ADVANCED COMPUTER ARCHITECTURE OPTIMIZATION FOR MACHINE LEARNING/DEEP LEARNING

Shefqet Meda [1], ***Ervin Domazet*** [2]

[1] Department of Electronics & Telecommunications Engineering, Faculty of Engineering, Canadian Institute of Technology, Albania, shefqet.meda@cit.edu.al, ORCID:0000-0003-3654-6606

[1] Department of Computer Engineering, Faculty of Engineering, International Balkan University, Skopje, North Macedonia, meda.shefqet@ibu.edu.mk, ORCID:0000-0003-3654-6606

[2] Department of Computer Engineering, Faculty of Engineering, International Balkan University, Skopje, North Macedonia, ervin.domazet@ibu.edu.mk, ORCID:0000-0001-7743-469X

Abstract

The recent progress in Machine Learning (Géron, 2022) and particularly Deep Learning (Goodfellow, 2016) models exposed the limitations of traditional computer architectures. Modern algorithms demonstrate highly increased computational demands and data requirements that most existing architectures cannot handle efficiently. These demands result in training speed, inference latency, and power consumption bottlenecks, which is why advanced methods of computer architecture optimization are required to enable the development of ML/DL-dedicated efficient hardware platforms (Engineers, 2019). The optimization of computer architecture for applications of ML/DL becomes critical, due to the tremendous demand for efficient execution of complex computations by Neural Networks (Goodfellow, 2016). This paper reviewed the numerous approaches and methods utilized to optimize computer architecture for ML/DL workloads. The following sections contain substantial discussion concerning the hardware-level optimizations, enhancements of traditional software frameworks and their unique versions, and innovative explorations of architectures. In particular, we discussed hardware including specialized accelerators, which can improve the performance and efficiency of a computation system using various techniques, specifically describing accelerators like CPUs (multicore) (Hennessy, 2017), GPUs (Hwu, 2015) and TPUs (Contributors, 2017), parallelism in multicore architectures, data movement in hardware systems, especially techniques such as caching and sparsity, compression, and quantization, other special techniques and configurations, such as using specialized data formats, and measurement sparsity. Moreover, this paper provided a comprehensive analysis of current trends in software frameworks, Data Movement optimization strategies (A.Bienz, 2021), sparsity, quantization and compression methods, using ML for architecture exploration, and, DVFS (Hennessy, 2017),, which provides strategies for maximizing hardware utilization and power consumption during training, machine learning, dynamic voltage, and frequency scaling, runtime systems. Finally, the paper discussed research opportunity directions and the possibilities of computer architecture optimization influence in various industrial and academic areas of ML/DL technologies.

The objective of implementing these optimization techniques is to largely minimize the current gap between the computational needs of ML/DL algorithms and the current hardware's capability. This will lead to significant improvements in training times, enable real-time inference for various applications, and ultimately unlock the full potential of cutting-edge machine learning algorithms.

Keywords: Computer Architecture Optimization, Machine Learning, Deep Learning, Parallelism, Sparsity, Data Movement Optimization, Quantization, Compression, Software Framework Optimization, DVFS, TPU, CPU, GPU, TensorFlow, Pytorch.

1. INTRODUCTION

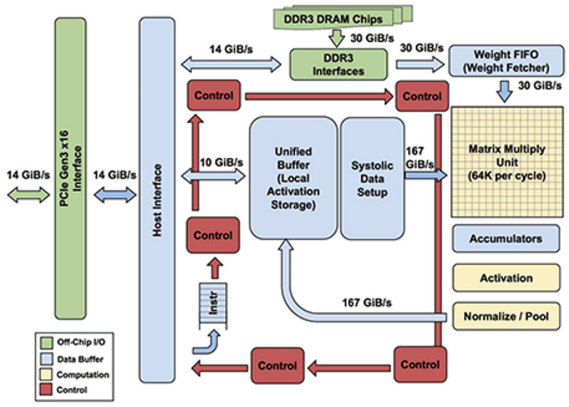
Machine Learning/Deep Learning are among the rapidly growing and most transformative fields of science. Image recognition, natural language processing, health care, autonomous vehicles, and several other fields of applications have seen a massive progress in recent years. This progress is driven by the development of increasingly complex models capable of tackling ever-more challenging tasks. As the scale and complexity of ML/DL applications continue to grow, there is an increasing demand for efficient and scalable computing infrastructure to support these workloads. However, these advancements are often bottlenecked by the limitations of traditional computer architectures. These architectures, designed for general-purpose computing, struggle to efficiently handle the massive computational demands of modern ML/DL algorithms. The inefficiency manifests in several ways:

*Corresponding author:

- **Training takes too long.** Entire teams are spending days or weeks training models to find better hyperparameters and experiments in fine print.
- **High Inference Latency** (Hennessy, 2017),. Deploying these models on resource-constrained devices, like mobile phones and embedded systems, is often impractical due to the high computational requirements for real-time inference.
- **Significant Power Consumption.** Large-scale training of complex models incurs substantial energy costs, raising concerns about sustainability and cost-effectiveness. To solve these challenges, we shall focus on advanced computer architecture optimization for ML and DL. This research theme encompasses specialized hardware designs and software techniques that target ML/DL applications' unique characteristics. By optimizing the underlying hardware and software infrastructure, we consider the following objectives-to be achieved:

  - **Accelerate and optimize training speeds.** This involves reducing the time required to train complex ML/DL models, which speeds up the development process and allows for more iterative experimentation.
  - **We will also aim to develop real-time inference capabilities**. This will allow us to deploy ML /DL models to inferences close to the edge devices through faster, less efficient hardware, thereby enabling a greater range of AI applications on - devices.
  - **Finally, we will optimize the power efficiency of ML/DL models.** This last objective seeks to reduce power consumption needed for training and inference, which will make it more feasible and cost-effective to deploy ML/DL models.

This paper discusses the techniques of advanced computer architecture optimization for ML/DL. We will discuss the specific challenges faced with traditional architectures, analyse relevant hardware and software optimization strategies, and explore the potential benefits of these techniques. We will focus on techniques and strategies that enhance performance, energy efficiency, and scalability. The optimization of computer architecture encompasses a wide range of topics, except above mentioned, including hardware design, software frameworks, and system-level optimizations, all of which are critical for achieving optimal performance in ML/DL workloads. Specialized hardware accelerators (Sze, Chen, Yang, & Emer, 2017) such as CPUs (multicore), GPU, TPU, and custom ASICs have shown more efficiency on ML/DL computation. These accelerators are specifically designed to accelerate the matrix and tensor operations that are prevalent in ML/DL applications further neural network computations, enabling faster model training and inference. They have made it feasible for model training to be faster.



TensorFlow processing unit architecture. Source: Google (Google, 2017)

In addition to hardware optimizations, software frameworks and runtime systems also contribute significantly to optimizing computer architecture for ML/DL applications. While frameworks such as TensorFlow (TensorFlow, 2024), PyTorch (Migacz, 2024), and Apache MXNet (apache.org, 2024) offer higher-level abstractions and API to build, train, and deploy neural network models (Sze, Chen, Yang, & Emer, 2017), enabling faster model training and inference.

Runtime systems are also optimized to run ML/DL workloads on diverse hardware platforms. Moreover, machine learning techniques for architecture exploration have shown promise in optimizing computer architecture for ML/DL applications. Specifically, the techniques using supervised learning, reinforcement learning, and Bayesian optimization (Tony Pourmohamad, 2021) have proved very effective in exploring the design space and discovering the optimal configuration to optimize performance and power efficiency. In conclusion, advanced optimization of computer architecture for ML/DL is a multidisciplinary field that mixes expertise in hardware, software, and machine learning. Through state-of-the-art in computer architecture optimization, researchers and engineers can unlock incredible possibilities for ML/DL technology that results in groundbreaking AI advancements and applications across a range of fields. This paper has tried to provide a clear picture of this exciting and rapidly growing field and its recent progress and future work.

2. HARDWARE OPTIMIZATION:

ACCELERATORS – CPU, GPU, TPU

Hardware optimization is a technique used to enhance the performance and efficiency of computing systems (Larkin Ridgway Scott, 2021). It involves increasing performance of accelerators like GPUs, TPUs and FPGA, considering designing to handle heavy workloads more efficiently. Traditional

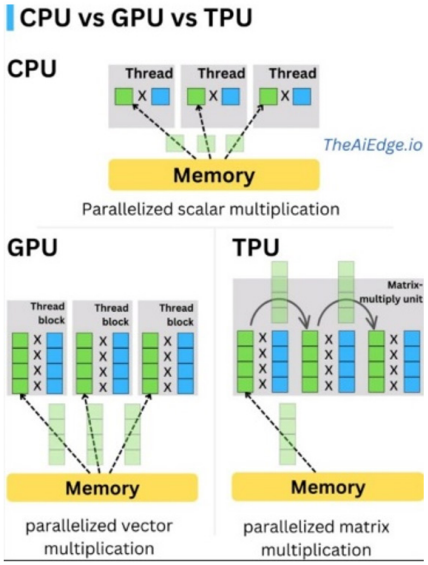
ones have their limitations include: CPUs not being efficient in support for upsized vectorized instructions such as AVX, along with efficient memory access to handle large datasets and model parameters. Limitations in parallelism is slowing down performance (Hennessy, 2017), in some of the highly parallel tasks like Matrix multiplications. Most of them are not fully optimized for some ML/DL operations and the high-power consumption among others.
In the below Pseudocode 1, an example of how a TPU might function to achieve matrix multiplication is presented.

```
// Pseudocode just for example
void matrixMultiply(float* X, float* Y, float* Z) {
   // Load matrices X and Y into on-chip memory
  MXE_Load(X, Y);

   // Perform matrix multiplication using MXE hardware
  MXE_Multiply();

  // Store result matrix Z in on-chip memory
  MXE_Store(Z);
}
```

Pseudocode 1: *TPU pseudocode for matrix multiplication.*

There are many strategies and efforts to optimize hardware. After we have carefully studied them, we have grouped of the most optimal ones that can be considered applied not specifically to solve specific problems and tasks but to cope with the processing, analysis, and output of significant amounts of data for applications of Artificial Intelligence (ML/DL):
- Parallelism (Hennessy, 2017),. This involves utilizing the parallel processing capabilities of the hardware architecture. It enables multiple processing units within the core, CPU to compute multiple independent instructions; other techniques include SIMD and distributed computing.
- Vectorization (Hwu, 2015). It considers vector processing units such as SSE (Streaming SIMD Extensions) and AVX in CPUs and GPUs to perform operations on more than one data simultaneously. It sends one operation to the core simultaneously sent to multiple data elements.
- Cache Optimization (Hendrik Borghorst, 2019). Maximizing cache utilization to reduce memory access latency and improve data locality. Techniques include cache blocking, prefetching, and data alignment.
- Instruction-Level Optimizations. Optimizing code to make the best use of instruction pipelines, branch prediction, and instruction scheduling.
- Memory Optimization (Joo-Young Kim, 2022). This reduces memory consumption and accesses, to reduce stalls and improve overall performance.

- Data Layout Optimization. Organizing data structures and memory layouts to optimize for cache coherence, spatial locality, and access patterns.
- Algorithmic Optimization. Choosing or designing algorithms that are well-suited for the underlying hardware architecture, minimizing unnecessary computations and memory accesses.
- Power Efficiency Build. It is especially critical to mobile devices and some limited energy environments, particularly when predicting systems in real-time.
- Profiling and Benchmarking. This process allows optimizing critical codes by observing or finding the bottlenecks of performance.
- Hardware-Specific Optimization. Specific features such as specific to CPU, GPU, or TPU architectures, specialized instruction sets, tensor cores, or memory hierarchies.
- Furthermore, the below flowchart explains hardware specialization. While traditional CPUs serve as effective general-purpose chips, TPUs or AI accelerators have dedicated characteristics based on the specific requirements for DL



CPU, GPU, or TPU architectures. Source: eitc.org (eitc.org, 2023)

- **Compiler Optimizations.** Making use of compiler optimizations to automatically transform code for better performance, including loop unrolling, function inlining, and auto-vectorization.
- **FPGA and ASIC Optimization:** For specialized hardware like FPGAs and ASICs, optimization involves hardware design at a lower level, focusing on logic utilization, routing efficiency, and clock frequency (Wijtvliet, 2019).
We can recommend another approach like

algorithms specifically for FPGA execution, considering the parallel and pipelined nature of FPGAs. An example is conclusions of a group of researchers for accelerating the learning process of CRBM (Restricted Boltzmann Machine) with FPGAs and OpenCL, and conducting an extensive scalability study for different model sizes and system configurations (Zoran Jakšić, 2020).

- **Quantization and Pruning:** Additional optimizations like for neural network accelerators such as TPUs include Quantization (Jacob, 2018) and pruning accelerate computation by reducing the precision of weights and activations of weights and removing redundant connections of the neural network. Give developers exercises to apply these enhancements to hardware accelerators across all computing platforms.

3. PARALLELISM - MULTICORE ARCHITECTURES

Utilizing parallelism is a fundamental consideration of computer architecture optimization for ML/DL applications, especially when it comes to multicore architectures. Enabling computers to utilize parallelism is critical for expediting the compute-intensive nature of ML/DL workloads and realizing the desired performance and efficiency levels (Larkin Ridgway Scott, 2021). There are several restrictions connected with traditional computers when it comes to parallelism: they were not designed to capitalize on parallelism in the context of ML/DL workloads by parallelizing matrix operations, convolutions, and other operations conducted by neural networks across various CPU cores.

Such multicore architectures (Hennessy, 2017), enable the following benefits, over traditional one's, for ML/DL on CPUs:

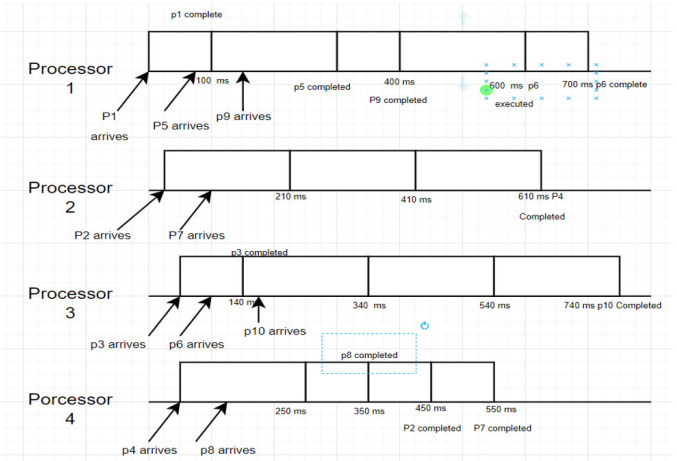
- faster training and inference,
- added scalability,
- parallel processing power.

Multicore CPUs put more cores on the same chip, essentially allowing them to run more instructions at once. This leads to better performance for ML/DL tasks that can be subdivided into separate computations.

- **Core specialization:** Modern cores are rich in dedicated units, such as vector processing units. These types of units can run the same operation on multiple data elements simultaneously. This is very beneficial for ML/DL workloads that involve a lot of vector or matrix computations.

- **Multiple cores that can run in parallel:** Multiple cores on a CPU can operate at the same time. When subtasks of certain instructions can be run at the same time, multicore CPUs will show an excellent parallel processing speedup. For instance, ML/DL heavily rely on linear algebra and matrix multiplication, which can be trivially partitioned per sub-step. To implement CPU, all the processor splits the workload per step and runs multiple cores to speed up the whole calculation.

- **Thread-level parallelism:** Modern CPUs support additional execution threads along cores, so-called hyper-threading or simultaneous multi-threading Further enhance parallelism because it keeps the cores busy even while waiting for data or cheesing on another thread. Below is an example of the application of quantization according to Round Robin in the multicore architecture and a part of the corresponding code. If the outputs for two different algorithms are compared, as we did, the superiority of quantization and multicore in execution is clearly observed.



Multicore architecture running threads and code

```
int main() {
 int n, quantum;
 cout << "Enter The Number of Processes: ";
 cin >> n;
 cout << "Enter The Time Quantum: ";
 cin >> quantum;

 Process processes[n];
 cout << "Enter process details:\n";
 for (int i = 0; i < n; i++) {
  cout << "Process " << i + 1 << ":\n";
  processes[i].id = i + 1;
  cout << "   Arrival Time: ";
  cin >> processes[i].arrivalTime;
  cout << "   Burst Time: ";
  cin >> processes[i].burstTime;
 }

 calculateTimes(processes, n, quantum);
 calculateTurnaroundTime(processes, n);
 calculateWaitingTime(processes, n);

 cout << "\nRound Robin Scheduling Results:\n";
 printTable(processes, n);

 return 0;
}
```

- **Software optimization for Multicore Exploitation:** Be able to save the workload of multiple cores properly, the framework has to be appropriately designed. Software frameworks like TensorFlow or PyTorch use python Threading to subdivide the load of data on the CPUs. They chop small slivers of data, and the computation is about a model into smaller

clones of the data and add them to separate cores (sample above).

- Libraries such as OpenMP or Intel Threading Building Blocks (Ruud Van Der Pas, 2017): Provide tools for programmers to write code that takes advantage of multiple cores.

• **Data and Model Parallelism:** Both data and model parallelism techniques are enabled by multi-core architectures. Splits the training data into smaller batches and distributes them across multiple cores. Each core processes its assigned data independently, accelerating computations, especially for large datasets.

- Model parallelism divides the machine learning or deep learning model across cores. For extremely large and difficult models which would not fit on a single core, this is necessary. However, it must be done with caution, since the various cores must collaborate to ensure the model changes are coherent.

- Simple code in Python:

```
# Pseudocode for data parallel training with model parallelism on two GPUs
def train_step(data_batch):
# Split data batch across GPUs
data_batch_gpu1, data_batch_gpu2 = split_data_batch(data_batch)
# Forward pass on GPU 1 (first half of the model)
with tf.device("/gpu:0"):
activations1 = model_part1(data_batch_gpu1)
# Send activations from GPU 1 to GPU 2
activations1 = send_data(activations1, "/gpu:1")
# Forward pass on GPU 2 (second half of the model)
with tf.device("/gpu:1"):
output = model_part2(activations1)
# Backpropagation and gradient accumulation (simplified)
return output
# Training loop
for epoch in range(num_epochs):
for data_batch in training_data:
output = train_step(data_batch)
# ... update model weights
```

Pseudocode 2: *For data parallel training with model parallelism on two GPUs*

- Software Optimization Frameworks like TensorFlow and PyTorch are optimized in deployed on multicore architectures. Automatically divide data and models across cores for parallel processing. cores must communicate and collaborate on model changes.

3.1. Challenges and Considerations

While it is easy to understand the benefits of processing multiple tasks simultaneously, a variety of factors determine how effective such parallel execution can be. While cores can simultaneously perform in parallel, communication between them may be the bottleneck if they have to share data and any updates. Techniques such as gradient accumulation and efficient communication protocols are critical. Additionally, optimized memory access patterns are crucial to success.
Cores must be allowed to quickly and efficiently access whichever data they require from memory, and if this is not allowed or limited by memory bandwidth, the numbers of parallel operations cannot be optimized for faster performance. Finally, not all parts of an ML/DL application can be perfectly parallelized. Amdahl's law determines the maximum speedup due to parallelism. Since there will always be serial sections of the code that cannot be divided into parallel parts, the maximum speedup can only go so far. Memory bandwidth influences cannot be avoided; rather than intensified, these drawbacks are magnified.

Conclusion: so we can conclude that, conventional CPU architectures are vial in achieving the parallelism in ML/DL applications. When combined with several cores and specialty hardware and software representations, they lead to quicker trainand and recast times. Optimal algorithms and communication strategies will be key to ensure the best potential from these machines as core counts proceed to expand.

We suggest that the use of high-level languages, or otherwise modern programming languages and open source platforms would be another opportunity to enable researchers and developers to provide optimal solutions for ML/DL applications and increasing performance.

4. DATA MOVEMENT OPTIMIZATION - DATA PREFETCHING, MEMORY HIERARCHY, CACHING

Traditional computer architectures can have limitations in data movement that can impact the performance and efficiency of Machine Learning (ML) and Deep Learning (DL) workloads. Here's are listed some aspects how these limitations affect ML/DL specifically:

Data Prefetching. In traditional architectures, software prefetching is used, where the processor asks for the data from memory explicitly before its need. This can introduce latency if the request isn't anticipated correctly or if there's concurrency for memory bandwidth. In ML/DL, algorithms often exhibit predictable access patterns, repeatedly

accessing specific data elements. Inefficient prefetching can lead to situations where the needed data isn't readily available when the processing unit requires it, causing delays.

Memory Hierarchy Bottlenecks (Joo-Young Kim, 2022). Traditional architectures utilize a hierarchy of memories with varying speeds and capacities (e.g., registers, cache, main memory, disk). Accessing data from slower levels in the hierarchy takes longer. ML/DL algorithms often work with large datasets. If the frequently accessed data resides in slower memory, retrieving it for each operation can significantly impact performance.

Another limitation is - Storage I/O Bottlenecks. Traditional architectures use mechanical hard disk drives or even slower to use this data for storing load from memory, and the storage I/O bottleneck can occur in ML/DL systems. The problem with storage I/O bottlenecks is that systems require huge data and input/outing this big data makes the system slow. There are slow read/write speeds and high latency in storage devices that can prevent ML/DL storage from efficient work, especially when loading data and preprocessing it. Therefore, these bring us to the following limitations that bottleneck ML/DL workloads:

Increased Latency and Reduced Throughput.
Delays caused by inefficient prefetching, accessing slower memory levels, and cache misses all contribute to longer execution times for training and inference. Limited data movement capabilities can restrict the amount of data processed per unit time, hindering overall performance.
What we can do to optimize and avoid these limitations in performing with ML/DL apps includes:

Data Prefetching:
- Software Prefetching: by predicting and prefetching data into the cache before accessed by the computation to reduce latency, the computation can be overlapped with data movement.
- Hardware Prefetching: Exploit hardware mechanisms of modern processors to automatically prefetch data into cache according access patterns observed during runtime.
- Spatial and Temporal Prefetching: Prefetch data blocks that are spatially or temporally related to the currently accessed data, such as prefetching neighboring elements in arrays or prefetching data expected to be accessed in future iterations.

Memory Hierarchy Optimization:
- Cache Awareness (Hendrik Borghorst, 2019): design algorithms and data structures that can maximize cache usage and take advantage of spatial and temporal locality.
- Cache blocking: Partition data into blocks that fit into the cache, enabling reuse of cache lines and reducing the frequency of cache misses.
- Cache Replacement Policies: Use cache replacement policies (e.g., LRU, LFU, Optimal) that prioritize keeping frequently accessed data in the cache to minimize cache thrashing.
- Cache Compression: compress the stored data in the caches to increase the capacity of the caches and lessen the impact of cache pollution.

Caching (Hendrik Borghorst, 2019):
- Data Reuse: identify and leverage the opportunities to reuse intermediate results and computation to minimize data movement across memory levels.
- Result Caching: considering that computations in DNN are iterative, the results can be stored in memory to prevent repetitive computations and lower latency.
- Model Caching: the trained models and parameters can be cached in memory to reduce the load on the main memory and enhance the performance. The example illustrated below is an application of Fuzzy Logic in cache optimization by applying dynamic membership. As can be seen from the result, we have an execution time almost twice as short.

RESULT AND PERFORMANCE EVALUATION

Process ID	Arrival Time (ATI)	Burst Time (BTI)	Static Priority (PTI)	Dynamic Priority (DPI)	Deadline (D)
P1	0	18	1	0.136	53
P2	0	2	3	0.894	18
P3	0	1	2	0.95	13
P4	0	4	6	0.79	35
P5	0	3	5	0.84	22
P6	0	12	11	0.917	15
P7	0	13	7	0.587	53

Comparison Table

Algorithm	Average Waiting Time	Average Turnaround Time	Average Response Time
Priority Algorithm	23.86	31.43	23.86
IFCS	14.86	22.43	14.85
AFCS	14.71	22.43	14.57

Fuzzy Logic in cache optimization with dynamic membership

Memory access patterns.
- Sequential Access Optimization: Arrange data structures and access patterns to optimize for sequential memory access, minimizing the overhead of random memory accesses.
- Data Layout Optimization: Organize data in memory to improve cache locality and alignment

with the underlying hardware architecture, such as using row-major or column-major layouts for matrices depending on access patterns.
• Strided Access Optimization: Minimize the stride (distance between consecutive elements) in memory access patterns to improve cache utilization and reduce memory access latency.

Data Compression and Encoding:
- Lossless Compression: Compress data before storing it in memory or transferring it between levels of the memory hierarchy to reduce storage requirements and bandwidth usage.
- Lossy Compression: Apply the lossy compression technique to sacrifice precision for better memory bandwidth utilization through varied sparsity methods for ML/DL such as quantization and low-precision data representation.

Advantages of Data Movement Optimization:
- Decrease Training and Inference Time: By minimizing wasted time waiting for data movement, these techniques can significantly accelerate the overall training and inference processes in ML/DL applications.
- Improved Resource Utilization: Efficient data movement allows for better utilization of processing power by reducing idle time due to data access delays.

5. SPARSITY: SOME SPECIAL TECHNIQUES, LIKE USING SPECIALIZED DATA FORMATS AND HARDWARE UNITS.

- Sparsity (Brandon Reagen, 2017) is essential in the computer architecture of Machine Learning and Deep Learning applications because it refers the presence of a large number of zeros in the data being processed. Some of the most relevant limitations from several issues that the traditional computer has when performing ML/DL apps are as follows:
- Sparsity Handling: Generally, sparsity is not directly addressed in traditional computer architecture design because most algorithms and hardware units are designed to work with the dense data format efficiently without specific optimizations for sparse data.
- Data Formats: Classic computer architectures support standard numerical data format, like float-point or integer, and do not directly support sparse data representation.
Hardware units: Sparse data is leveraged for many high-performance applications because the units in classic computer architecture, including CPUs and GPUs, are optimized for dense computations; they are not equipped with specialized units optimally designed for sparse data. Sparse matrix-matrix multiplication and other sparse operations are performed on these general-purpose hardware units, providing suboptimal performance due to the sparse data. Specialized techniques yield significant performance improvements because of the sparsity of the data.
The following are some of the aspects of computer optimization for ML/DL applications that use sparsity:

Sparse data formats: Compressed sparse formats. These arise through the use of compressed formats such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) to represent sparse matrices efficiently. They only store the non-zero elements, with their indices, yielding reduced storage requirements and better cache locality.

Sparse tensor representation: The data stored in these structures will be sparse, so the storage pattern must account for this. Design specialized data structures to optimize the storage and access patterns because sparse data structures such as graphs and embeddings are seen frequently in ML/DL models

Sparse matrix operations: Sparse matrix multiplication. It can be performed by developing optimized algorithms and units for sparse matrix multiplication, as this is fundamental to ML/DL applications. Memory access reduction and computational complexity can be achieved by explicitly identifying how to achieve modular multiplication based on sparsity.
Sparse tensor operations. We can extend hardware accelerators to support the computation of sparse tensors as the representations and operations involved in many ML/DL applications are seen as tensors.

Sparsity-aware hardware acceleration: Sparse tensor cores. Design specialized hardware units capable of efficiently handling sparse tensor operations, exploiting sparsity to reduce computational and memory requirements. These units can perform computations only on non-zero elements, improving energy efficiency and performance for sparse ML/DL models. Sparsity-Aware Memory Systems in developing memory architectures and controllers optimized for handling sparse data access patterns, with mechanisms for efficient storage and retrieval of non-contiguous data elements in sparse tensors.

Quantization and Pruning: Quantize models or model activations for sparse representation and reduce the precision for zero and near-zero values while maintaining the accuracy of non-zero elements. Sparse quantization can considerably reduce memory bandwidth and reduce memory storage overheads required to support large-scale ML/DL models. Remove redundant or insignificant

weights from neural network models to obtain sparse weight matrices. Sparse weight matrices can be further sparsified using special storage formats and hardware units designed specifically for sparse data representation. Moreover, sparse weight matrices will reduce the memory overhead and the computational complexity of their processing during inference.

Dynamic Sparsity: Develop algorithms and hardware mechanisms that can detect sparsity and use the acquired knowledge durably at runtime. Dynamic sparsity algorithms and methods may detect variation in sparsity and adjust the computational approach based on changes in input data, sparsity models, and other factors. Integrating specialized data formats, hardware units and using sparsity – aware algorithms in computer systems can significantly accelerate ML/DL applications, improve their energy efficiency, and enhance scalability.

6. QUANTIZATION AND COMPRESSION.

The increasing size and complexity of machine learning based algorithms and models create two main challenges: it may not be possible to deploy an ML/DL model on the resource – constrained devices, and training such big models may be slow. The number of bits in which the data are represented in the model is reduced using quantization. In a traditional setting, high precision formats like 32-bit floating point numbers are used to represent the weights and activations of a model. Techniques considered in quantization include being able to represent these values in lower precision formats such as 8-bit integers with minimal effects on the model's accuracy. The model's memory footprint is significantly reduced, easing the storage on devices with fewer memories. During training, it reduces the storage knowledge footprint. Compression techniques are considered to work in tandem with quantization as quantization is focused on reduction of the redundant bits within an encoding while compression thrives from a reduction in information in the representation. To remove redundancy information within the representation, a for example may be removed. Examples of compression approaches include pruning, where unimportant figures are removed from the model or knowledge distillation, where a shallower model is trained on the end weights to mimic the complex model. The limitations of the traditional computers that affect quantization and compression (Jacob, 2018) in ML/DL applications are: Compression and quantization are unfamiliar concepts in classic computer architecture. In classic computer architecture, compression algorithms aim at storage and transmission of the data. The data can be files, as in Huffman compression, or progression, as seen in LZ compression techniques. Sounds and images are quantized to reduce the size before being transmitted since this approach savesbits. Data quantization and compression are techniques used in many computer ways in storage, transmission, and retrieval. These two techniques have not been adequately studied since accuracy in representation is less of concern. The critical issue is reduced bit storage while assured of minimal to no information or precision loss. A crucial area that further needs to be researched is promoting zero or one firing without compromising the model since all that does is alters the model behaviour. Identifying this optimal precision or compression level is a challenge if the problem area or the algorithms do not dictate the amount of information retained in a unit.

Techniques in ML/DL apps.

- quantization and compression algorithms to be used to reduce the precision of model parameters and activations thereby decreasing memory usage and computational complexity

- while preserving model accuracy techniques like weight pruning, quantization, and low-rank factorization are commonly employed to compress neural network models for deployment on resource-constrained devices or for efficient distributed training.

Benefits

- Reduced memory footprint. Lower precision data representations require less storage space, enabling the training of larger models on limited hardware resources
- Faster training and inference. Reduced precision leads to faster computations during the training and inference stages
- Lower power consumption. Lower power consumption due to smaller data footprints and potentially less complex computations makes these techniques attractive for mobile and embedded devices

Challenges

- Finding the Right Balance: Determining the optimal level of quantization or compression requires careful consideration, as too much reduction in precision can significantly impact model accuracy.
- Framework and Hardware Support: Not all frameworks and hardware platforms offer the same level of support for various quantization and compression techniques.
- Potential Need for Algorithm Adjustments: Depending on the technique used, algorithms within the neural network might need adjustments to maintain accuracy with lower precision data.

7. SOFTWARE FRAMEWORK OPTIMIZATION

Software framework optimization is a critical aspect of computer architecture optimization for machine learning (ML) and deep learning (DL) applications, facilitating efficient execution of ML/DL workloads on diverse hardware platforms while maximizing performance and scalability. ML/DL frameworks provide high-level abstractions and APIs for developing, training, and deploying neural network models, abstracting away hardware-specific details and enabling portability across different architectures. (Abadi, 2016). This kind of optimization serves to support the efficient execution of machine learning and deep learning workloads on any hardware platform and to maximize ML/DL task performance and out-of-the-box scalability. ML/DL frameworks are high-level abstractions and APIs that let users create, train and deploy neural network models without worrying about underlying hardware-specific details. However, before listing what might be optimized to enhance the performance of traditional computers, let's outline what such computers imply prevents such optimizations:

- Software frameworks. In classic computer architecture, software frameworks are developed for any general-purpose computing work that may apply to many different applications. They include an operating system, a programming language and libraries.
- Limited Hardware acceleration (Vinh Nguyen, 2020). Hardware acceleration to leverage optimizations for explicit instruction sets and memory-specific optimizations is mostly not used by this framework.
- Programmer Responsibility. More of the burden of optimization code is on the program, using techniques like manual vectorization or optimization of cache sometimes. Lower the frameworks offer much lower to no leverage of hardware-specific optimizations for specific instruction set and memory hierarchy.
- Lower-Level Control. The frameworks provide greater control over memory and instruction schedule but require more effort from the programmer to optimize.

Now let's see what can be optimized under given constraints.

- Optimizing ML/DL frameworks to leverage hardware-specific features and optimizations, such as specialized instructions, memory hierarchy, and parallelism capabilities. Frameworks like TensorFlow, PyTorch, Keras, and Apache MXNet incorporate backend optimizations to exploit hardware acceleration, enabling efficient execution of neural network computations on CPUs, GPUs, and specialized accelerators like Tensor Processing Units (TPUs). These optimizations include autotuning techniques, kernel fusion, and memory layout optimizations tailored to specific hardware architectures, ensuring optimal performance across a wide range of hardware platforms. They offer High-Level Abstractions, Automatic Optimization, and Focus on Parallelism
- Focus on minimization of overhead and improvement of efficiency in data processing and model inference pipelines. This is possible due to for example model quantization, tensor slicing, or even asynchronous execution which allows for efficient memory usage and pipelining of calculations. In both cases, latency is minimized and throughput is maximized for ML/DL workflows.
- Distributed training and inference are also considered to require minimal overhead with, for example, parameter servers or communication primitives, which allow distributed computing to be scalable and fault-tolerant. This extends to runtime systems and libraries supporting ML/DL applications and offering low-level abstractions and optimizations. Libraries such as cuDNN, or math Kernel Library MKL-DNN known for optimized implementations of common neural network operations. They utilize hardware-specific optimizations and parallelism for GPUs, and CPUs that can be used during ML and DL computational tasks to improve the efficiency and performance of neural networks.

8. DYNAMIC VOLTAGE AND FREQUENCY SCALING (DVFS)

DVFS is a power management technique that adjusts the voltage and frequency of a processor dynamically based on workload requirements to optimize energy efficiency while maintaining performance.

Let's see limitation of Dynamic Voltage and Frequency Scaling (DVFS) in traditional computer architecture before optimization.

- DVFS is commonly implemented at the operating system level or through hardware features provided by the CPU. The operating system dynamically adjusts the CPU voltage and frequency based on workload characteristics, system power constraints, and thermal considerations. Needs to reduce power consumption during periods of low workload while maintaining adequate performance during periods of high workload.
- DVFS in classic computer architecture is used in a variety of applications, including general-purpose computing tasks, web browsing, office productivity, and multimedia playback. We have to been focused on improving energy efficiency and prolonging battery life in mobile devices, laptops, and servers for better performance.

Considerations for Optimization in case of ML/DL Applications:

- In architectures optimized for ML/DL applications,

DVFS may be implemented at multiple levels of the hardware and software stack, including specialized hardware accelerators (e.g., GPUs, TPUs) and ML/DL frameworks. DVFS techniques are used to dynamically adjust the voltage and frequency of processing units (e.g., CPU cores, GPU cores) based on the computational demands of ML/DL workloads.
- Optimization Goals: DVFS in ML/DL-optimized architectures aims to maximize the performance and efficiency of neural network computations by dynamically adjusting the voltage and frequency of hardware accelerators and processing units. The goal is to balance computational throughput with energy consumption, ensuring optimal performance for ML/DL workloads while minimizing power consumption.
- Applications: DVFS in ML/DL-optimized architectures is used to accelerate model training and inference tasks in neural network frameworks such as TensorFlow, PyTorch, and MXNet. Optimization efforts focus on improving the efficiency of ML/DL workloads on specialized hardware accelerators (e.g., GPUs, TPUs) and maximizing the throughput of neural network computations.
- Some frameworks might integrate with DVFS controls to dynamically adjust processing power based on the specific workload characteristics of the ML/DL task.
Challenges when we go for optimization of DVFS:
- Thermal Constraints: ML/DL workloads can be computationally intensive, leading to heat generation. DVFS can be used to manage thermal throttling by dynamically adjusting voltage and frequency to stay within safe operating temperatures.
- Impact on Accuracy: Very aggressive DVFS settings might introduce slight rounding errors in some ML/DL models. Careful calibration is needed to balance power savings with acceptable accuracy loss.

9. UTILIZING MACHINE LEARNING FOR DESIGN.

The ever-growing complexity of Machine Learning (ML) and Deep Learning (DL) workloads demands continual innovation in computer architecture design. Traditionally, this exploration process has relied on manual design and simulation, which can be time-consuming and computationally expensive. Advanced computer architecture optimization leverages machine learning (ML) for design space exploration, accelerating the discovery of efficient hardware for ML/DL tasks. The design space for ML/DL architectures is vast, encompassing numerous parameters like memory organization, processing unit configurations, and interconnect structures. Manually evaluating all potential configurations is impractical. Simulating the performance of different architectures for various workloads is computationally expensive, hindering the exploration process.
ML techniques offer a powerful approach to address these challenges. By employing techniques like reinforcement learning or evolutionary algorithms, we can automate the exploration of the architecture design space.

These algorithms can:
-Efficiently Search the Design Space: ML models can be trained on historical data of architecture performance and workload characteristics. This allows them to identify promising design configurations and prioritize them for further evaluation through simulation or hardware prototyping.
-Learn from Experience: As the exploration progresses, the ML model can learn from the performance of evaluated architectures. This allows it to refine its search strategy and focus on more promising design directions over time.
-Benefits of ML-driven Exploration: Faster Design Exploration: ML can significantly accelerate the discovery of efficient architectures compared to traditional methods. This allows for quicker adaptation to the evolving demands of ML/DL algorithms.
-Improved Design Quality: By exploring a wider range of configurations, ML can potentially identify more efficient architectures compared to what might be discovered through manual exploration.
-Data-Driven Design Decisions: The exploration process becomes data-driven, with decisions based on performance evaluations and learned relationships between architecture parameters and workload characteristics.

However, utilizing Machine Learning for Design Space Exploration also presents challenges:
-Data Quality: The effectiveness of the exploration process heavily relies on the quality and quantity of data used to train the machine learning models. High-fidelity simulations or real-world performance data are crucial for accurate exploration.
-Interpretability: Understanding the rationale behind the proposed architecture by the machine learning model can be challenging. This limits the ability to refine the exploration process or incorporate domain knowledge from human architects.

10. CASE STUDIES WHERE THESE OPTIMIZATION TECHNIQUES HAVE BEEN SUCCESSFULLY APPLIED

- Google's BERT Model Optimization
BERT model – popular state-of-the-art natural language processing model. The goal was to decrease memory size and make inference faster. Model size decreased by 60%, inference made 1.5x faster. Techniques used: sparsity handling to work with sparse tensor representations more effectively. quantization and compression techniques to reduce precision and memory usage.
- NVIDIA's GPU Optimization for Deep Learning (Vinh Nguyen, 2020)
Techniques employed: Hardware optimization: Accelerators – GPUs, Software Framework optimization: Using CUDA – Compute Unified Device Architecture.
NVIDIA improved its GPU architecture and CUDA software framework to be used effectively in deep learning. It resulted in reduction in training and inference time for deep learning models, more efficient use of the GPU, hence savings. It used accelerated matrix multiplication to exploit the matrix operations that power deep learning. Optimization of the CUDA software framework to provide edge execution of neural network operations.
- Qualcomm's Snapdragon AI Engine used Hardware Optimization: AI accelerators (Hexagon DSP, Adreno GPU). Enabled on-device AI capabilities for tasks such as image recognition, natural language processing, and object detection. Improved power efficiency and reduced latency for AI tasks on mobile devices. Applied Leveraged specialized AI accelerators (Hexagon DSP, Adreno GPU) for efficient execution of neural network computations. Optimized memory access and caching to reduce latency and improve performance.
- Tesla's Autopilot Hardware 3. Used Hardware Optimization - Custom hardware (Tesla FSD Chip). It enabled real-time processing of sensor data and AI-based decision-making for autonomous driving tasks.

11. CONCLUSION AND FUTURE CONSIDERATIONS

In conclusion, computer architecture optimization for machine learning and deep learning applications is essential for realizing the full potential of ML/DL workloads by enabling the execution of computation and neural network computations. As discussed in this paper, there are multiple ways that we optimize computer architecture to meet the needs of ML/DL workloads. These include hardware-level optimizations, software framework enhancements, and other novel architectures for exploration. Hardware optimizations, including the use of GPUs, TPUs, and custom ASICs, and other accelerators, have been widely employed. Researchers have also developed multicore architectures to take advantage of parallelism and have also optimized data movement. Software frameworks have also grown and improved dramatically, enabling the scaling and execution of ML and DL models. Furthermore, the application of machine learning to explore architecture is speeding up the design process, allowing researchers to explore the design space much more quickly and, with the help of few compute resources, identify optimal hardware configurations. With the use of supervised learning, reinforcement learning, and Bayesian optimization (TonyPourmohamad,2021),learning-basedmethods enable researchers to effectively probe the variety of architectural parameters and results, leading to the creation of high-performance, energy-efficient computing systems suited to ML/DL workloads. As machine learning and deep learning applications thrive, and novel technologies and computational strategie s are introduced into sophisticated and varied hardware devices, the field of computer architecture optimization for ML/DL has ample room for growth. In practice, new technologies may include greater exploration of novel hardware architectures and optimized algorithms for ML/DL workloads, as well as incorporating hardware and software optimizations to ensure end-to-end efficiency across heterogeneous hardware. To summarize, computer architecture optimization is critical for the successful rise of machine learning and deep learning applications, allowing an efficient manner to execute ML workloads and DL across a broad array of hardware devices and use settings. By extending existing limits of knowledge in computer architecture optimization (Hennessy, 2017), researchers and engineers play an essential role in addressing the key computational constraints of the future of ML and DL applications and promoting AI's influence in advancement across different fields.
This work has established a solid foundation for progressing in advanced computer architecture optimization for ML/DL research. Hence, here, some promising pathways for future exploration have been highlighted (Gao, 2019):
- Co-optimizing software and hardware for even more effective optimizations.
- Exploring emerging technologies, such as neuromorphic computing, for hardware acceleration potential.
- Developing domain-specific hardware architectures designed for families of ML/DL algorithms' requirements.
- Improving Machine Learning for Design Space Exploration to generate more precise, interpretable outcomes.
- High-level languages, or otherwise modern programming languages (PATTERSON, 2019).
- Security and privacy considerations: Architectures

that incorporate hardware-based security measures to protect against the data privacy and model integrity threats facing modern models.
- Domain-specific architectures: Customizing hardware architectures to the needs of the specialized application domain such as image processing, NLP, recommender systems.
By pursuing these and related directions for further advancements in computer architecture optimization, we can enable the next stage of progress in ML/DL models and maximize their impact across the numerous fields.

References

A.Bienz, L. N. (2021). Modeling Data Movement Performance on Heterogeneous Architectures. *IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-7). Waltham, MA, USA: Institute of Electrical and Electronics Engineers Inc.

Abadi, M. B. (2016). TensorFlow: A System for Large-Scale Machine Learning. *12th USENIX Symposium on Operating Systems Design and Implementation* , 265–283.

apache.org. (2024). *APACHE MXNET:A FLEXIBLE AND EFFICIENT LIBRARY FOR DEEP LEARNING*. Retrieved from https://mxnet.apache.org/versions/1.9.1/

Brandon Reagen, R. A.-Y. (2017). Deep Learning for Computer Architects. In P. U. Margaret Martonosi, *Synthesis Lectures on Computer Architecture*. Springer Nature Switzerland.

Contributors. (2017, June 26). *In-Datacenter Performance Analysis of a Tensor Processing Unit*. Retrieved from https://arxiv.org/pdf/1704.04760

eitc.org. (2023, January). *cpu-vs-gpu-vs-tpu*. Retrieved from cpu-vs-gpu-vs-tpu: http://www.eitc.org/research-opportunities/photos1/cpu-vs-gpu-vs-tpu_012023a/image_view_fullscreen

Engineers, I. o. (2019). 25th IEEE International Symposium on High Performance Computer Architecture. *IEEE International Symposium on High Performance Computer Architecture*, p. 734 .

Gao, L. W. (2019). An Overview of Machine Learning in Computer Architecture. *Journal of Computer Science and Technology*, 709–731.

Géron, A. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*. O'Reilly Media.

Goodfellow, I. B. (2016). *Deep Learning*. Cambridge, Massachusetts, London: MIT.

Google. (2017, May 12). *An in-depth look at Google's first Tensor Processing Unit (TPU)*. Retrieved from https://cloud.google.com/blog/u/1/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Hendrik Borghorst, O. S. (2019). CyPhOS – A Component-Based Cache-Aware Multi-core Operating System. *Architecture of Computing Systems – ARCS 2019* (pp. 171–182). Springer, Cham.

Hennessy, J. L. (2017). *Computer Architecture-A quantitave Aproach* . Morgan Kaufman.

Hwu, W.-m. W. (2015). GPU Computing Gems. *Emerald Edition*.

Jacob, B. K. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2704-2713.

Joo-Young Kim, B. K.-H. (2022). *Processing-in-Memory for AI: From Circuits to Systems*. Springer Nature.

Larkin Ridgway Scott, T. C. (2021). *Scientific Parallel Computing*. Princeton University.

Migacz, S. (2024). *Performance Tuning Guide*. Retrieved from https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

PATTERSON, J. L. (2019). A New Golden for Computer Architecture. *Communications of the ACM*, 48-60.

Ruud Van Der Pas, E. S. (2017). *Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD (Scientific and Engineering Computation)*. MIT Press.

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017, November 20). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, pp. 2295 - 2329.

TensorFlow. (2024). *Theoretical and advanced machine learning with TensorFlow* . Retrieved from https://www.tensorflow.org/resources/learn-ml/theoretical-and-advanced-machine-learning

Tony Pourmohamad, H. K. (2021). *Bayesian Optimization with Application to Computer Experiments*. Springer.

Vinh Nguyen, T. G. (2020, June 18). *Optimizing the Deep Learning Recommendation Model on NVIDIA GPUs*. Retrieved from https://developer.nvidia.com/blog/optimizing-dlrm-on-nvidia-gpus/

Wijtvliet, M. W. (2019). Accelerating Machine Learning Workloads with OpenCL on FPGAs. *IEEE International Conference on Cluster Computing*.

Zoran Jakšić, N. C. (2020). A highly parameterizable framework for Conditional Restricted Boltzmann Machine based workloads accelerated with FPGAs and OpenCL. *Elseiver*, 201-211.

Glossary

Term	Definition
Machine Learning (ML)	A subfield of artificial intelligence (AI) that uses statistical techniques to give computers the ability to learn from data without being explicitly programmed.
Deep Learning (DL)	A subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data.
Computer Architecture Optimization	Techniques aimed at enhancing computer systems to better accommodate the computational demands and data requirements of machine learning and deep learning algorithms.
CPUs (Central Processing Unit)	The primary component of a computer that performs most of the processing inside a computer.
GPUs (Graphical Processing Unit)	A specialized electronic circuit designed to accelerate the creation of images in a frame buffer intended for output to a display.
TPUs (Tensor Processing Unit)	An application-specific integrated circuit (ASIC) developed by Google specifically for accelerating machine learning workloads.
Parallelism in Multicore Architectures	Utilizing multiple cores within a CPU to execute multiple processes simultaneously, thereby increasing computational efficiency.
Data Movement Optimization	Techniques aimed at improving the movement of data within a computer system, including caching, addressing sparsity, compression, and quantization.
Sparsity	A property of large data sets in which only a small percentage of the data is non-zero.
Compression	Techniques used to reduce the size of data, which can lead to improved storage and processing efficiency.
Quantization	The process of reducing the precision of a numerical representation, which can lead to reduced memory and computational requirements.
Software Framework Optimization	Enhancements made to software frameworks such as TensorFlow, PyTorch, and Apache MXNet to improve their performance and efficiency in handling machine learning and deep learning workloads
DVFS (Dynamic Voltage and Frequency Scaling)	A technique used in computer systems to optimize power consumption by adjusting the voltage and frequency of the CPU according to the current workload.
Runtime Systems	Systems optimized to run machine learning and deep learning workloads on various hardware platforms.

Supervised Learning	A type of machine learning where the algorithm learns from labeled data, making predictions and decisions based on that data.
Reinforcement Learning	A type of machine learning where an agent learns to make decisions by trial and error, receiving feedback in the form of rewards or penalties.
Bayesian Optimization	A method for optimizing objective functions that are expensive to evaluate, by building a probabilistic model of the objective function it to select the most promising points to evaluate.
ASICs (Application-Specific Integrated Circuits)	Customized microchips designed for a particular application, such as machine learning or deep learning.
Edge Devices	Devices that perform data processing at or near the source of data generation, rather than relying on a centralized data-processing warehouse or cloud storage.
API (Application Programming Interface)	A set of protocols, tools, and definitions that allow different software applications to communicate with each other.
Hardware Optimization	Techniques used to enhance the performance and efficiency of computing systems.
Accelerators	Specialized hardware components designed to handle heavy workloads more efficiently than traditional CPUs.
FPGA (Field-Programmable Gate Arrays)	Customizable integrated circuits that can be reprogrammed to implement different hardware functionalities after manufacturing.
Vectorization	Utilizing vector processing units such as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) in CPUs and GPUs to perform operations on more than one data simultaneously.
Multicore Architectures	Architectures that enable multiple processing cores on the same chip, allowing for parallel processing and improved performance for ML/DL tasks.
Data Prefetching	A technique used to improve memory access latency by predicting and loading data into cache before it is actually needed.
AVX (Advanced Vector Extensions)	Is a set of instructions that extend the capabilities of x86 processors from Intel and AMD. It allows them to perform operations on multiple pieces of data simultaneously, significantly improving performance for specific workloads.