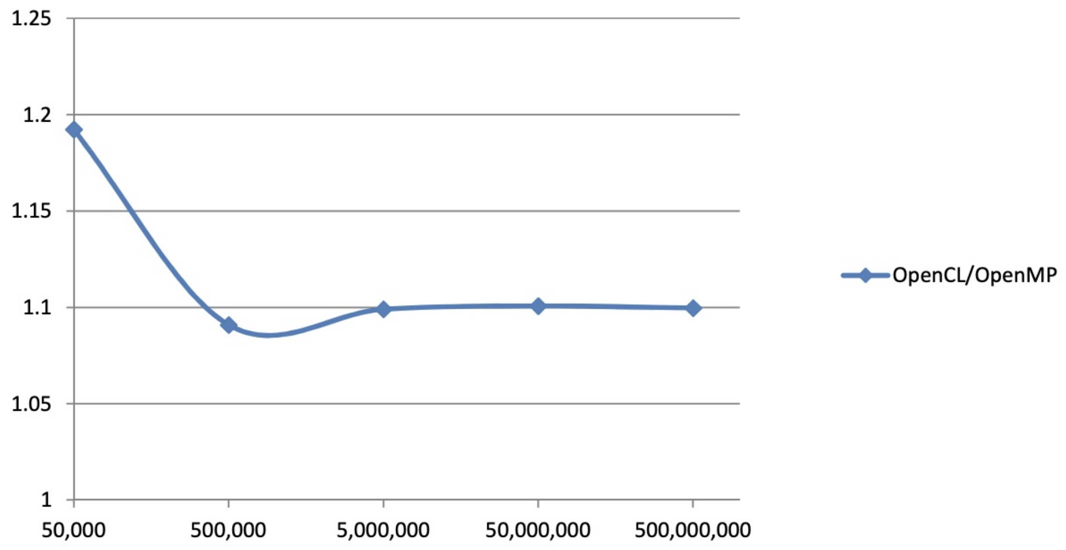


Figure 1: OpenCL/OpenMP running time ratios vs. innerIters (Kamran Karimi et al.[15])

2.4 OpenH

OpenH presents a sophisticated toolkit tailored for managing threads, synchronization, and workload distribution effectively. What sets OpenH apart is its strategy: rather than competing with widely supported mainstream solutions, it leverages and integrates them. For CPU components, OpenH utilizes OpenMP; for accelerators, it employs OpenACC and vendor APIs. Additionally, it utilizes Pthreads and an OpenH-specific API to seamlessly integrate CPU and accelerator components into a unified hybrid program.

This approach enables OpenH to achieve superior performance portability by leveraging established mainstream solutions for programming individual devices within the hybrid server. By balancing computational load between CPUs and GPUs, users can optimize resource utilization and minimize computational overheads, ensuring efficient utilization of system resources.

2.5 Ensemble Learning Method

Ensemble learning is a robust machine learning paradigm where multiple models, often referred to as "weak learners," are trained to solve the same problem and then combined to improve the robustness and accuracy of predictions. This technique capitalizes on the strength of numerous learners to achieve better performance than any single model could accomplish alone. Figure 4 illustrates the concept of ensemble learning, where multiple models are trained independently and then combined to produce a final prediction.

Because of its capability to enhance predictive performance across models, ensemble learning has emerged as a significant research trend in recent years. This trend has consequently resulted in a rise in the utilization of ensemble learning across various domains and applications. As is shown in figure 5, the number of publications on ensemble learning has been increasing steadily over the past decade, indicating the growing interest and relevance of this technique in the machine learning community.

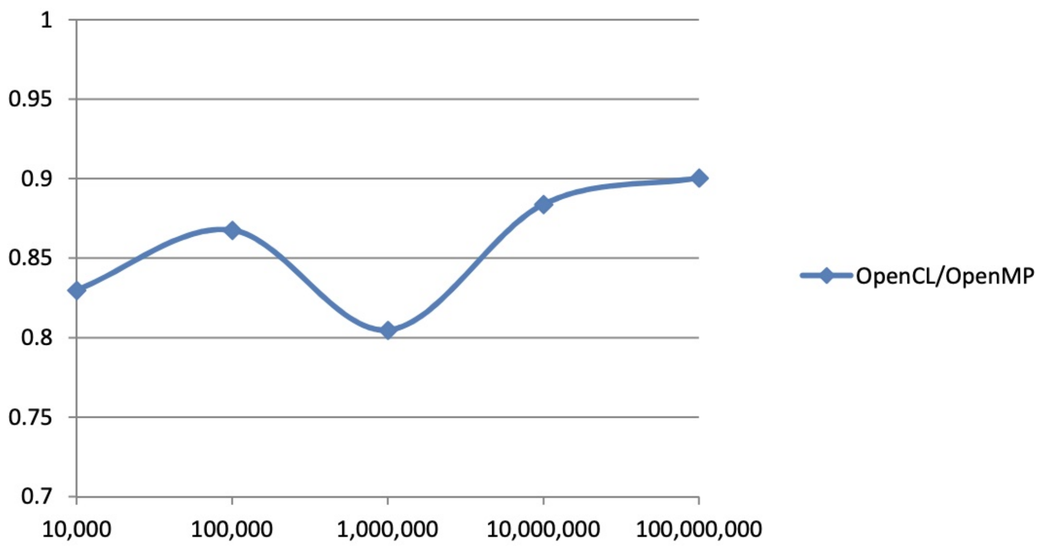


Figure 2: OpenCL/OpenMP running time ratios vs. outerIters (Kamran Karimi et al.[15])

The ensemble methods are generally classified into two main categories: bagging and boosting.

- **Bagging (Bootstrap Aggregating):** Bagging involves training multiple models in parallel, each on a slightly different data sample—typically a random subset with replacement. The individual models are often less correlated with each other, allowing the ensemble to average out their biases and reduce variance. Decision trees are commonly used in bagging setups, with Random Forest being one of the most popular bagging ensemble methods.
- **Boosting**: Boosting involves sequentially training a series of models, where each new model attempts to correct errors made by the previous ones. The models build upon each other, learning from the mistakes, thus improving the overall performance. Gradient boosting and AdaBoost are prominent examples of this approach.

2.6 Random Forest Model

The Random Forest algorithm, introduced by Breiman in 2001 [19], is a powerful ensemble learning method that involves the aggregation of multiple decision trees to improve prediction accuracy and control over-fitting. Random forests consist of an amalgamation of tree predictors, where each tree relies on the values of a random vector sampled independently and from the same distribution for all trees within the forest. As the number of trees in the forest increases, the generalization error for forests asymptotically approaches a limit. This generalization error of a forest composed of tree classifiers is contingent upon both the individual strength of the trees within the forest and the degree of correlation among them. Since random forests perform well with large data sets and are inherently suitable for parallel processing due to the independence of each decision tree during both training and inference phases, it is an ideal candidate for exploring hybrid computing strategies.

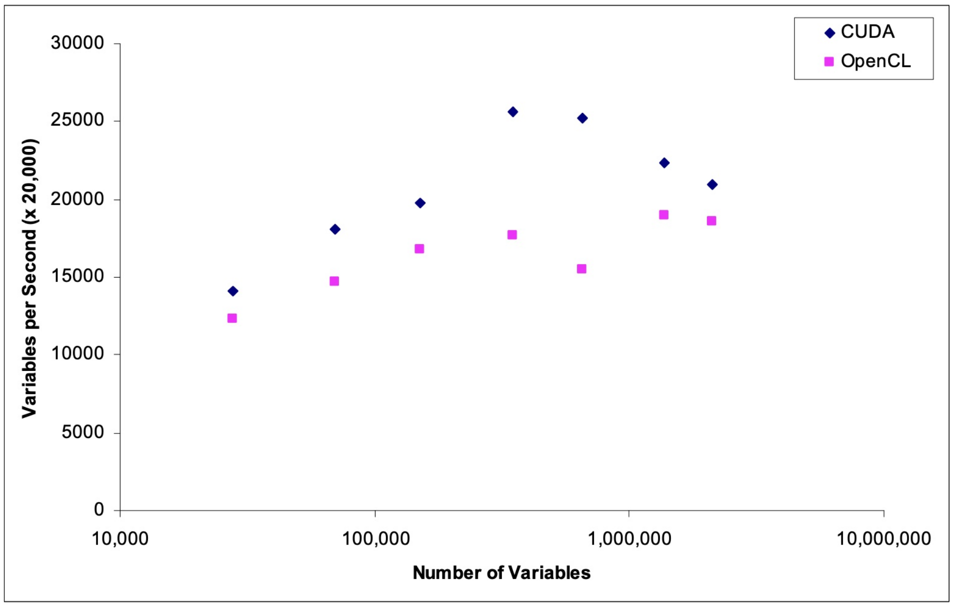


Figure 3: OpenCL vs OpenMP on CPUs

3 Project Design

3.1 Overview

The project is designed to implement a hybrid training approach for a Random Forest model using OpenH, integrating both CPU and GPU resources. The model's performance and resource utilization are then compared against implementations solely using OpenMP (CPU) and OpenACC (GPU).

3.2 Sytem Architecture

OpenH is used as the hybrid programming model to orchestrate the CPU and GPU resources effectively. The system architecture is designed to distribute the workload between CPUs and GPUs using thread management and synchronization mechanisms provided by OpenH.

As illustrated in the figure 6, the system architecture comprises three main steps:

- **Initiation:** The training process commences with the instantiation of a thread. Leveraging OpenH, the system conducts an assessment of device capabilities in the main thread, identifying the number of CPUs and GPUs available and their respective topologies. Then, threads with the same number of accelerators are created to manage the affinity between computation tasks and devices. Different computation tasks are like training different number of decision trees. The workload is distributed across the available accelerators to ensure optimal resource utilization and efficient computation.

- **Training:** After each thread is assigned a specific workload, and receives a set of sampled data, the training

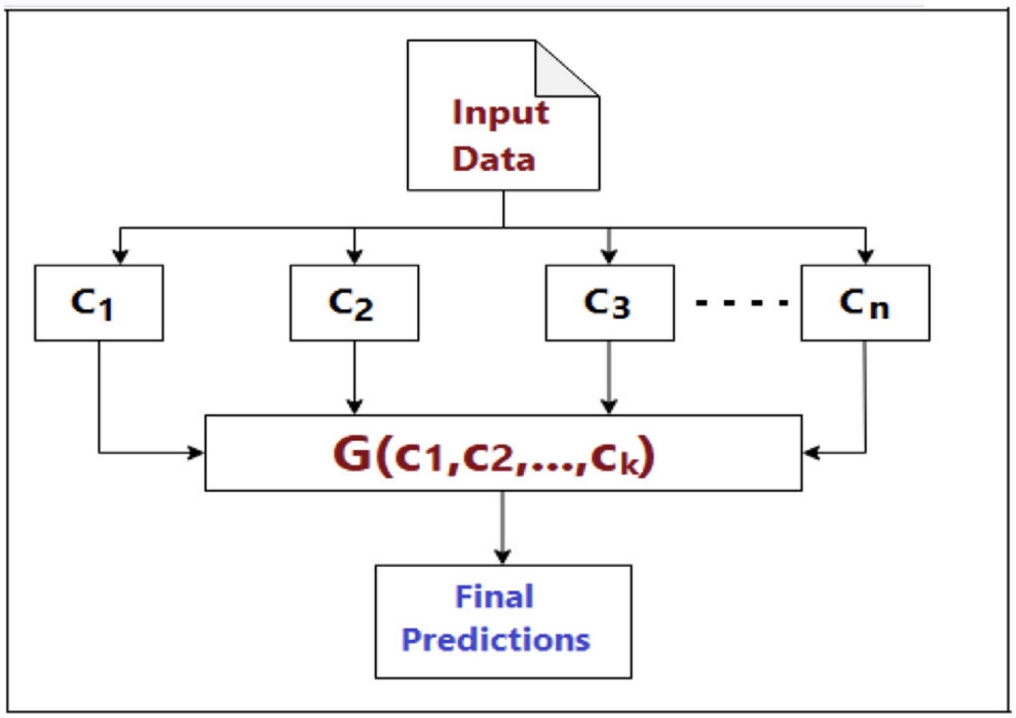


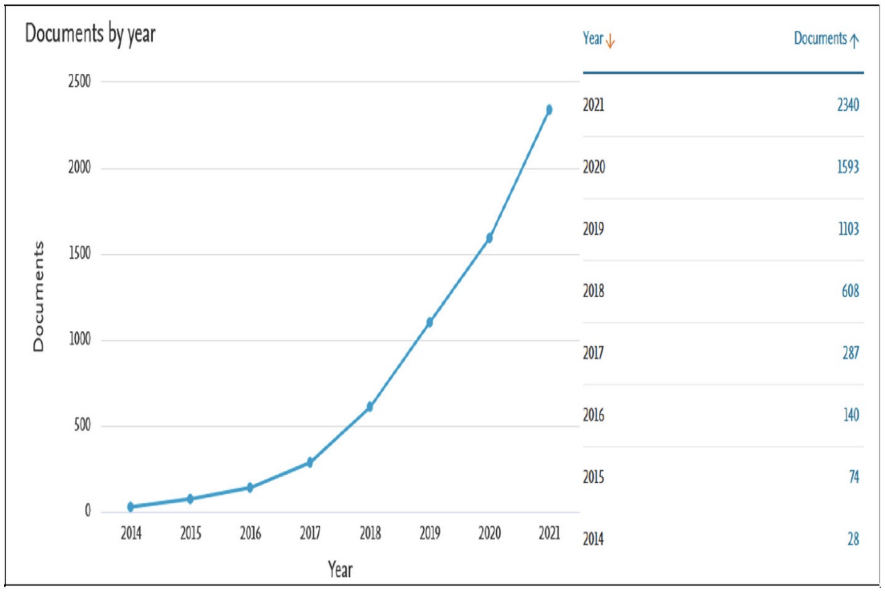
Figure 4: Ensemble Learning (Mohammed et al. [17])

process begins. The decision tree training algorithm is executed in parallel across the multi-core CPU and GPU resources, with each device handling a portion of the workload.

- **Joining:** Upon the conclusion of the training phase, the disparate results obtained from each device are consolidated. These consolidated results are amalgamated to yield the final model output, synthesizing the contributions from all devices into a cohesive and comprehensive model representation.

3.3 Metrics

For a machine learning model, there are many metrics used to evaluate its performance and efficiency like precisions, recalls, F1 scores, and ROC curves. However, these metrics are dependent one the data input and the hyper-parameters of the model. In this project, we are using parallel computing to train the Random Forest model, so these matrics are expected to be the same as the sequential training model. Therefore, the main metric we are focusing on is the training time only.



Year	Documents
2021	2340
2020	1593
2019	1103
2018	608
2017	287
2016	140
2015	74
2014	28

Figure 5: Trend in Ensemble Learning Research (Scopus [18])

4 Experimental Setup

4.1 Dataset

The dataset used for training is Kuzushiji-49, a MNIST-like dataset that has 49 classes (28x28 grayscale, 270,912 images) from 48 Hiragana characters and one Hiragana iteration mark.

4.2 Library

The programming model is implemented using OpenH, which strategically integrates Pthreads, OpenMP, and OpenACC, enabling a robust hybrid programming environment suitable for high-performance computing.

4.3 Profiling Tool

Profiling is conducted using NVIDIA Nsight Systems, providing a deep analysis of performance and GPU utilization to optimize computational workloads efficiently.

4.4 Compiler

The compiler used for the implementation is NVC (NVIDIA HPC Compiler), fully supporting OpenH, OpenACC, and OpenMP directives. The compiler support for OpenACC and OpenMP is shown in table 1.

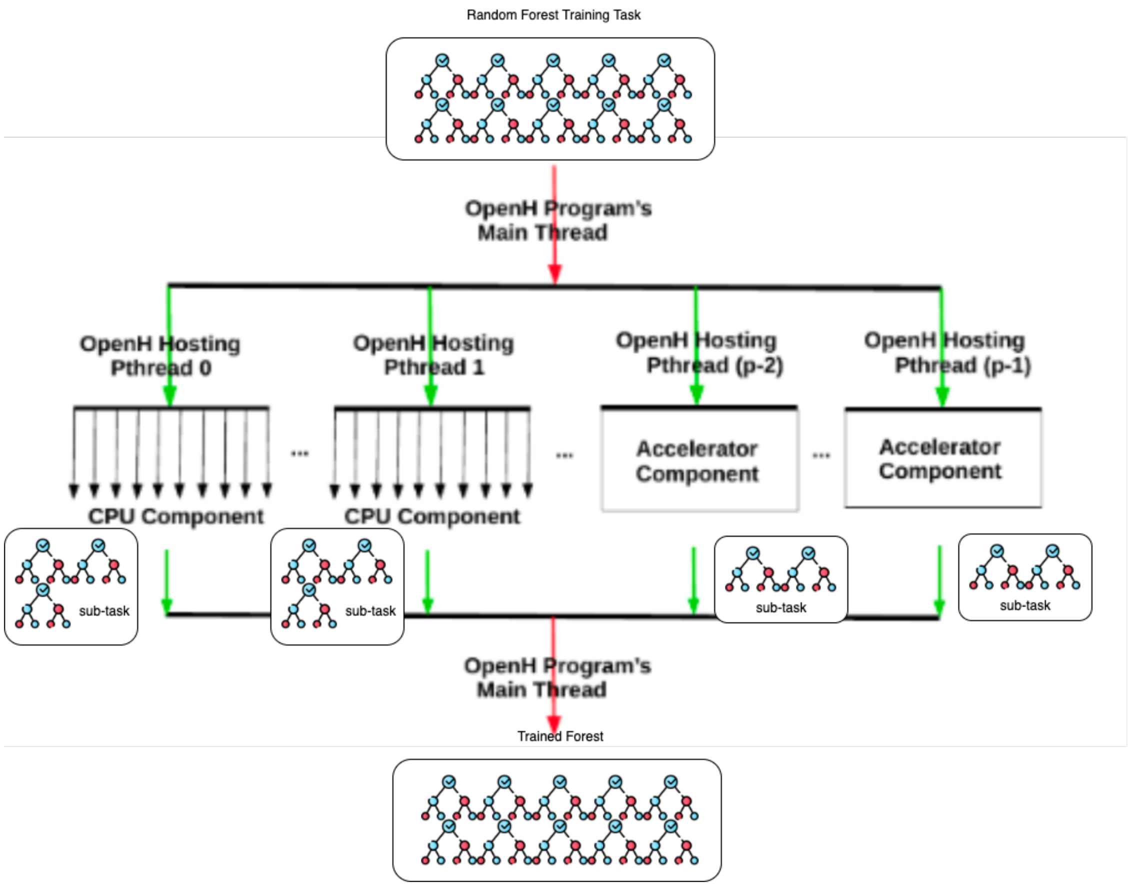
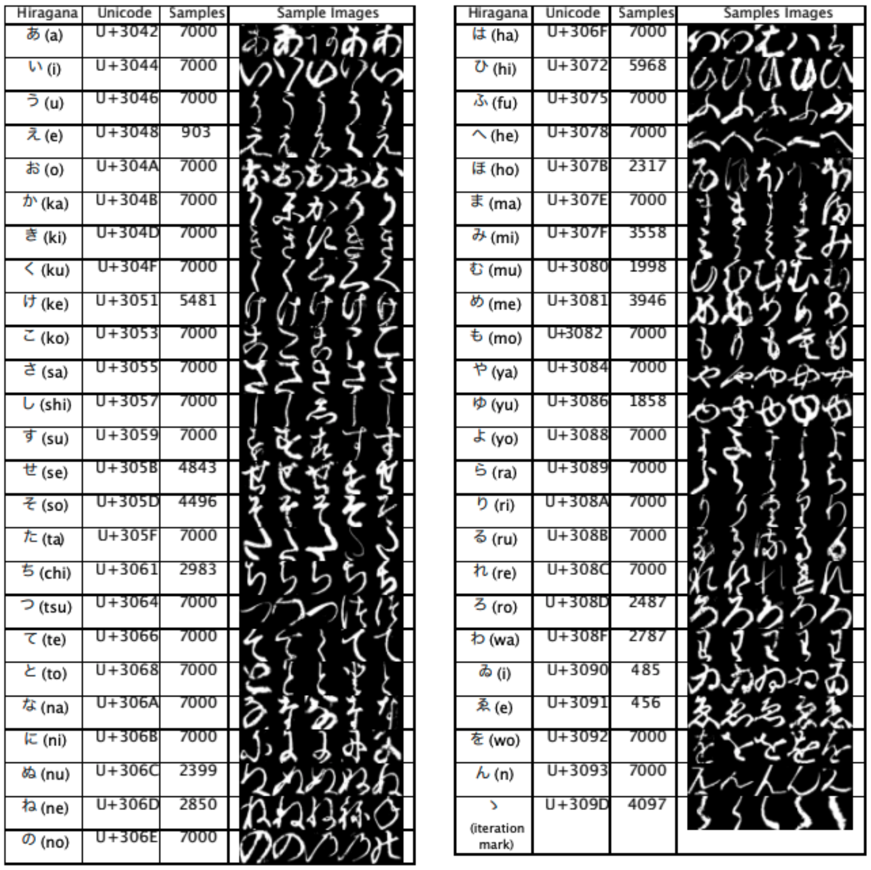


Figure 6: System Architecture

4.5 Operating System and Hardware

- **Operating System:** As the OpenH library utilizes the POSIX API for thread management, the experiments are conducted on a Linux-based operating system.
- **Hardware:**
  1. **CPU Architecture:** Intel(R) Xeon(R) Platinum 8362 CPU @ 2.80GHz, adept at handling high computational demands.
  2. **CPU Cores:** Dual-threaded 32 cores provide robust parallelism on 64 logical processors.
  3. **GPU Model:** Dual NVIDIA A40 GPUs, renowned for their high computational power and substantial memory allocation.
  4. **GPU Memory:** Each GPU supports 46,068 MiB, facilitating extensive parallel data processing.

The hardware configuration is depicted in figure 8, which is generated using the lstopo tool.



Hiragana	Unicode	Samples	Sample Images
あ (a)	U+3042	7000	
い (i)	U+3044	7000	
う (u)	U+3046	7000	
え (e)	U+3048	903	
お (o)	U+304A	7000	
か (ka)	U+304B	7000	
き (ki)	U+304D	7000	
く (ku)	U+304F	7000	
け (ke)	U+3051	5481	
こ (ko)	U+3053	7000	
さ (sa)	U+3055	7000	
し (shi)	U+3057	7000	
す (su)	U+3059	7000	
せ (se)	U+305B	4843	
そ (so)	U+305D	4496	
た (ta)	U+305F	7000	
ち (chi)	U+3061	2983	
つ (tsu)	U+3064	7000	
て (te)	U+3066	7000	
と (to)	U+3068	7000	
な (na)	U+306A	7000	
に (ni)	U+306B	7000	
ぬ (nu)	U+306C	2399	
ね (ne)	U+306D	2850	
の (no)	U+306E	7000	

Hiragana	Unicode	Samples	Samples Images
は (ha)	U+306F	7000	
ひ (hi)	U+3072	5968	
ふ (fu)	U+3075	7000	
へ (he)	U+3078	7000	
ほ (ho)	U+307B	2317	
ま (ma)	U+307E	7000	
み (mi)	U+307F	3558	
む (mu)	U+3080	1998	
め (me)	U+3081	3946	
も (mo)	U+3082	7000	
や (ya)	U+3084	7000	
ゆ (yu)	U+3086	1858	
よ (yo)	U+3088	7000	
ら (ra)	U+3089	7000	
り (ri)	U+308A	7000	
る (ru)	U+308B	7000	
れ (re)	U+308C	7000	
ろ (ro)	U+308D	2487	
わ (wa)	U+308F	2787	
ゐ (i)	U+3090	485	
ゑ (e)	U+3091	456	
を (wo)	U+3092	7000	
ん (n)	U+3093	7000	
ゝ (iteration mark)	U+309D	4097	

Figure 7: Dataset (Clanuwat et al.)[20]

Compiler	OpenACC Support	OpenMP Support
GCC	Partial	Yes
NVC	Yes	Yes
PGI	Yes	Yes
Clang	Limited	Yes

Table 1: Compiler support for OpenACC and OpenMP

5 Implementation

The implementation of our Random Forest training involved several key phases, focusing on profiling, parallelization, and efficient management of hybrid computing resources:

5.1 Profiling and Analysis

5.1.1 Decision Tree Training Process

The training process for a decision tree involves recursively partitioning the data based on the best split feature and value at each node. The pseudo-code for the decision tree training process is outlined in Algorithm 1.

The training process starts by verifying whether the depth limit has been reached or if the data is homogeneous.

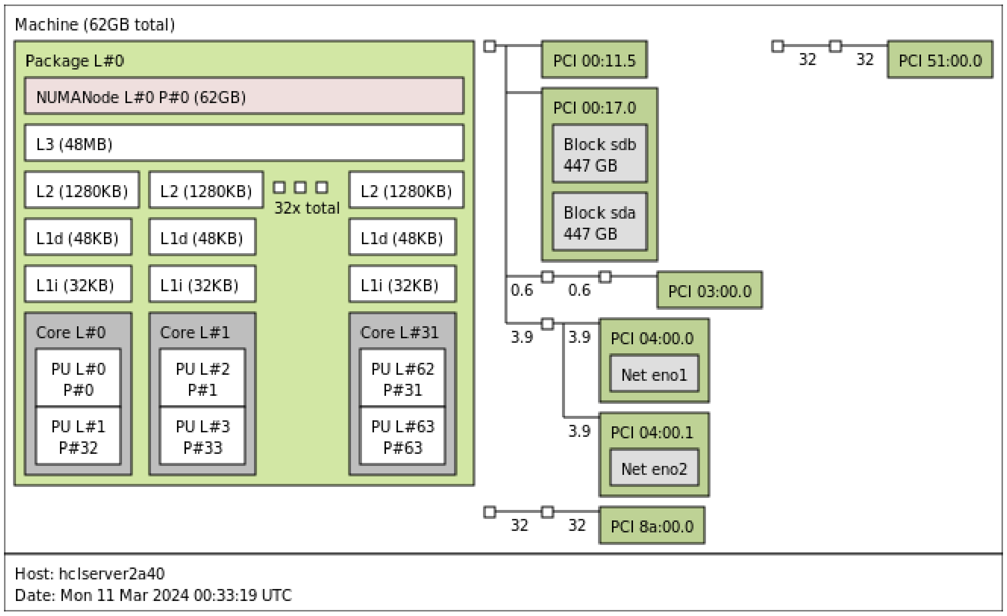


Figure 8: Hardware Configuration

In either case, a leaf node is formed. If the depth limit hasn't been reached, the algorithm proceeds by iterating through each feature to identify the optimal split using a designated metric (such as information gain). The feature yielding the maximum gain is chosen, and the dataset is divided accordingly. This process recurses for the resulting left and right subsets until either the depth limit is attained or the data becomes homogeneous.

5.1.2 Profiling Results

Initial efforts were directed towards profiling the existing sequential decision tree training algorithm. It was determined that the major computational bottleneck was the function to find the best split feature and value, which consumed approximately 99% of the training time. This insight was crucial as it highlighted a potential area for optimization through parallelization.

5.1.3 Profiling Analysis

The most computationally intensive tasks in training decision trees is the process of identifying the optimal split point at each node, known as 'finding the best split'. The significant time consumption of this process is attributed primarily to the series of operations required to evaluate every conceivable split across all features in a dataset.

At each node, the decision tree algorithm assesses various possible ways to split the data based on each feature. This entails examining each unique feature value and computing a specific metric to ascertain the quality of the split. Common metrics used are information gain for classification tasks or variance reduction for regression. Calculating these metrics necessitates dividing the dataset into subsets according to the split criterion and subsequently measuring how well these subsets align with desired outcomes. For categorical targets, this might

Algorithm 1 Decision Tree Training Process

```
procedure TrainDecisionTree($Data, DepthLimit$)
    if $DepthLimit = 0$ or $Data$ is homogeneous then
        return CreateLeaf($Data$)
    end if
    $BestSplit \leftarrow$ None
    $MaxGain \leftarrow 0$
    for each feature $f$ in $Data$ do
        $Split, Gain \leftarrow$ FindBestSplit($Data, f$)
        if $Gain > MaxGain$ then
            $BestSplit \leftarrow Split$
            $MaxGain \leftarrow Gain$
        end if
    end for
    if $BestSplit$ is None then
        return CreateLeaf($Data$)
    end if
    $LeftData, RightData \leftarrow$ SplitData($Data, BestSplit$)
    $LeftTree \leftarrow$ TrainDecisionTree($LeftData, DepthLimit - 1$)
    $RightTree \leftarrow$ TrainDecisionTree($RightData, DepthLimit - 1$)
    return CreateNode($BestSplit, LeftTree, RightTree$)
end procedure
```

involve evaluating metrics such as entropy or class frequency, while for numerical targets, statistical operations like computing means and variances are necessary.

The primary reason this task consumes considerable computational resources is its inherent complexity, often exacerbated by the size of the data and its feature space. The complexity reaches up to O(n * m * log(n)), where (n) is the number of data points and (m) is the number of features, exacerbated by the necessity for sorting data points for continuous features. Larger datasets further amplify computation times by linearly increasing the number of potential splits to be evaluated.

Moreover, the procedure of iterating over all potential splits involves nested loops over data points and features, significantly increasing the total number of operations required. This exhaustive approach ensures that the algorithm considers all available information, but at the cost of increased computation time.

To mitigate these challenges, parallel computing is adapted where the task of evaluating different splits is distributed across multiple processors or GPUs. This division of labor can dramatically reduce the time required to find the optimal splits.

5.2 Parallelization

To address the identified computational bottleneck, we implemented parallelization at two levels: decision tree level and Random Forest level. During the implementation, I encountered several challenges and I tried different approach to solve them.

5.2.1 Decision Tree Level Parallelization:

Attempts 1: The initial parallelization strategy focused on parallelizing the process of sub-tree generation, where training each sub-tree can be executed independently. This is because the data subsets used to train each sub-tree are distinct and do not overlap, making them ideal candidates for parallel execution. Here is the pseudo-code for the parallelization of the decision tree training process: The red lines in the pseudo-code indicate the parallelized

Algorithm 2 Parallelization in Sub-tree Generation

```
1: procedure TrainDecisionTree($Data, DepthLimit$)
2:     if $DepthLimit = 0$ or $Data$ is homogeneous then
3:         return CreateLeaf($Data$)
4:     end if
5:     // get the best split feature and threshold
6:     generate the left sub-tree based on the data with the split feature less than the threshold
7:     generate the right sub-tree based on the data with the split feature greater than the threshold
8: end procedure
```

sections of the decision tree training process. The left and right sub-trees are generated independently, allowing for parallel execution of the training process as the figure 9 shows.

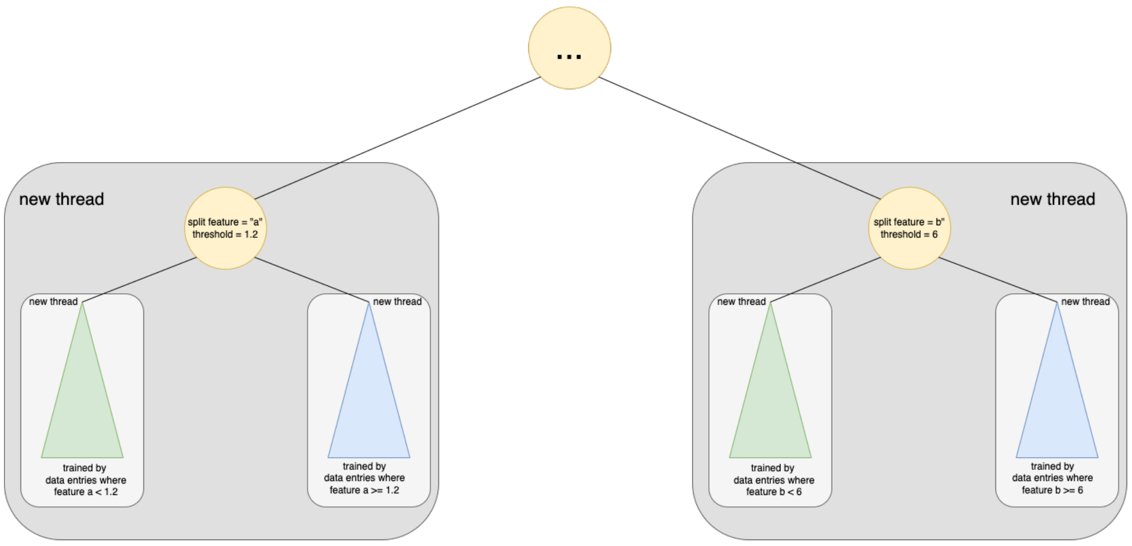

...
new thread
split feature = "a" threshold = 1.2
new thread
split feature = b" threshold = 6
new thread
trained by data entries where feature a < 1.2
new thread
trained by data entries where feature a >= 1.2
new thread
trained by data entries where feature b < 6
new thread
trained by data entries where feature b >= 6

Figure 9: Parallelization of Find Best Split Function

However, this parallelization strategy did not yield the expected performance improvements due to the limited parallelism achieved at the decision tree level. No matter how many sub-trees are generated in parallel, there is only around 40% speed up compared to the sequential implementation. Given there are 64 logical processors available, the performance gain was suboptimal, indicating that the sub-tree generation process was not sufficiently granular to exploit the full potential of parallel computing resources.

After analyzing the parallization strategy, I found that the bottleneck is due to workload imbalance. At the beginning of the training process, there is only one thread that generates the sub-trees in the root layer, then two threads are used to generate the sub-trees in the second layer, and

$$2^k$$

threads are used to generate the sub-trees in the k-th layer. This leads to a workload imbalance issue, where the thread generating the sub-trees in the root layer has to wait for the threads generating the sub-trees in the deeper layers to finish.

From this experience, I realized that in this case, the parallelization strategy should be more fine-grained to achieve better performance. As a result, I start my second attemptation.

Attempts 2: Taken the previous experience, I tried a more fine-grained parallelization strategy by parallelizing the process of finding the best split feature and threshold. This process involves calculating the largest information gain across all features and their respective values. As this operation is computationally intensive and independent for each feature, it is well-suited for parallel execution. The pseudo-code for this parallelization strategy is as follows:

Algorithm 3 Parallelization in Find Best Split Function

```
1: procedure FindBestSplit($Data$)
2:     $BestSplit \leftarrow$ None
3:     $MaxGain \leftarrow 0$
4:     for each feature $f$ in $Data$ do
5:         $Split, Gain \leftarrow$ FindBestSplit($Data, f$)
6:         if $Gain > MaxGain$ then
7:             $BestSplit \leftarrow Split$
8:             $MaxGain \leftarrow Gain$
9:         end if
10:    end for
11:    return $BestSplit, MaxGain$
12: end procedure
```

The red line in the pseudo-code indicates the parallelized section of the decision tree training process. The process of finding the best split feature and threshold is parallelized across all features, enabling concurrent evaluation of potential splits and enhancing computational efficiency, as shown in figure 10.

This parallelization strategy significantly improved the performance of the decision tree training process by distributing the computational workload more evenly across the available processors. The workload imbalance issue was mitigated, and the training time was reduced by more than 99.5% compared to the sequential implementation.

5.2.2 Random Forest Parallelization:

Random Forest model comprises several decision trees, each trained on a distinct data subset. As depicted in Figure 6, the training process of decision trees can be parallelized, enabling workload distribution across multiple

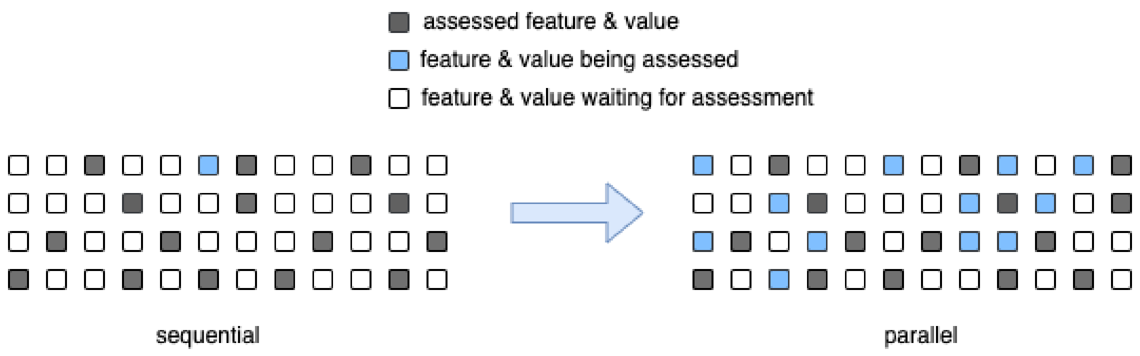

assessed feature & value
feature & value being assessed
feature & value waiting for assessment
sequential
parallel

Figure 10: Parallelization of Find Best Split Function

CPUs and GPUs.

Initially, we can evaluate the efficiency of pure CPU and GPU training processes to determine the optimal workload distribution between them. Based on the performance of each device, users can effectively allocate data subsets and corresponding decision tree computations across various accelerators (GPUs) and CPUs, achieving load balancing and optimal resource utilization.

To facilitate efficient data transfer and synchronization between devices, we utilize OpenH to manage thread affinity and resolve conflicts arising from the concurrent use of OpenMP and OpenACC.

As illustrated in the pseudo-code of the Random Forest training process in Algorithm 4, the training process comprises two primary phases: training decision trees on CPUs and GPUs, and aggregating the results to form the final Random Forest model.

Code segments highlighted in red and blue are designated to different threads for training decision trees on CPUs and GPUs, respectively. Once all threads complete their assigned tasks, the results are automatically combined to generate the final Random Forest model.

5.3 Aggregation and Model Output

After completing the training process for each decision tree, the tree is copied to a specified memory location. In my implementation, this memory location resides in a contiguous memory space with a fixed size allocated for each tree. Once all decision trees have been trained, their results are automatically aggregated to form the final Random Forest model.

This aggregation process avoids potential conflicts arising from concurrent memory access and ensures that the final model is accurately constructed.

5.4 Performance Evaluation

The performance of the hybrid training approach was evaluated based on the execution time of the Random Forest training process. The training time was compared against implementations using OpenMP (CPU) and OpenACC (GPU) solely to assess the efficiency of the hybrid approach in optimizing training speed.

Algorithm 4 Random Forest Level Parallelization

```
1:  procedure TrainDecisionTreeOnCPU($SampledData, DepthLimit$)
2:      $Tree \leftarrow$ TrainDecisionTree($SampledData, DepthLimit$)
3:      return $Tree$
4:  end procedure
5:
6:  procedure TrainDecisionTreeOnGPU($SampledData, DepthLimit$)
7:      $Tree \leftarrow$ TrainDecisionTree($SampledData, DepthLimit$)
8:      return $Tree$
9:  end procedure
10:
11: procedure TrainRandomForest($Data, DepthLimit, NumCPUs, NumGPUs$)
12:     $Forest \leftarrow []$
13:     $SampledData \leftarrow$ SampleData($Data$)
14:     for $i \leftarrow 1$ to $NumCPUs$ do
15:         $Tree \leftarrow$ TrainDecisionTreeOnCPU($DataChunks[i], DepthLimit$)
16:         $Forest[i] \leftarrow Tree$
17:     end for
18:     for $i \leftarrow 1$ to $NumGPUs$ do
19:         $Tree \leftarrow$ TrainDecisionTreeOnGPU($DataChunks[NumCPUs + i], DepthLimit$)
20:         $Forest[i + NumCPUs] \leftarrow Tree$
21:     end for
22:     Join all threads
23:     return $Forest$
24: end procedure
```

6 Results

To use the hybrid training model effectively, we need to determine the optimal workload distribution between CPUs and GPUs. The workload distribution is determined based on the performance of each device, ensuring that the training process is efficiently balanced across all available resources. As a result, the training time against data size is compared between OpenMP and OpenACC, which provides insights into the performance of each device and guides the workload distribution strategy. After determining the optimal workload distribution, the hybrid training approach is implemented, leveraging the strengths of both CPUs and GPUs to achieve optimal performance.

6.1 Training Time vs. Dataset Size between OpenMP and OpenACC

The figure 11 shows the training time of the Random Forest model using OpenMP, OpenACC. From the figure, we can see that the training time increases linearly with the size of the dataset. However, the training time of OpenACC is longer than OpenMP. There are several possible reasons for this result. One reason is that the dataset is not large enough to fully utilize the GPU resources. Another reason is that the training process is not optimized for GPU, for example, too much data transfer between CPU and GPU making the training process slower.

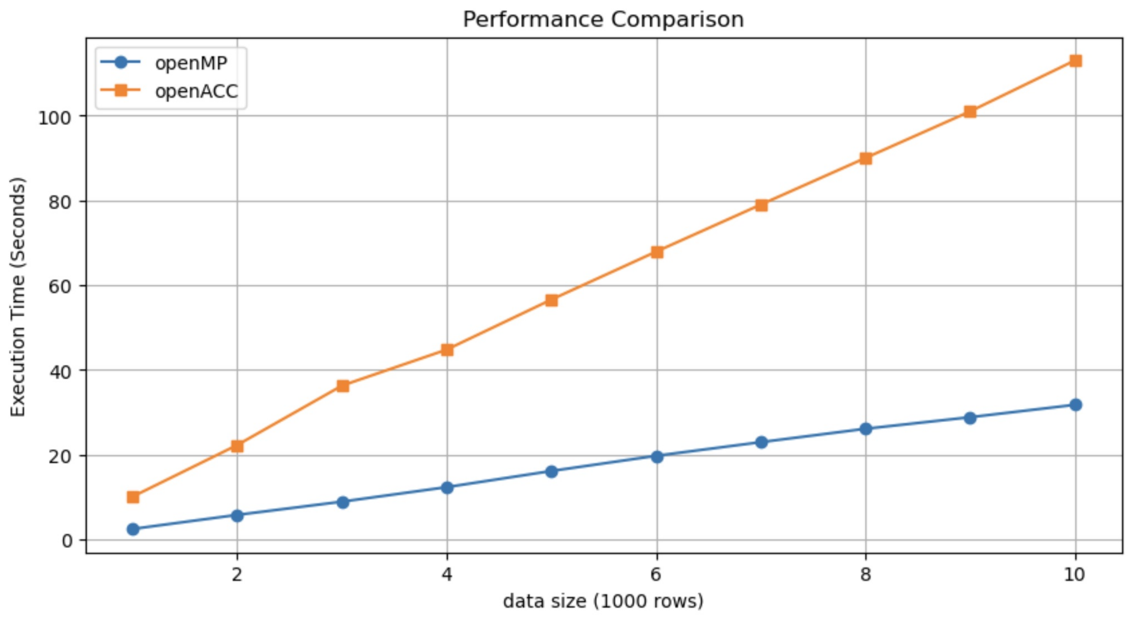


Figure 11: Training Time vs. Dataset Size

6.2 Speed up of Hybrid Training Approach

Although the current version of GPU training have a huge space for improvement, we can still utilize the hybrid training approach to improve the overall training speed. The figure 12 shows the training time of the Random Forest model using the hybrid training approach.

Based on the previous comparison, I have allocated the workload to the CPU and GPU accordingly. Given that the CPU outperforms the GPU by 3 to 4 times, I've assigned 4/5 of the workload to the CPU and 1/5 to the GPU. Consequently, during the training process, the CPU handles the training of 8 decision trees, while the GPU takes responsibility for training the remaining 2 decision trees.

From Figure 12, it's evident that the hybrid training approach exhibits shorter training times compared to both OpenMP and OpenACC. Specifically, the execution time of the hybrid approach is reduced by 15% compared to OpenMP and 76% compared to OpenACC. These results underscore the effectiveness of the hybrid training approach in harnessing the combined strengths of CPUs and GPUs to achieve optimal performance.

Further performance enhancements can be achieved by optimizing the GPU training process, such as minimizing data transfers between the CPU and GPU and maximizing the utilization of GPU resources. Additionally, if workload distribution achieves a balance, the training time of the hybrid approach can be further halved compared to the minimum training time attained with OpenMP and OpenACC.

6.3 Summary

When employing OpenH, developers must carefully consider workload distribution between the CPU and GPU to attain optimal performance. To achieving the optimal performance, developers may need to adopt various

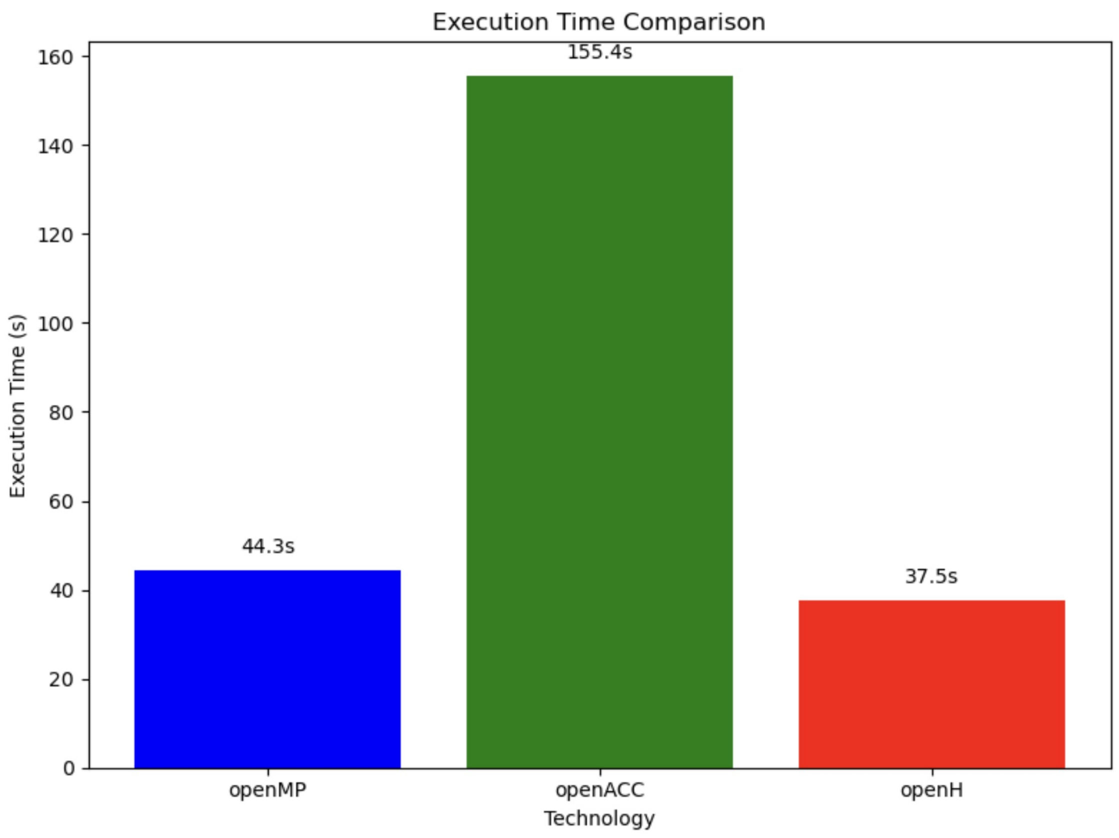


Figure 12: Training Time vs. Dataset Size (Hybrid Training)

parallelization strategies for tasks training on different accelerators, which demands a high level of expertise in parallel computing.

Despite these challenges, the hybrid computing model presents an opportunity to harness the advantages of both CPUs and GPUs, potentially achieving the best of both worlds in terms of performance. By leveraging the respective strengths of CPUs and GPUs, developers can strive towards achieving optimal performance for their applications.

6.4 Further Optimization

From the results from the experiment above, we can see a huge space for improvement in the GPU training process. If the GPU training process is much slower than the CPU training process, the hybrid training approach will not be able to achieve a good performance no matter how the workload is distributed.

To further optimize the GPU training process, we can consider the following aspects:

- **Data Transfer Optimization:** Minimize data transfers between the CPU and GPU by optimizing data structures and memory allocation. Utilize shared memory or pinned memory to enhance data transfer speeds and reduce latency.

- **Memory Management:** Efficiently manage GPU memory by minimizing memory allocations and deallocations. Utilize memory pools and reuse memory buffers to reduce memory overhead and enhance memory access speeds.
- **Try Other Parallel Programming models:** If the OpenACC model is not efficient enough, we can try other parallel programming models like CUDA or OpenCL. Since OpenH supports multiple parallel programming models, we can easily switch between different models to find the most efficient one for our application.

7 Additional Contribution to OpenH

OpenH has recently been developed and has limited documentation, with full testing yet to be completed. During the implementation of the Random Forest model, I encountered a portability issue with OpenH, necessitating further modifications to improve its adaptability and usability. After reviewing the source code of the OpenH library and engaging in several discussions with the developers, I proposed a solution to enhance the device detection process within OpenH. The modified implementation incorporates the hwloc library[21], a widely-used tool for hardware topology detection, to dynamically identify the system's hardware configuration and topology.

7.1 Issue Encontered

During the implementation of the Random Forest model using OpenH, I encountered a portability issue related to device detection. After reading the source code, I found that in the original OpenH implementation, the device detection process was completed by reading a static configuration file. This static configuration file contained information about the number of CPUs and GPUs available, as well as their respective topologies. However, this static configuration approach was not flexible enough to adapt to dynamic changes in the system configuration. This is because different device may adopt different thread-core mapping strategies, which will change the format of the configuration file and lead to the failure of the device detection process.

7.2 Possible Solutions

subsubsectionBuild Wheels from Scratch: Given the encountered issue, we can initially identify the mapping scheme utilized by the device or prompt the user to provide it. Then, we can employ different approaches to read the configuration file accordingly.

- **Pros:** This approach offers complete customization and control over the device detection process. It is straightforward and can be implemented with minimal effort.
- **Cons:** This method necessitates significant manual intervention and may not be scalable for large-scale systems. Building wheels from scratch can be time-consuming and prone to errors. Additionally, maintaining the tool poses challenges, as the encountered mapping scheme might be just the first of many, potentially caused by different devices or versions of the same device.

subsubsectionUsing existing tools: Another approach involves leveraging existing tools to simplify the device detection process. Fortunately, there exists a widely-used library called hwloc, which offers a comprehensive API for detecting and managing hardware resources, including CPUs, GPUs, and memory. By integrating hwloc into OpenH, we can harness its functionalities to dynamically detect the system's hardware configuration and topology, ensuring accurate and adaptable device detection.

- **Pros:** This approach utilizes existing tools and libraries to streamline the device detection process. Incorporating hwloc enables us to benefit from its robust hardware topology detection capabilities, ensuring accurate and adaptable device detection. Moreover, maintenance of this aspect can be delegated to the hwloc team, enhancing reliability and efficiency.

- **Cons:** Integrating hwloc may introduce additional dependencies to the OpenH library, potentially complicating the build process. Efforts are required to learn the API of hwloc and understand how to integrate it into the OpenH library.

7.3 Proposed Solution

After evaluating the two possible solutions and discussed with the OpenH developer, I proposed to integrate the hwloc library into the OpenH library to enhance the device detection process. Just as the old saying goes in industry, "Don't reinvent the wheel", the hwloc library is a widely-used tool for hardware topology detection, and it has been well-tested and maintained by the community.

After spending time learning hwloc, I found that hwloc offers detailed information about the system's hardware topology and provide a tree-like data structure as shown in figure 13, which can be used to identify the relationships between different hardware components.

We can use this tree-like topology data structure to reconstruct the library's device detection process, ensuring accurate and adaptable device detection. For example, to detect the mapping between logical threads and cores, we can traverse the hwloc topology tree and extract the information of each core and its children threads. By extracting these information, we can dynamically identify the system's hardware configuration and topology, ensuring accurate and adaptable device detection.

By now, the logical thread and physical core mapping scheme detection is fully implemented and tested. My hybrid training model is currently using the new version of OpenH.

7.4 Further Improvements

By testing the ability of hwloc to detect GPU devices, I found that its GPU detection capabilities are limited and unstable. Since there is no standard API for different GPU vendors, building the wheels from scratch is the only way to detect GPU devices accurately.

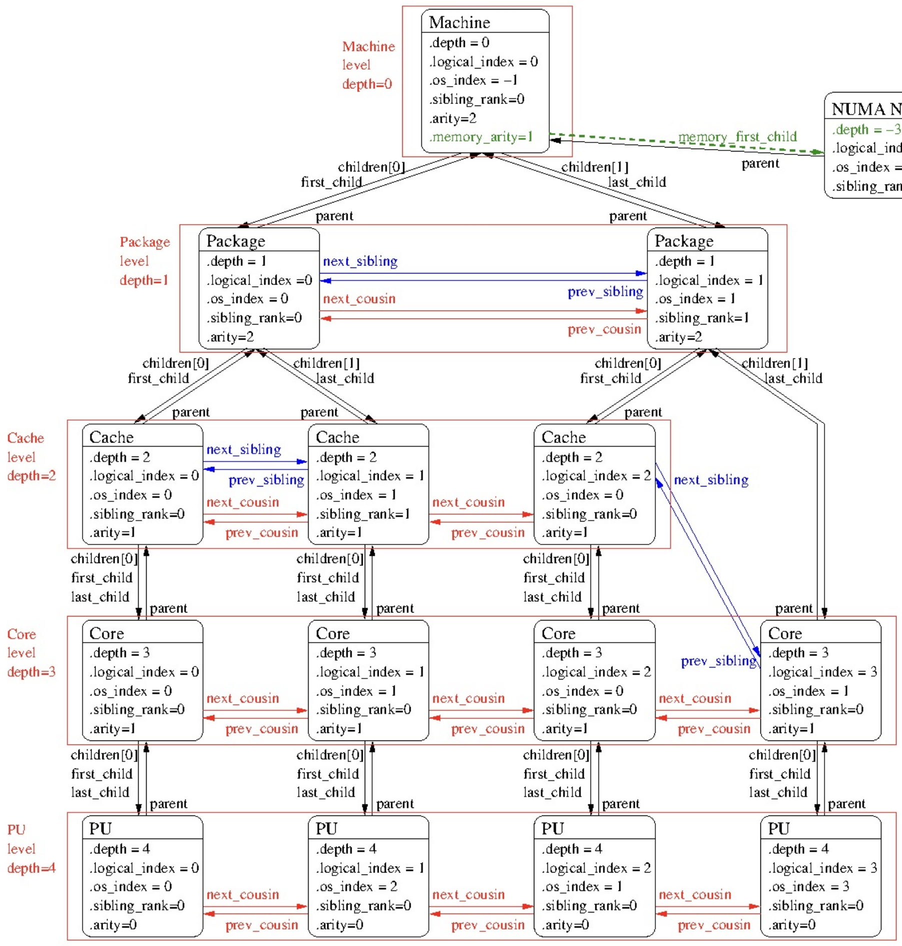


Figure 13: hwloc topology data structure

8 Future Work

The successful implementation of OpenH in training Random Forest models represents a significant advancement in the utilization of hybrid computing techniques for machine learning. This methodology has demonstrated substantial benefits in handling high-dimensional and large-scale data by efficiently distributing and balancing computational loads across both CPU and GPU resources. Looking ahead, there are several promising directions for extending this work to further enhance the capabilities and applications of ensemble machine learning techniques:

8.1 Expansion to Other Ensemble Models:

Our current focus on Random Forest models opens the door to applying similar hybrid training methodologies to other ensemble-based machine learning models. Techniques such as boosting and stacking could benefit from the parallelization strategies developed in this project, potentially leading to significant performance improvements in a broader range of machine learning applications.

8.2 Dynamic Load Balancing:

Currently, OpenH requires users to manually specify the distribution of data subsets and decision tree computations across different devices. Future work could explore the implementation of dynamic load balancing algorithms that automatically adjust the workload distribution based on the system's performance metrics and resource availability.

8.3 Cross-Platform Scalability:

OpenH is currently based on POSIX APIs, limiting its portability across different operating systems. However, its idea can be extended to other platforms, such as Windows, by leveraging platform-specific APIs and libraries to achieve cross-platform scalability. Expanding the hybrid training framework to support a broader range of computing environments would enhance its accessibility and applicability in diverse machine learning scenarios.

8.4 Memory Management:

For large-scale machine learning applications, efficient memory management is crucial to optimize resource utilization and minimize data transfer overhead. In my project, memory switch between CPU and GPU is a major bottleneck, future work could focus on developing advanced memory management techniques to streamline data transfer and enhance overall system performance.

8.5 Supporting More Parallel Programming Models:

While OpenH currently supports Pthreads, OpenMP, and OpenACC, future work could explore integrating additional parallel programming models, such as OpenCL, CUDA, Kokkos, and OCCA, to provide users with a more comprehensive set of tools for hybrid computing. As the core of OpenH is designed to be extensible and decentralized, integrating new parallel programming models would enhance its flexibility and adaptability to diverse hardware architectures.

9 Conclusion

In conclusion, the incorporation of OpenH to facilitate the hybrid training of Random Forest models represents a transformative step in leveraging the full potential of modern computational resources. This approach not only maximizes the efficiencies of both CPU and GPU architectures but also serves as a blueprint for optimizing machine learning workflows that deal with extensive datasets and computationally intensive algorithms. Future explorations, as outlined, will focus on refining these methodologies and expanding their applications. By extending the hybrid training approach to encompass more algorithms and integrating advanced model management and scalability features, we aim to contribute further to the evolving field of machine learning.

10 Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Alaxey Lastovetsky, for his invaluable guidance and support throughout this project. His expertise and insights have been instrumental in shaping the direction and outcomes of this research. Also, I would like to thank Dr. Ravi Manumachu for his suggestions and guidance on improving the OpenH library.

References

[1] Farrelly, Simon, Ravi Reddy Manumachu, and Alexey Lastovetsky. “OpenH: A Novel Programming Model and API for Developing Portable Parallel Programs on Heterogeneous Hybrid Servers.” *IEEE Access* 12 (2024): 23666-94.

[2] Flynn, Michael J. *Computer Architecture: Pipelined and Parallel Processor Design*. 1st ed. USA: Jones and Bartlett Publishers, Inc., 1995. ISBN: 0867202041.

[3] Patterson, David A., and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1558800698.

[4] TOP500 Supercomputers. [Online]. `https://www.top500.org/lists/top500/2021/06/`

[5] Green500 Supercomputers. [Online]. `https://www.top500.org/lists/green500/2023/06/`

[6] CUDA toolkit - free tools and training. [Online]. `https://developer.nvidia.com/cuda-toolkit`

[7] OpenCL - the open standard for parallel programming of heterogeneous systems [Online]. `https://www.khronos.org/opencl/`

[8] Heterogeneous Device Programming Using Sycl. [Online]. `https://www.khronos.org/sycl/`

[9] Kokkos: Performance Portability Programming for Manycore Systems. [Online]. `https://https://kokkos.org/`

[10] Medina, David S., Amik St-Cyr, and T. Warburton. ”OCCA: A unified approach to multi-threading languages.” arXiv preprint arXiv:1403.0968. 2014 Mar 4.

[11] Steuwer, Michel, and Sergei Gorlatch. ”SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems.” In *Parallel Computing Technologies*, pp. 258-272. Springer, Berlin, Heidelberg, 2013.

[12] Ernstsson, August, Lu Li, and Christoph Kessler. ”SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems.” *International Journal of Parallel Programming* 46, no. 1 (February 1, 2018): 62-80.

[13] OpenMP Application Program Interface Version 4.0 [Online]. `https://www.openmp.org/`

[14] The OpenACC API Specification for Parallel Programming. `https://www.openacc.org/`

[15] Karimi, Kamran. ”The Feasibility of Using OpenCL Instead of OpenMP for Parallel CPU Programming.” arXiv preprint arXiv:1503.06532. 2015 Mar 23.

[16] Karimi, Kamran, Neil G. Dickson, and Firas Hamze. ”A Performance Comparison of CUDA and OpenCL.” arXiv preprint arXiv:1005.2581. 2011 May 16.

[17] Mohammed, Ammar, and Rania Kora. "A comprehensive review on ensemble deep learning: Opportunities and challenges." *Journal of King Saud University - Computer and Information Sciences* 35, no. 2 (2023): 757-774.

[18] Scopus, 2023. scopus preview `https://www.scopus.com/`

[19] Breiman, Leo. "Random forests." *Machine learning* 45, no. 1 (2001): 5-32.

[20] Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., & Ha, D. "Deep learning for classical japanese literature". arXiv preprint arXiv:1812.01718. 2018 Dec 3.

[21] Broquedis, François, et al. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications." *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Feb 2010, Pisa, Italy.