

6 Hardware-Aware Execution

Efficient learning has been the focus of research for decades. Many studies explore various software/hardware techniques to improve the efficiency of learning process while preserving the accuracy of the derived learning models. Along with the various demands of applications nowadays, from simple like image recognition to advanced like fundamental steering, how to deploy learned models and execute them efficiently has been of key interests in the industry. Considering various resource constraints such as throughput, timeliness, and energy consumption imposed by the targeted scenarios and the adopted hardware platforms, most machine learning techniques, which often rely on high-performance computers and clusters, must be carefully redesigned to fulfill the assigned missions at edge devices while addressing the efficiency of resource usage.

To this end, we summarize in this chapter relevant research conducted in CRC 876, which is oriented to the awareness of hardware execution, and supplement two external contributions to cover a broader spectrum of this research direction. Unlike most existing techniques for executing neural networks on Field-Programmable Gate Arrays (FPGAs), we focus on the of learning, which is actually more computationally demanding (see Section 6.1). In addition, we exploit modern graphics processing units (GPUs) for efficient database query processing (see Section 6.2) and study how parallelization on multicore systems should be deployed for accelerating extreme multi-label classification (see Section 6.3). At the end, we present our RAMBO framework, which can efficiently optimize machine learning models even on heterogeneous distributed systems (see Section 6.4). The mentioned techniques in this chapter tend to reveal different perspectives to achieve efficient learning on various hardware platforms. Although it is not possible to cover all relevant techniques, the introduced insights should clearly reveal that a proper usage of hardware can be very effective, especially for the efficiency of learning process.

6.1 FPGA-Based Backpropagation Engine for Feed-Forward Neural Networks

Wayne Luk
Ce Guo

Abstract: Feed-Forward Networks (FFNs), or multilayer perceptrons, are fundamental network structures for deep learning. Although feed-forward networks are structurally uncomplicated, their training procedure is computationally expensive. It is challenging to design customized hardware for training due to the diversity of operations in forward- and backward-propagation processes. In this contribution, we present an approach to train such networks using Field-Programmable Gate Arrays (FPGAs). This approach facilitates the design of reconfigurable architectures by reusing the same set of hardware resources for different operations in backpropagation. In our empirical study, a prototype implementation of the architecture on a Xilinx UltraScale+ VU9P FPGA achieves up to a 5.2 times speedup over the PyTorch platform running on 8 threads on a workstation with two Intel Xeon E5-2643 v4 CPUs.

6.1.1 Introduction

The majority of FPGA-based deep learning architectures are for inference procedures, which makes predictions using pre-trained networks. However, training is a computationally demanding procedure that limits the application of deep neural networks. Backpropagation is the core process in the training procedure of neural networks. This contribution discusses an FPGA-based architecture for backpropagation for Feed-Forward Networks (FFNs). An FFN consists of multiple layers of connected nodes. Each node in a layer takes the weighted sum of the activation signals from the previous layer and generates an activation signal by evaluating a nonlinear activation function.

We study FFNs for two reasons. First, as stand-alone models, they are useful in learning problems with unstructured information such as non-image and non-sequential data. For instance, in the event classification problem for the Higgs boson [3], the correlation between attributes are difficult to capture with other neural networks such as Convolutional Neural Networks (CNNs), while FFNs can provide decent accuracy. Second, as deep network components, FFNs usually appear as the decision-maker in convolutional neural networks; they also serve as generators and discriminators in Generative Adversarial Networks (GANs) [266].

Although FFNs appear less complicated than other neural types of neural networks, training FFNs efficiently on FPGAs is challenging. For instance, in convolutional neural networks, a layer of nodes may share a small set of parameters. This parameter-sharing

property naturally reduces on-chip memory usage. However, FFNs do not have similar properties as each connection between a pair of nodes carries a unique scalar parameter, resulting in a high memory bandwidth usage.

Unlike research in general hardware design for neural network training like [431] and [374], we pay attention to the features and limitations of reconfigurable hardware. The following summarizes the challenges that we face.

1. **Challenge of diverse arithmetic operations.** The hardware resources on an FPGA chip stay unchanged while the arithmetic operations frequently change during backpropagation. To reuse the hardware resources and minimize the overhead for reconfiguration, it is desirable to design multifunctional hardware modules that support multiple operations without runtime reconfiguration.
2. **Challenge of hardware adaptability.** The size of the network and hardware resources vary greatly in different applications. A proper hardware design should be adaptive to all reasonable network sizes and hardware platforms.
3. **Challenge of complex control logic.** When a multifunctional hardware module supports more operations, the control logic becomes more complicated. In particular, it is challenging to define a set of commands to control the hardware modules to collaborate at different stages in backpropagation.

It is difficult to find off-the-shelf solutions because most existing systems for gradient computation and training run on CPUs and GPUs. The novel aspects covered in this contribution include the following:

1. **Reuse of hardware resources for different operations.** We extend the multifunctional multiplication block in [278] to allow different operations to share the same set of hardware resources without runtime reconfiguration, which addresses the challenge of diverse arithmetic operations. In addition, the resource usage of the architecture is independent of the network layout, which addresses the challenge of hardware adaptability.
2. **Commands to control the hardware.** To use the same set of hardware resources for different operations without significant loss of efficiency, it is necessary to find a proper set of commands to control the hardware. We define a small set of commands in this section, addressing the challenge of complex control logic.
3. **Empirical evaluation.** We conduct experiments to compare an experimental implementation of the hardware on an FPGA platform with the PyTorch deep learning framework running on CPUs and GPUs.

6.1.2 Background and Related Work

This section provides a short introduction to Feed-forward networks and their hardware-based training approaches.

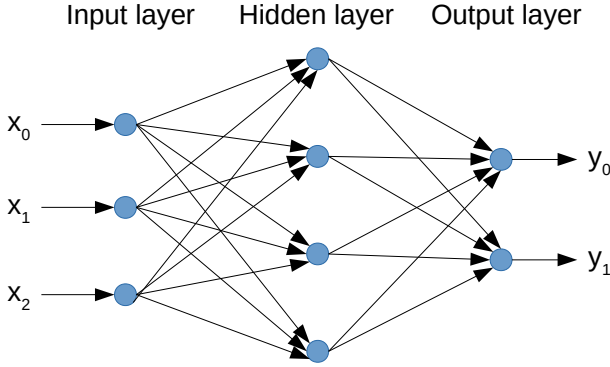


Fig. 6.1: An example feed-forward network.

6.1.2.1 Feed-Forward Networks and Backpropagation

A feed-forward network contains nodes arranged in layers. For instance, Figure 6.1 shows a layout of an FFN with three layers of nodes and two layers of connections. Each node in the input layer feeds a feature of the data point to the network. All nodes in layer l receive signals from all nodes in the previous layer $(l - 1)$ and produce an output signal in the form of a vector. The generation of output signal involves two steps. The first step is to calculate the weighted-sum vector:

$$\mathbf{t}_l = W_{l-1} \mathbf{x}_{l-1} \quad (6.1)$$

where \mathbf{x}_l is a vector containing the output of all nodes in layer l ; and W_{l-1} is a matrix of real numbers specifying N_l weight vectors. The second step is to produce an output signal x_l with

$$\mathbf{x}_l = f_{\text{act}}(\mathbf{t}_l) \quad (6.2)$$

where $f_{\text{act}}(\cdot)$ is a non-linear function defined on real vectors.

It is necessary to specify the weights $W = \{W_0 \dots W_{L-1}\}$ before using the network to make predictions. One may find out the weights using data via training. A training algorithm searches for a set of weights that fits a dataset D with respect to an error measure $E(W, D)$. An efficient training algorithm typically updates the parameter set using the gradient

$$\nabla W = \frac{\partial E(W, D)}{\partial W} \quad (6.3)$$

to minimize the error on the training data. The de facto method to compute the gradient ∇W is backpropagation [422].

An episode of backpropagation includes a forward pass and a backward pass of signals. In the forward pass, the network takes a data point and computes a prediction following the direction of the network. In the backward pass, the network propagates

an error signal in the opposite direction to compute the gradient. Note that the backpropagation process does not include the optimization algorithm, which uses gradients to update the weights [421].

The backpropagation process is computationally demanding. The sheer amount of network parameters results in problems in the algorithms and hardware. With regard to algorithms, the high dimensionality of the parameter space slows down the convergence of the optimization procedure. As a result, the optimization algorithm needs to invoke a large number of backpropagation episodes before obtaining an accurate network model. With regard to hardware, parameters consume considerable memory space and IO bandwidth during computation because the nodes do not share parameters.

The Graphics Processing Unit (GPU) is arguably the most widely-used hardware platform to implement training algorithms for neural networks. The GPU platform provides high performance with relatively low hardware costs and a short design cycle. However, trends in the development of machine learning suggest that FPGAs may become more promising than GPUs for two reasons. First, more neural networks will be based on customized data types, especially quantized numbers [326]. GPUs can natively support only a limited number of data types. By contrast, FPGAs can support customized data types efficiently. Second, the performance gap between GPUs and FPGAs is narrowing fast [541]. In particular, the size of on-chip memory, clock speed, number of hardware DSP units, memory bandwidth, and the process technology of FPGAs have significantly advanced.

6.1.2.2 Reconfigurable Computing for Neural Network Training

Among the statistical models for classification and regression, neural networks are some of the most popular candidates for reconfigurable acceleration [488]. Because we focus on the training process, we do not cover the hardware engines that only perform inference. Reviews that cover inference engines include [408, 497, 500, 546]. The first type of solutions is the acceleration of the training process of general-purpose neural network architectures. Eldredge and Hutchings [198] divide the backpropagation algorithm into three stages and design hardware for each stage separately. During the training process, the hardware performs a runtime reconfiguration at the beginning of each stage. Paul and Rajopadhye [558] propose a systolic backpropagation engine that avoids the runtime reconfiguration. In their design, all calculations in a complete background procedure are mapped to hardware. Murugan et al. [522] designs a training architecture for a network with five nodes. An implementation on a Xilinx Virtex-E FPGA runs at 5.332 MHz. Li and Pedram [437] propose a coarse-grain architecture mainly to implement the matrix multiplication operations in training. Langhammer and Pasca [417] discuss architectures that evaluate common activation functions with different approximation methods. Kim et al. [374] present the DeepTrain platform to perform energy-efficient training for various types of deep networks. The DeepTrain platform offers tools to generate sequences of operations for the hardware architecture using

network descriptions extracted from the TensorFlow deep learning framework. Maeda and Tada [458] propose a training engine for neural networks using the simultaneous perturbation rule [457] to avoid the gradient computation in the training process.

The second type of solutions is the design and optimization of hardware-oriented neural network structures. A popular network structure in this category is the Block-based Neural Network (BbNN). Moon and Kong [502] first propose the BbNN and implement the prediction facility on the FPGA platform. A BbNN connects a collection of neutron blocks. A neutron block carries four numeric I/O ports. Each I/O port may serve as either an input port or an output port. An output port provides an activation signal computed from the input signals on the same neutron block. Jiang et al. [344, 345, 346] study the training process of the BbNN using evolutionary algorithms on the CPU platform. The idea behind their training approach is to encode the topology of the network and the configuration of the neural blocks into a vector so that an evolutionary algorithm may improve the network by manipulating the vector. Merchant and Peterson [488] make it possible to train the block-based neural networks on the FPGA platform. The third type of solutions is the customization of domain-specific or problem-specific neural networks. A representative neural network structure in this category is the convolutional neural network. The convolutional neural network [403] is a feed-forward network structure inspired by the visual cortex of animals. The major application of CNNs is image recognition [403]. The reconfigurable acceleration solutions for CNNs usually take advantages of the unique properties of structured data. Farabet et al. [212] propose an FPGA-based RISC processor that matches basic operations in the CNN. The processor uses a description of a pre-trained CNN in the form of a sequence of instructions to make predictions. A useful observation in this section is that a proper reduction of the precision of image operations results only in a little negative impact on the predictive accuracy. However, the precision reduction may save a considerable amount of hardware resources. Further optimizations [428, 684, 734] have made FPGA devices faster and more energy efficient than CPUs and GPUs. Zhao et al. [744] propose the first stand-alone training engine for CNNs using a streaming data path. The data path contains a collection of parameterized modules. The organization of these modules changes over time with the runtime reconfiguration, which enables the data path to train different layers of a network. In addition to CNNs, FPGA-based reinforcement learning methods have emerged in recent years. Shao and Luk [625] present an architecture trust region policy optimization, which allow robots or agents to efficiently learn policies by interacting with an environment. An implementation on an Intel Stratix-V FPGA achieves up to a 20 times speedup against a 6-thread software reference on an Intel Core i7-5930K CPU running at 3.5 GHz. Gankidi et al. [240] design a Q-learning architecture for a planetary robot. An implementation of the architecture on an Xilinx Virtex-7 FPGA achieves 43 times speedup compared to an Intel 6-gen i5 CPU running at 2.3 GHz.

6.1.3 Architecture for Backpropagation

This section presents the hardware architecture of the backpropagation engine and automated generation of control sequences. During a backpropagation process, two major operations consume the majority of execution time.

1. **Linear combination.** A node takes the linear combination of the outputs from all nodes in the previous layer. In the forward pass, the operation combines activation signals. In the backward pass, the operation combines error signals and computes gradients.
2. **Non-linear function evaluation.** A node evaluates a real-valued non-linear function. In the forward pass, the operation evaluates an activation function. In the backward pass, the operation evaluates the derivative of the activation function.

The two major operations are computationally demanding because their time complexity depends on the network layout [421]. Specifically, the time spent on linear combination grows linearly with the number of network parameters. By contrast, the time spent on function evaluation grows linearly with the number of nodes in the network. Other operations in backpropagation are less computationally expensive than the two major operations. For instance, it is necessary to compare the actual label of the training data and the prediction to calculate the initial error signal for the backward pass. However, the comparison runs only once in a backpropagation episode, and the calculation typically has linear complexity regarding the data dimension.

The two major operations are fundamentally different regarding arithmetic operations. A straightforward way to customize hardware architecture is to create separate arithmetic modules for both operations [437]. However, the sequential dependencies between the calculations allow only the execution of one type of operation at the same time. Therefore, when the arithmetic module for one operation is active, the corresponding arithmetic modules work with a full load, but the modules for all other operations are idle. In other words, only a small fraction of logic units work, resulting in wastage of hardware resources. Admittedly, it is possible to prepare separate bitstreams such that each operation takes all hardware resources [198, 744]. However, it is necessary to reconfigure the hardware to switch between operations. Frequent runtime reconfiguration may take a considerable amount of execution time.

We design a hardware block that works for all backpropagation operations. Our objective is to allow different operations to share as many arithmetic facilities as possible. We use a command sequence to dynamically switch between operations by adjusting the behavior of a small subset of arithmetic units without incurring runtime reconfiguration. Figure 6.2 shows the top-level diagram of the architecture, which supports both linear combination and function evaluation. Major components include the buffer crossbar and the arithmetic block.

- The buffer crossbar communicates with two buffers. At any time, the buffer control signal from the command specifies a source buffer and a target buffer. The crossbar

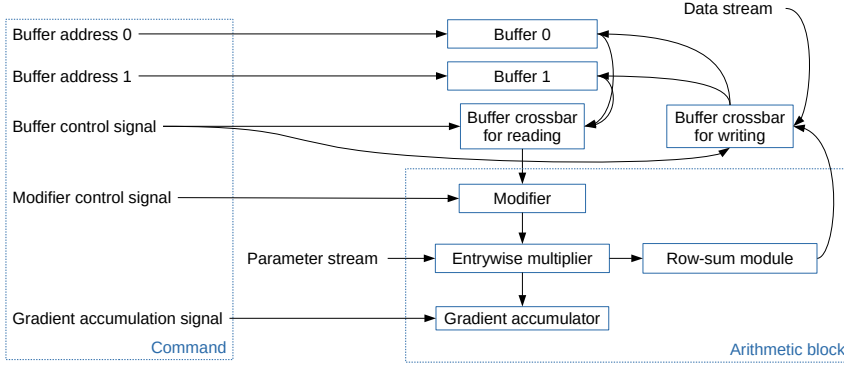


Fig. 6.2: Top-level diagram of the backpropagation engine.

reads a vector from the source vector and passes the vector to the arithmetic block. The crossbar also accumulates the output vector from the arithmetic block to the target buffer.

- The arithmetic block extends the multifunctional multiplication block proposed in [278]. The modifier in the arithmetic block corresponds to the overrider in [278]. This block has two execution modes: the linear mode, which multiplies a matrix by a vector, and the function evaluation mode, which evaluates a non-linear function for all elements in the vector. Our extension includes the gradient accumulator and the row-sum module. A binary signal from the command controls whether the multiplier accumulates the entry-wise products to the gradient. The entrywise multiplier feeds its results to a row-sum module which calculates the sum of each row in the linear mode.

The arithmetic module switches to the linear mode for linear combination. The core calculation for a linear combination operation is to evaluate a vector of weighted sums using a b -dimensional input vector \mathbf{x} and a $b \times b$ weight matrix W . We evaluate an approximate version in the form of a piecewise linear function.

The architecture addresses two challenges in Section 1. First, the components and their connections are independent of the operation. As a result, it is unnecessary to perform a runtime reconfiguration when the operation changes, which addresses the challenge of diverse arithmetic operations. Second, the resource usage is independent of the layout of the network. Therefore, it is possible to scale the architecture for different hardware platforms to control cost or power consumption, which addresses the challenge of hardware adaptability.

One may follow the design flow illustrated in Figure 6.3 to apply the architecture to a backpropagation task. The design flow includes hardware customization and command sequence generation. Hardware customization is the process to set two design parameters. One design parameter is the batch size b , which determines the size of

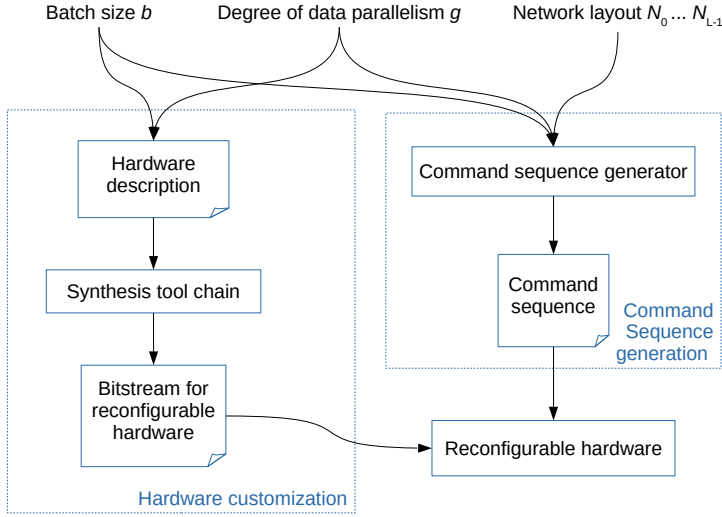


Fig. 6.3: Design flow.

Tab. 6.1: Memory traffic (number of data entries per command).

Operation	ME	Inward	Outward
Data load	0	bg	0
Linear combination for forward pass	0	b^2	0
Function evaluation for forward pass: batch $0..(U-2)$	1	b^2	0
Function evaluation for forward pass: batch $(U-1)$	1	b^2	b
Linear combination for backward pass: node error	0	b^2	0
Linear combination for backward pass: gradient output	0	b	b^2
Function evaluation for backward pass	0	0	0

the entrywise multiplier. A larger b allows the entrywise multiplier to process more multiplications in parallel for a single data point. The other parameter is the degree of data parallelism g , which determines the number of data points processed in parallel. After customization, the architecture has g arithmetic blocks. Each arithmetic block contains a modifier, an entrywise multiplier, and a row-sum module. Each entrywise multiplier includes $b \times b$ scalar multipliers. After filling the design parameters to the hardware description, it is possible to generate a bitstream to program the reconfigurable hardware using the synthesis toolchain. Command sequences generation is the process that produces a sequence of commands from the layout of the network.

The memory bandwidth usage of the hardware depends on the operation and the hardware parameters. For ease of discussion, we assume that all data entries in the feature matrix, network parameters, and gradients have the same width. We may

measure the memory traffic by the number of data entries transmitted per command. Table 6.1 summarizes the memory traffic for different operations.

6.1.4 Collaboration of Components

In this section, we explain how the components collaborate to execute different operations in backpropagation. The modifier in the arithmetic block determines whether the system performs linear combination or non-linear function evaluation.

- The modifier switches to the linear mode for linear combination. The core calculation for a linear combination operation is straightforward. In particular, the arithmetic block evaluates a vector of weighted sums using a b -dimensional input vector \mathbf{x} and a $b \times b$ weight.
- The modifier switches to the function evaluation mode for non-linear function evaluation. Rather than evaluating an activation function following its mathematical definition, the arithmetic block evaluates an approximate version in the form of a piecewise linear function.

Before running the hardware, it is necessary to define a list of commands to control the hardware architecture. We briefly discuss a set of commands that can be used to perform backpropagation in a straightforward manner. This command set addresses the challenge of complex control logic discussed in Section 1. We first describe two commands that read and write the same buffer including data load and memory reset. We then present the commands for linear combination and function evaluation where the arithmetic block read and write different buffers.

The architecture supports two commands that operate on a single buffer. The first command sets the addressed location in the target buffer to zero. As the arithmetic block always accumulates to the target buffer, it is necessary to initialize N_l entries in the target buffer to zero to ensure correct calculation. A command to reset a memory location needs to set the source buffer to be the same as the target buffer and point both addresses to the location to reset. The parameter memory provides a $b \times b$ negative identity matrix. With these settings, the output of the row-sum module is the opposite of the original value $-\mathbf{x}_l$. The accumulation of the value back to the target memory location resets the content to zero. The second command loads d dimensions from g data points to the target location. The memory crossbar directly reads a data point from the data stream, ignoring the output of the arithmetic block.

The other two commands operate on two buffers. The first command is for the linear combination operation. In each batch, the arithmetic block takes b signals as the input and begins to propagate the signals to b nodes in the adjacent layer in parallel. The second command is for the function evaluation operation. Each modifier takes a copy of the variable and evaluates the piecewise linear function that approximates the activation function.

Tab. 6.2: Resource usage.

Resource	Total	Used	Percentage
Logic utilization	1 182 240	382 256	32.33 %
DSP blocks	6840	4105	60.01 %
Block memory (BRAM18)	4320	1292	29.91 %
Block memory (URAM)	960	150	15.63 %

6.1.5 Evaluation

We empirically evaluate the architecture in this section by comparing an FPGA implementation of the architecture and the PyTorch machine learning platform on a dual-CPU workstation.

6.1.5.1 Experiment Settings

We compare our architecture running on an FPGA-based acceleration card with a CPU implementation running on a multicore CPU. The architecture runs on a Xilinx UltraScale+ VU9P FPGA with 16 nm technology. We run the FPGA chip at 120 MHz. The architecture executes command sequence to $g = 16$ data instances in parallel with dimensional batch size $b = 8$. Table 6.2 shows the resource usage of the implementation. The software implementation runs on the PyTorch 1.0 machine learning platform running on a workstation with two Intel Xeon E5-2643 v4 CPUs and 128 GB DDR4 memory. The process technology of the CPUs is 14 nm, which is slightly more advanced than the FPGA. The workstation has 12 physical cores supporting 24 threads in total. The base frequency of the CPU cores is 3.4 GHz, and the maximum turbo frequency is 3.8 GHz.

We consider two representative types of network layouts in the experiments. We call them *bucket-shaped* networks and *cone-shaped* networks respectively for ease of discussion. In a “bucket-shaped” network, all layers contain an identical number of nodes. These networks usually appear in stand-alone classifiers, generative models, and reinforcement learning. In a “cone-shaped” network, a hidden layer has no more nodes than its previous layer. These networks learn compressed features and representation as each layer introduces information loss in a controlled manner.

Table 6.3 shows the test cases we designed using network layouts similar to those in real-world applications. We test two activation functions—rectifier linear function (relu) and the hyperbolic tangent function (tanh)—for each layout. Due to the alignment requirement of our hardware platform, we round the size of each layer to the next multiple of 32. We also produce challenging test cases for each application by linearly scaling the size of all layers. Specifically, the design of test cases is as follows.

- The experiments with “bucket-shaped” networks include 12 test cases. Test cases B0 and B1 correspond to the network structure for reinforcement learning in [276]. The network has 2 hidden layers with 200 nodes in each layer. Test cases B2 and B3

correspond to a study of traffic-flow prediction [454]. The network with the largest layer size contains 3 layers of hidden nodes with 400 nodes in each layer. Test cases B4 and B5 correspond to the network in the generative adversarial networks in [23]. The network contains 4 hidden layers with 512 nodes in each layer. Test cases B6–B11 are challenging versions for B0–B5, where the layer size of each case is 8 times that of the original version.

- The experiments with “cone-shaped” networks include 8 test cases. Test cases C0 and C1 correspond to the stacked autoencoder in [713]. The network has two hidden layers containing 400 and 225 hidden nodes, respectively. Test cases C2 and C3 correspond to the denoising autoencoder for speech data recognition in [221]. The network contains two hidden layers, one with 1000 nodes and another with 500. Test cases C4–C7 are challenging versions for C0–C3, where the layer size of each case is 4 times that of the original version.

We use randomly generated data and network parameters to test the efficiency of the system. Assuming that the function evaluation procedure takes the same time for different inputs, the total execution time for each backpropagation process is independent of the data distribution and the network parameters. In other words, given the same data size, the total execution time should stay unchanged regardless of the data source. As a result, using randomly generated data and network parameters facilitates experiments with various data sizes without affecting observations. The number of data instances for each test case is 2^{20} . In each test case, we calculate the gradient with respect to 100 sets of weights.

6.1.5.2 Results and Discussion

Table 6.3 records the experimental results. In this table, the benchmark column records the numbers of data instances; the ‘CPU-1T’, ‘CPU-4T’, and ‘CPU-8T’ columns contain the execution times in seconds for the corresponding implementation. The ‘SU’ columns give the speedup of the FPGA implementation over the CPU with 1 thread, 4 threads, and 8 threads, respectively.

The architecture discussed in this contribution is faster than the CPU system in all but one test case. In the tests with bucket-shaped networks, the architecture achieves up to a 9.4, 5.4, and 5.2 times speedup compared with the software reference on 1, 4, and 8 threads. In the tests with cone-shaped networks, the architecture achieves up to a 7.8, 4.6, and 4.7 times speedup compared with the software reference on 1, 4, and 8 threads. In addition to the overall speed advantage of the architecture, we have the following additional observations:

1. The software implementation scales poorly with the number of threads. The software running 4 threads achieves only around 2 times speedup against a single thread. The speed advantage on 8 threads over 4 threads is insignificant. In test

Tab. 6.3: Execution time (seconds) and speedup.

ID	Layers	Activation	CPU-1T	CPU-4T	CPU-8T	FPGA	SU-1T	SU-4T	SU-8T
B0	224x2	tanh	0.285	0.219	0.155	0.071	4.0	3.1	2.2
B1	224x2	relu	0.181	0.104	0.090	0.071	2.5	1.5	1.3
B2	416x3	tanh	0.731	0.356	0.274	0.105	7.0	3.4	2.6
B3	416x3	relu	0.521	0.202	0.162	0.104	5.0	1.9	1.6
B4	512x4	tanh	1.020	0.504	0.430	0.138	7.4	3.7	3.1
B5	512x4	relu	0.786	0.320	0.235	0.137	5.7	2.3	1.7
B6	1792x2	tanh	4.102	2.493	2.109	0.615	6.7	4.1	3.4
B7	1792x2	relu	3.715	2.163	1.820	0.615	6.0	3.5	3.0
B8	3328x3	tanh	21.758	13.120	13.176	2.576	8.4	5.1	5.1
B9	3328x3	relu	20.827	13.783	12.400	2.574	8.1	5.4	4.8
B10	4096x4	tanh	45.501	24.847	25.286	4.822	9.4	5.2	5.2
B11	4096x4	relu	44.755	24.036	24.320	4.817	9.3	5.0	5.0
C0	416,256	tanh	0.179	0.125	0.167	0.089	2.0	1.4	1.9
C1	416,256	relu	0.137	0.086	0.102	0.087	1.6	1.0	1.2
C2	1024,512	tanh	0.577	0.339	0.421	0.169	3.4	2.0	2.5
C3	1024,512	relu	0.494	0.266	0.299	0.169	2.9	1.6	1.8
C4	1664,1024	tanh	2.092	1.189	1.259	0.375	5.6	3.2	3.4
C5	1664,1024	relu	2.121	1.104	1.021	0.375	5.7	2.9	2.7
C6	4096,2048	tanh	13.467	7.997	8.147	1.732	7.8	4.6	4.7
C7	4096,2048	relu	13.207	7.643	7.636	1.725	7.7	4.4	4.4

cases B10 and B11, the software running on 8 threads is even slower than when running on 4 threads.

2. The software is more sensitive to the selection of the activation function than the architecture. With the same network layout, the software tends to be faster with the rectifier linear function than with the hyperbolic tangent function. The speed advantage is especially significant when the software runs on 4 and 8 threads. This observation confirms the speed advantage of the rectifier linear function on CPUs [255]. By contrast, the architecture on FPGA evaluates any activation using the same number of commands. Therefore, the execution time of the architecture for a given layout is independent of the activation function.
3. The experimental implementation achieves slightly higher speedup with larger networks that carry more network parameters. For instance, in the experiments with bucket-shaped networks with the ‘tanh’ function, the speedup over 8 threads rises from 2.2 to 5.2 while the layer size expands from 224 to 4096. An exception of the observation is that architecture achieves high speedup against a single CPU thread in test case B0. A possible reason for the exception is that artifacts such as memory initialization for both software and hardware take significant time when the network is small. In this case, the comparison is less reliable than it is in other tests.

Besides the observations above, we have two conjectures based on the hardware design and the experimental results. First, the speed of the architecture will grow if more DSP blocks are available. The arithmetic block contains b^2g scalar multipliers in parallel. Our synthesis tool implements these multipliers mainly with DSP blocks. Therefore, DSP blocks become the critical resource for the design, as shown in Table 6.2. The number of data points processed in parallel grows linearly with the number of multipliers. As a result, when more DSP blocks are available, we may set a larger g to deploy more multipliers to improve the speed. Second, given the same set of hardware resources, our architecture can process networks with more nodes and parameters than some existing solutions such as [558] and [522] for two reasons. One reason is that the number of multipliers is independent of the network layout. The other reason is that the on-chip memory only needs to keep the activation and error signals for two adjacent layers.

6.1.6 Conclusions

We presented a hardware architecture to perform backpropagation for training multi-layer perceptrons. The key to acceleration is to reuse the same set of hardware resources to process different operations involved in backpropagation. Our architecture does not incur runtime reconfiguration when switching between operations. The hardware resource usage is independent of the network layout. A prototype implementation of the architecture on a Xilinx UltraScale+ VU9P FPGA achieves up to 5.2 times speedup over PyTorch running on 8 threads on a workstation with two Intel Xeon E5-2643 v4 CPUs.

6.2 Processor-Specific Code Transformation

Henning Funke
Jens Teubner

Abstract: During the last decade, the compilation of database queries to machine code has emerged as a very efficient alternative to classical, interpretation-based query processing modes [529]. Compiled code can better utilize advanced features of modern CPU instruction sets; avoid interpretation overhead; and—most importantly—minimize data I/O (e.g., to main memory).

This success story raises the hope that compilation strategies can be lifted to non-standard architectures, such as GPUs or other accelerators, as well as to support other data-intensive processing tasks. However, as we shall see in this section, the data-parallel nature of the devices is at odds with established techniques in query compilation, resulting in massive resource under-utilization if compilation strategies are applied too naively.

As a remedy, we propose two novel mechanisms that re-establish compute efficiency of compiled code on data-parallel hardware: *Lane Refill* and *Push-Down Parallelism* are “virtual operators” that participate in optimization and code generation just like true query operators (making our approach seamlessly integrate with existing systems). At runtime, they compensate for lurking resource under-utilization by adapting parallelization strategies on-the-go. The outcome is a resource utilization that is close to the hardware’s maximum, while causing negligible overhead even in unfavorable situations.

Lane Refill and *Push-Down Parallelism* are part of our compiler platform *DogQC*, which leverages modern graphics processors for efficient database query processing.

6.2.1 Data-Parallel Processing Models

Data-parallel processing models are a particularly promising way to max out the achievable compute performance within the constraints of hardware technology (power and heat dissipation). Instead of dedicating chip resources to control flow management, data-parallel architectures target throughput. For instance, executing an instruction for 32 fields at a time can reduce the control flow management work by a factor of 32, when compared with a scalar execution.

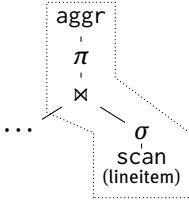



Fig. 6.4: Plan excerpt.

6.2.1.1 Divergence in Data-Parallel Architectures

GPUs are a popular incarnation of this idea, and spectacular performance results have been reported in various application domains. However, actually leveraging the available hardware resources in a beneficial way can be challenging. *Divergence effects*, which may arise whenever data is not perfectly regular, may compromise the benefits.

In this section, we will look at mechanisms to combat performance penalties that may result from divergence effects. To understand the divergence problem, let us consider the execution of a database query, as illustrated here in Figure 6.4 for Query Q10 from the TPC-H benchmark set. A query compiler will attempt to compile the plan region  into a straight-line sequence of code, *i.e.*, a *pipeline*. The motivation to do so is to propagate tuples within registers rather than spilling data to (slow) memory.

During execution, not all lineitem tuples will actually traverse the full pipeline. Some tuples might instead be *eliminated* by operators such as filter σ or join \bowtie . If this happens, a sequential processor will immediately abort the pipeline, continue with the next input item, and hence keep CPU efficiency at peak.

Data-parallel execution back-ends, by contrast, do not have the option of aborting a pipeline early, unless *all* tuples in the same batch of work are eliminated.

Figure 6.5 illustrates this effect for a GPU-based back-end (assuming a batch—or “*warp*”—size of eight for illustration purposes). In some warp iteration, only *warp lanes* 1, 5, and 7 might have passed the filter σ , leaving the five remaining warp lanes *inactive* (indicated as dashed arrows $- \rightarrow$). The following join de-activates another two warp lanes, bringing GPU efficiency down to $1/8$ in this example.

The resulting GPU under-utilization is even worse in real settings. To scan a lineitem table with 150 million rows, actual GPUs will require 5 million *warp iterations*, each consisting of 32 warp lanes. Although σ filters out about $2/3$ of all rows, it is extremely unlikely that all lanes within a warp become inactive. Therefore, (almost) all 5 million warp iterations proceed into the join operator \bowtie . Only 1% of the remaining rows find a match during the join. In an actual dataset, 2.9 million rows remain after the join, but they are spread across 1.1 million warp iterations. Ideally, the projection π and aggregation *aggr* operators could have been processed by only $2.9 \text{ M} / 32 = 90 \text{ K}$ warp iterations. In other words, state-of-the-art query compilation techniques will leave 92% of the GPU’s processing capacity unused.

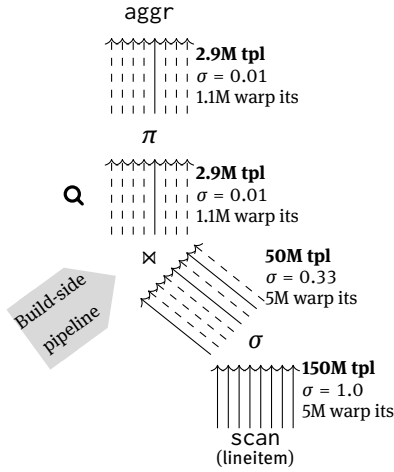


Fig. 6.5: GPU under-utilization due to filter divergence.

6.2.1.2 DogQC: A Database Query Compiler for GPUs

GPU code generated by our query compiler *DogQC*¹ leverages *Lane Refill* and *Push-Down Parallelism* techniques to counter divergence effects like the ones illustrated above. In the rest of this section, we will give a high-level idea of the *Lane Refill* and *Push-Down Parallelism* techniques (Sections 6.2.2 and 6.2.3), then report on experimental results for DogQC (Section 6.2.4), and wrap up in Section 6.2.5. More details on the *Lane Refill* and *Push-Down Parallelism* mechanisms can be found in the respective full paper [237].

6.2.2 Lane Refill Technique

Divergence effects (here: *filter divergence*) are a consequence of the SIMT (“single instruction, multiple threads”) execution paradigm embodied in all modern graphics processors. A number of threads (or *lanes*, typically 32 of them) are grouped into a *warp*. During execution, *all* lanes within a warp execute the *same* GPU instruction.

The SIMT model encounters a problem whenever some lanes or data elements need a different amount or kind of processing than others. In such situations, control flows will *diverge*. Since all lanes within a warp *still* execute the same instruction, lanes will be turned *inactive* and their computation result will be discarded. As illustrated above, this can result in resource under-utilization.

To illustrate the severity of this effect, we instrumented the query plan shown earlier (Figure 6.5) to monitor warp utilization at the plan point marked with a magnifying glass **Q**. Figure 6.6 shows a histogram on the number of warps that have passed this

¹ <https://github.com/Henning1/dogqc>.

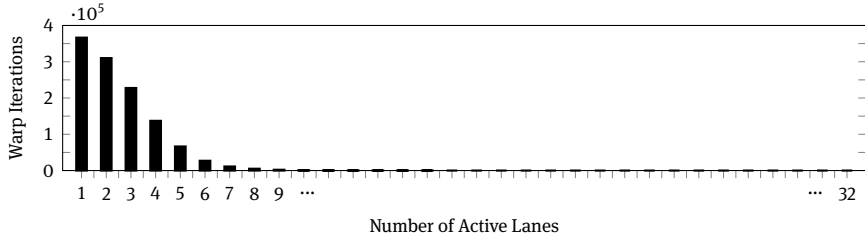


Fig. 6.6: Lane activity profile with filter divergence.

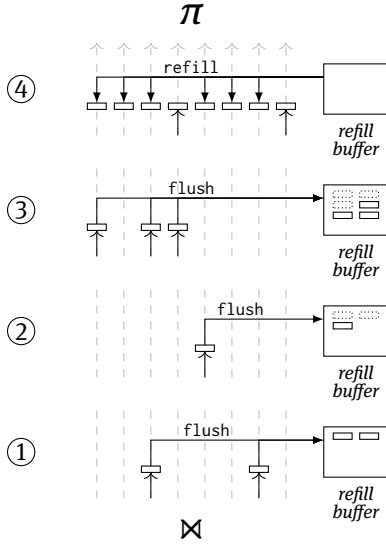


Fig. 6.7: *Lane Refill*: tuples from three low-activity iterations are suspended to the *refill buffer* and resumed for full lane activity in the fourth iteration.

plan stage with a warp utilization of 1, ..., 32 active lanes. It is easy to see that only a fraction of the available compute capacity is used; in most warps, only one or two out of 32 warp lanes performed actual work.

6.2.2.1 Balance Operators and Refill Buffers

To combat the situation, DogQC injects *balance operators* into the relational query plan. Code generated for these operators detects warp under-utilization at runtime. Whenever utilization drops below a configured threshold, the state of all remaining active lanes is suspended to a *refill buffer* and the pipeline starts over with a fresh set of input tuples.

Figure 6.7 illustrates this for three successive warp iterations ① through ③. Since only 2, 1, and 3 lanes remained active in these iterations (respectively), their state is

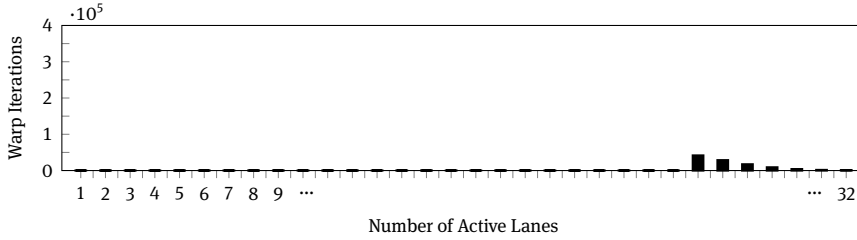


Fig. 6.8: Lane activity profile with lane refill buffer to consolidate filter divergence.

flushed to the refill buffer. After flushing, each of those warp iterations is terminated and processing starts over with the next set of input tuples.

6.2.2.2 Refilling

As soon as a sufficient number of lane states have been stored to the refill buffer, the buffer can be used to *refill* lanes that have become inactive. This time, the under-utilized warp iteration is not terminated but continues processing with full utilization after refilling. This is visualized in Step ④ of Figure 6.7. Here, only two out of eight warp lanes remained active after the downstream join operator. Using the refill buffer, the remaining six warp lanes can be filled with useful work, resulting in full warp utilization upstream.

Implementationwise, flushing and refilling are backed up in DogQC by CUDA’s `__ballot_sync`, `__popc` (“population count”), and shuffling primitives. These primitives are highly efficient; balance operators will cause little overhead even when only a few warps go below the utilization threshold.

6.2.2.3 Effect of *Lane Refill*

Lane Refill brings warp utilization back to a high compute efficiency. Following the balancing operator, all executed warps (except for the last warp in each grid block) are *guaranteed* to have a warp utilization above the configured threshold.

In Figure 6.8, this is illustrated with a histogram for the same plan point that we profiled earlier (Figure 6.6), but this time with a balance operator applied. The histogram confirms that (a) (almost) no warps exist with a utilization below 26 lanes (the threshold we configured); and (b) the total number of executed warps has dropped by a factor of about 10. In terms of overall execution performance, *lane refill* will improve execution times by about $2 - 3\times$ for the example plan shown in Figure 6.5.

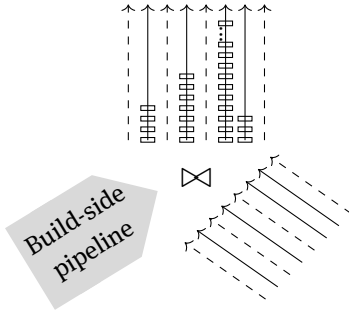


Fig. 6.9: Expansion divergence. Here, some rows in the probe-side relation will find more join partners on the probe side than on the other side.

6.2.3 Push-Down Parallelism Technique

DogQC’s *Push-Down Parallelism* technique addresses another flavor of divergence that may arise orthogonally to the aforementioned filter divergence. *Expansion divergence* is the effect when a different amount of work is needed to process each of the items within a warp. Database *join operations* are a common situation where this effect arises. Figure 6.9 on the right illustrates the effect. Probe side tuples coming from the right may find a different number of join partners each. Specifically, in the example, lane 6 will have significantly more tuples to process than the remaining warp lanes. In such a situation, existing query compilers will process all matches of a single probe-side tuple *within* the same warp lane. In the example, execution times would be dominated by the sequential processing of all matches for lane 6.

Push-Down Parallelism mitigates the situation by parallelizing the processing of the matches of a single probe-side tuple *across* the available warp lanes. To this end, the execution state of probe-side lanes is *broadcast* over lanes, as illustrated in Figure 6.10. Build-side matches are *partitioned* across. Again, we leverage efficient CUDA primitives, such as `__ballot_sync` and `__shfl_sync` (“shuffle sync”). Please refer to [237] for details.

As illustrated in Figures 6.11 and 6.12, *Push-Down Parallelism* improves lane utilization and reduces the overall number of iterations needed to complete the query. *Lane Refill* and *Push-Down Parallelism* complement one another, and Figure 6.9 shows an example where both flavors of divergence co-exist. Another typical occurrence of expansion divergence is the processing of *variable-length data*, strings in particular. If possible, DogQC will parallelize the processing of strings across warp lanes to improve resource utilization.

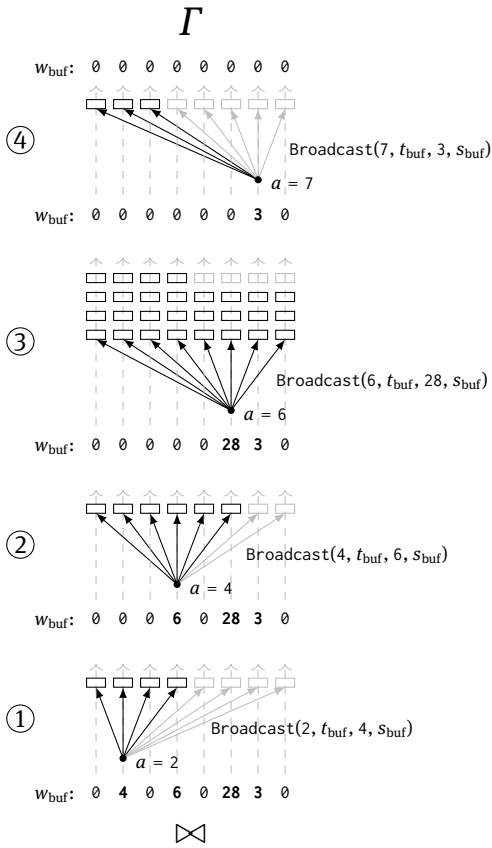


Fig. 6.10: Illustration of push-down parallelism that expands the join matches of four warp lanes.

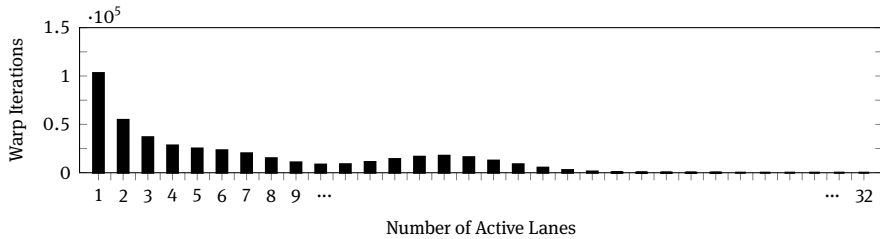


Fig. 6.11: Lane activity with expansion divergence.

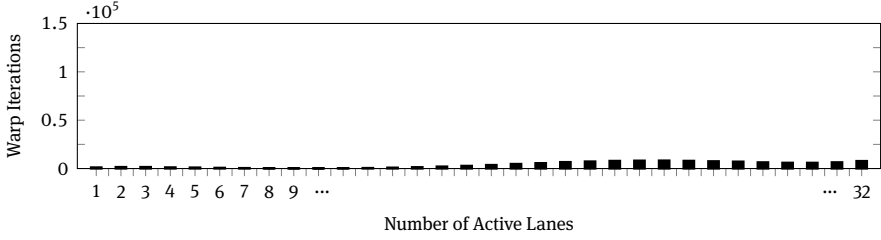


Fig. 6.12: Lane activity profile with push-down parallelism to consolidate expansion divergence.

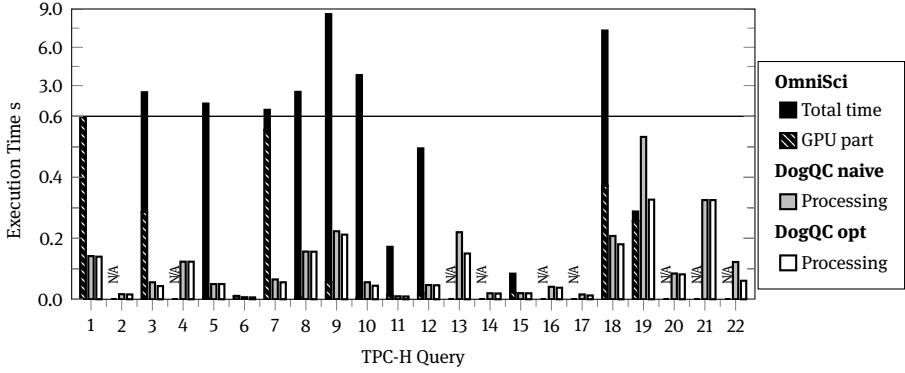


Fig. 6.13: Execution times of DogQC for TPC-H benchmark queries (scale factor 25). The divergence optimizations improve query performance.

6.2.4 Evaluation

With *DogQC*, we provide a query compiler with a wide range of SQL functionality, sufficient to support all queries from the TPC-H benchmark set. Here we use *DogQC* and the database domain to illustrate the aforementioned anti-divergence mechanisms, which could equally be applied to other data-intensive tasks, including those related to machine learning.

6.2.4.1 TPC-H Performance

To assess the benefits of measures to contain divergence, we performed a series of measurements with the TPC-H benchmark set. Our measurements were based on an NVIDIA RTX2080 GPU with 46 Streaming Multiprocessors and 8 GB GPU memory, installed in a host system with an Intel i7-9800X CPU and 32 GB of main memory. As a reference, we compared *DogQC* with the hybrid CPU/GPU system *OmniSci* [545].

Our benchmark results are depicted in Figure 6.13. For each of the 22 TPC-H queries, the bars indicate the query execution time assuming that the dataset is resident in GPU memory.

For OmniSci, we report the total wall clock time needed to execute the query as well as the amount of time spent on GPU processing. OmniSci is a hybrid execution engine in which both CPU and GPU will be used to jointly answer the query. As can be seen in Figure 6.13, several queries can, in fact, not benefit much from GPU in OmniSci. Also note that OmniSci could successfully execute only 13 of the 22 TPC-H benchmark queries. DogQC, by contrast, can run all 22 TPC-H queries entirely on the GPU, with execution times that are up to 86× faster than those of OmniSci.

6.2.5 Summary

In this research, we put the processing capabilities of data-parallel co-processors for non-uniform, data-intensive workloads to the test. DogQC introduces techniques that allow us to gracefully align parallel processing units with work items, even when problems are heavily skewed. We observe that *Lane Refill* and *Push-Down Parallelism* are able to increase processing efficiency for these non-uniform workloads, sometimes with dramatic effects on processing throughput.

Existing query coprocessors typically avoid imbalances by working on a uniform surrogate (such as dictionary keys or materialization barriers). This has led to the perception that GPUs have limited capabilities of processing irregular problems. DogQC avoids the overhead of maintaining such additional data structures and instead restores balance during non-uniform processing.

Here we showcase *Lane Refill* and *Push-Down Parallelism* based on an application to database query processing. Compared with state-of-the-art platforms, our prototype DogQC achieves better resource utilization, a bigger functionality range, and better runtime performance on realistic benchmarks. Looking ahead, our anti-divergence measures could be applicable to many machine learning scenarios, especially when the problems involved are heavily skewed and/or depend on non-linear computations.

6.3 Extreme Multicore Classification

Erik Schultheis

Rohit Babbar

Abstract: There are classification problems, such as assigning categories to a Wikipedia article, where the possible set of labels is very large, numbering in the millions. Somewhat surprisingly, these so-called Extreme-Multilabel Classification (XMC) problems can be solved quite successfully by applying a linear classifier to each label individually. This decomposition into binary problems is called a one-vs-rest reduction. As these problems are completely independent, the reduced task is embarrassingly parallel and can be trivially spread across multiple cores and nodes. After training, the model can be sparsified by culling small weights to only require a fraction of the memory and computational power for prediction on new samples.

6.3.1 Introduction to Extreme Multilabel Classification

Extreme Multi-label Classification (XMC) refers to supervised learning with a large target label set where each training/test instance is labeled with a small subset of relevant labels. Machine learning problems consisting of hundreds of thousands of labels are common in various domains such as annotating web-scale encyclopedias [585], hash-tag suggestion in social media [171], and image-classification [168]. For instance, all Wikipedia pages are tagged with a small set of relevant labels that are chosen from more than a million possible tags in the collection. It has been demonstrated that, in addition to automatic labelling, the framework of XMC can be leveraged to effectively address learning problems arising in recommendation systems, ranking, and web-advertising [9, 585].

Notation and Setup Let the training data $D := \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$ consist of input feature vectors $\mathbf{x}^{(i)} \in \mathcal{X} \subseteq \mathbb{R}^d$ and respective output vectors $\mathbf{y}^{(i)} \in \mathcal{Y} := \{0, 1\}^m$ such that $y_l^{(i)} = 1$ iff the l -th label belongs to the training instance $\mathbf{x}^{(i)}$. The feature vectors form the rows of the feature matrix \mathbf{X} . In XMC settings, the cardinality m of the set of target labels, the dimension of the input d , and the size of the dataset N can all be of the order of hundreds of thousands or even millions.

For text data, the input can be represented by term-frequency inverse-document-frequency (tf-idf) features. In that case, the dimensionality of the feature space is determined by the size of the vocabulary, and for each text $\mathbf{x} \in \mathcal{X}$ the feature x_j is nonzero only if the corresponding word appears in the text. As a result, the input features are highly sparse. The magnitude of the feature is determined by how often

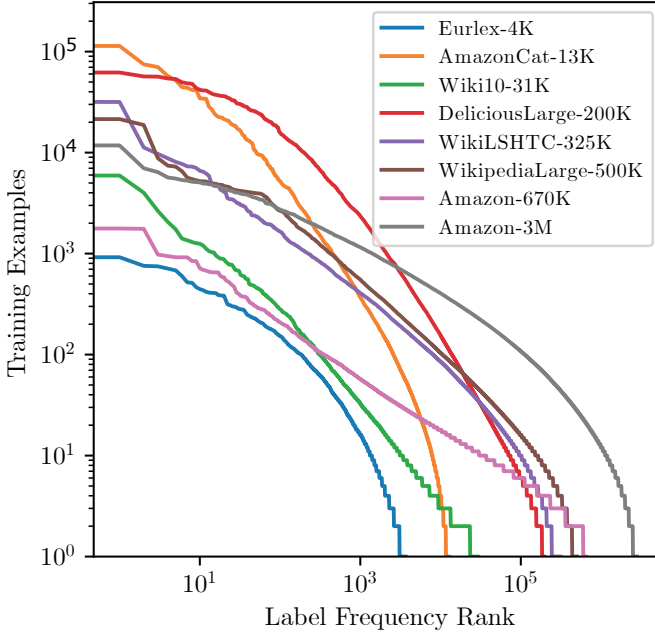


Fig. 6.14: Label frequency in XMLC datasets. X-axis shows the label rank when sorted by the frequency of positive instances and Y-axis gives the number.

the word appears in the document and in the entire corpus. For details on tf-idf, see, e.g., Manning, Raghavan, and Schütze [463].

Similarly, for any given instance $\mathbf{x}^{(i)}$ only a small subset of the labels will be relevant, $\|\mathbf{y}^{(i)}\|_1 \ll m$. Additionally, the number of instances for which a label is relevant is very imbalanced: Few labels will be relevant to many instances, but most labels will apply only to an extremely small fraction. This gives rise to a *long-tailed* label distribution, as shown in Figure 6.14. The labels with very few positives are called *tail labels*. The characteristics of well-known benchmark datasets in XMC are presented in Table 6.4.

In traditional multi-label classification, the goal is to learn a multi-label classifier in the form of a vector-valued output function $h : \mathbb{R}^d \mapsto \{0, 1\}^m$. In XMC, one often wants to restrict the classifier to predict a fixed number of labels because, say, a web interface might have a fixed number of slots in which to suggest related searches. This leads to classification functions $h_k : \mathbb{R}^d \mapsto \mathcal{Y}_k := \{\mathbf{y} \in \mathcal{Y} : \|\mathbf{y}\|_1 = k\}$. Such a function is typically constructed by first learning a score function $r : \mathbb{R}^d \mapsto \mathbb{R}^m$, and then taking the k highest-scoring labels as the prediction.

Evaluation Metrics Due to the extreme sparsity in the label vector, metrics such as accuracy are not informative in the case of XMC. Instead, one typically uses metrics that

Tab. 6.4: Multi-label datasets from XMC repository [50]. APpL and ALpP represent average points per label and average labels per point, respectively.

Dataset	#Training	#Features	#Labels	APpL	ALpP
AmazonCat-13K	1 186 239	203 882	13 330	448.6	5.0
AmazonCat-14K	4 398 050	597 540	14 588	1 330.1	3.5
Amazon-670K	490 449	135 909	670 091	4.0	5.5
Wiki10-31K	14 146	101 938	30 938	8.5	18.6
Delicious-200K	196 606	782 585	205 443	72.3	75.5
WikiLSHTC-325K	1 778 351	1 617 899	325 056	17.5	3.2
Wikipedia-500K	1 813 391	2 381 304	501 070	24.8	4.8

focus on the k predicted labels. Most commonly used are precision at k , denoted $P@k$, and normalized Discounted Cumulative Gain, denoted $nDCG@k$ [50]. Let $\mathbb{R}^m \ni \hat{\mathbf{y}} = r(\mathbf{x})$ be the predicted scores for an instance with corresponding label vector \mathbf{y} . These metrics are defined by

$$P@k(\mathbf{y}, \hat{\mathbf{y}}) := \frac{1}{k} \sum_{l \in \text{rank}_k(\hat{\mathbf{y}})} y_l \quad (6.4)$$

$$nDCG@k(\mathbf{y}, \hat{\mathbf{y}}) := \sum_{l \in \text{rank}_k(\hat{\mathbf{y}})} \frac{y_l}{\log(R_l(\hat{\mathbf{y}}) + 1)} \bigg/ \sum_{l=1}^{\min(k, \|\mathbf{y}\|_1)} \frac{1}{\log(l + 1)}, \quad (6.5)$$

where $\text{rank}_k(\mathbf{y})$ returns the k largest indices of \mathbf{y} ranked in descending order, and R_l gives the ordering of the l 'th index. Note that unlike $P@k$, $nDCG@k$ takes into account the ranking of the correctly predicted labels. For instance, if there is only one of the five labels that is correctly predicted, then $P@5$ gives the same score if the correctly predicted label is at rank 1 or rank 5. By contrast, $nDCG@5$ gives a higher score if it is predicted at rank 1 and the lowest non-zero score at rank 5.

6.3.2 Parallel Training of Linear One-vs-Rest Models

The $P@k$ (Equation 6.4) and $nDCG@k$ (Equation 6.5) metrics introduced above are non-differentiable and thus not directly usable in typical gradient-based empirical-risk-minimization procedures. However, it can be shown that in order to achieve optimal predictions for precision at k , one only needs to train the scoring function r in such a way that the scores are strictly monotone transformations of the labels' marginals [486]. Therefore, one can train the classifier by independently applying a classification-calibrated² loss ℓ_{BC} , such as binary cross entropy or (squared) hinge loss, to each label

² See e.g. Bartlett, Jordan, and McAuliffe [44]. Intuitively, this means that a classifier that minimizes ℓ_{BC} also minimizes the binary 0-1 loss.

individually. Therefore, the training objective is given by

$$\min_r \sum_{i=1}^N \sum_{l=1}^m \ell_{\text{BC}} \left(y_l^{(i)}, r_l(\mathbf{x}^{(i)}) \right). \quad (6.6)$$

Such a decomposition is called the *One-vs-Rest* (or One-vs-All) reduction.

Objective Functions for Linear One-vs-Rest This expression becomes particularly favorable if the scoring function r is linear. In that case, the minimization task decomposes into m completely independent subtasks

$$\forall l \in [m] : \min_{\mathbf{w}^{(l)} \in \mathbb{R}^d} \sum_{i=1}^N \ell_{\text{BC}} \left(y_l^{(i)}, \mathbf{x}^{(i)\top} \mathbf{w}^{(l)} \right). \quad (6.7)$$

Due to the embarrassingly parallel nature of the training tasks, the computation can easily scale to use thousands of compute cores. A scalable implementation of this method yielding state-of-the-art prediction performance was demonstrated via the DiSMEC algorithm [30], which is a multi-label wrapper around the Liblinear solver [210]. In DiSMEC, the underlying binary loss is the squared hinge loss with an additional l_2 regularization term. Its objective is

$$\forall l \in [m] : \min_{\mathbf{w}^{(l)} \in \mathbb{R}^d} \left(\|\mathbf{w}^{(l)}\|_2^2 + c \sum_{i=1}^N \left(\max(0, 1 - s_l^{(i)} \mathbf{x}^{(i)\top} \mathbf{w}^{(l)}) \right)^2 \right), \quad (6.8)$$

where $c \in \mathbb{R}_{>0}$ is the parameter to control the trade-off between empirical error and the model complexity and $s_l^{(i)} := 2y_l^{(i)} - 1$ is the label represented as $\{+1, -1\}$.

A similar method is ProXML [29], which switches the l_2 regularization for l_1 regularization in order to induce robustness to l_∞ perturbations in the input samples. This robustness is particularly helpful for tail labels, which have very few positive training instances. The objective of ProXML thus is

$$\forall l \in [m] : \min_{\mathbf{w}^{(l)} \in \mathbb{R}^d} \left(\|\mathbf{w}^{(l)}\|_1 + c \sum_{i=1}^N \left(\max(0, 1 - s_l^{(i)} \mathbf{x}^{(i)\top} \mathbf{w}^{(l)}) \right)^2 \right). \quad (6.9)$$

Suppose for now that we have a method $\mathcal{A} : (\mathbf{X}, \mathbf{s}) \mapsto \mathbf{w}_*^{(l)}$ available to solve these individual problems efficiently in a single thread. (This will be discussed in Section 6.3.3). Then the following framework can be used to scale the training process to multiple cores and nodes:

Two-Level Parallelization The distributed training for the optimization problems defined by equations (6.8) and (6.9) is implemented using a two-layer parallelization architecture. At the top level, labels are separated into batches of, say, $M = 1000$, which can be processed independently in parallel on available compute nodes, or sequentially

if the number of batches exceeds the number of nodes. On each node, training of a batch of M labels is parallelized using multiple threads, which forms the second layer of parallelization.

After each $\mathbf{w}_*^{(l)}$ is trained, weights of small magnitude are pruned to reduce overall model size drastically, often by more than 99 %. Since this can be performed as soon as $\mathbf{w}_*^{(l)}$ has been computed, there is no need to store the complete dense model, even for a single batch, which reduces the RAM requirements for the algorithm. Unfortunately, in typical sparse matrix formats such as compressed sparse row/column matrices, insertion of new values cannot be done by multiple threads in parallel, because it might require reallocation and the shifting of data in other parts of the matrix. For this reason, we represent the sparse weight matrix as an array of independently allocated sparse vectors that can be written to independently.

The two-layer distributed training framework is summarized in Algorithm 4.

Algorithm 4: Framework for hardware-aware embarrassingly parallel training in DiSMEC and ProXML solvers. The iterations of both loops are independent and can thus be run in parallel.

Input: Training data $D = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}) \dots (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$ in sparse representation,
input dimensionality d , label set $\{1 \dots m\}$, batch size M

Output: Learnt matrix $\mathbf{W} \in \mathbb{R}^{d \times m}$ in sparse format

// 1st parallelization; independent nodes

```

1 for  $\{b = 0; b < \lfloor \frac{m}{M} \rfloor + 1; b++\}$  do
2   Load single copy of feature matrix  $\mathbf{X}$  into main memory
3   Prepare array  $\mathbf{W}_b$  of  $M$  sparse vectors
   // 2nd parallelization; independent threads
4   for  $\{l = b \times M; l \leq (b + 1) \times M; l++\}$  do
5     Generate binary sign vector  $\mathbf{s}^{(l)} = \{+1, -1\}_{i=1}^N$ 
6     train weight vector  $\mathbf{w}_*^{(l)}$  on a single core using  $\mathcal{A}(\mathbf{X}, \mathbf{s}^{(l)})$ ,
7     Prune small weights in  $\mathbf{w}^{(l)}$ 
8   return  $\mathbf{W}_{d,M}$ 
9 return  $\mathbf{W}$ 
```

An advantage of the two-level parallelization over just running m instances of an off-the-shelf solver for binary problems is that the feature matrix \mathbf{X} can be shared for all training jobs running on the same node. This allows us to keep the entire dataset, which may be several gigabytes in size, in main memory. However, on modern CPUs with a large number of cores, or on nodes with a two-socket configuration, this might cause problems due to *Non-Uniform Memory Access* (NUMA). In such a system, the overall RAM is partitioned into regions, called *NUMA domains*. Even though all cores in the

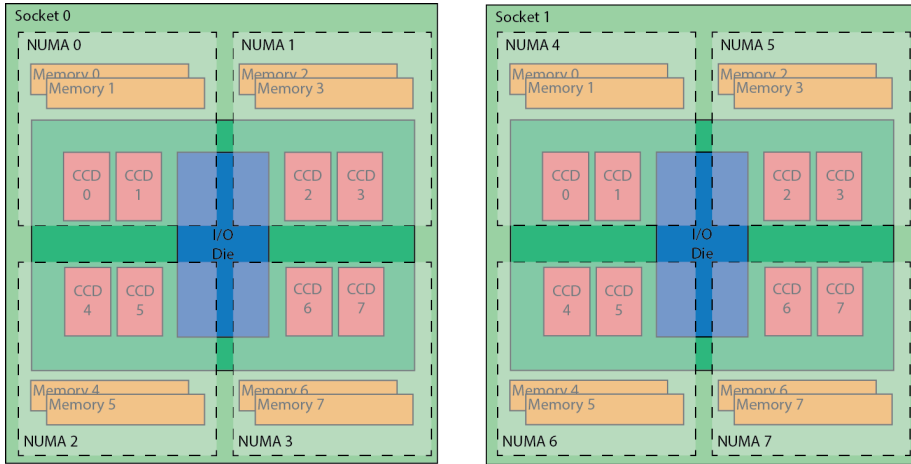


Fig. 6.15: NUMA memory in a dual-socket 64-core AMD Rome 7H12 system. Each CCD contains 8 cores, and each core has fastest access only to the memory within its own NUMA domain, marked with dashed lines. Image by CSC - IT Center for Science under CC-BY-4.0.

system can access the entirety of the memory, access latencies to the different domains vary depending on the distance of the core to the domain. An example of a NUMA setup is given in Figure 6.15.

In such a setup, a single copy of the feature matrix would be accessed by all cores on the system, quickly bottlenecking the memory bus and preventing the program from making efficient use of the available CPU cores. This can be mitigated by pinning threads to their CPU cores and replicating the feature matrix once per NUMA domain. In this way, each thread can read the feature matrix from its local domain, reducing latency, and the memory reads are spread out across different memory modules, improving throughput. In order to achieve this, the outer parallelization layer has to be performed not over physical nodes but over NUMA domains.

6.3.3 Second-Order Optimization Using Conjugate Gradients

The objective Equation 6.8 can be minimized in batch mode using second-order optimization. Compared with the popular (stochastic) gradient descent strategy, second-order optimization can take the curvature of the loss landscape into account and thus converges to the minimum in much fewer iterations. However, the computations for each single iteration are much more involved, as the second-order information is encoded in the potentially very large Hessian matrix. Fortunately, it is possible to implement this procedure without ever actually forming the Hessian, as will be described below.

Dropping the label index, we can write

$$R_D[\mathbf{w}] = \mathbf{w}^T \mathbf{w} + c \sum_{i=1}^N \ell_{\text{SH}} \left(s_i \mathbf{x}^{(i)T} \mathbf{w} \right), \quad (6.10)$$

where ℓ_{SH} is the squared hinge loss

$$\ell_{\text{SH}}(r) = \max(0, 1 - r)^2. \quad (6.11)$$

Note that this objective function is convex. As a consequence, the optimizer will converge to the global optimum regardless of the starting point.

Determining the Descent Direction The main idea of second-order optimization is to approximate the objective locally using its quadratic Taylor approximation

$$R_D[\mathbf{w} + \boldsymbol{\delta}] \approx R_D[\mathbf{w}] + \nabla R_D[\mathbf{w}] \boldsymbol{\delta} + 0.5 \boldsymbol{\delta}^T \nabla^2 R_D[\mathbf{w}] \boldsymbol{\delta}. \quad (6.12)$$

Therefore, the step $\boldsymbol{\delta}_*$, which is ideal, i.e. which leads to the minimum, in this approximation can be calculated by solving the linear system

$$\nabla^2 R_D[\mathbf{w}] \boldsymbol{\delta}_* = -\nabla R_D[\mathbf{w}]. \quad (6.13)$$

For Equation 6.8, the gradient and Hessian have a simple structural form [239, 368]

$$\nabla R_D[\mathbf{w}] = 2\mathbf{w} + c \sum_{i=1}^N \ell'_{\text{SH}} \left(s_i \mathbf{x}^{(i)T} \mathbf{w} \right) s_i \mathbf{x} \quad (6.14)$$

$$\nabla^2 R_D[\mathbf{w}] = 2\mathbf{I} + c \mathbf{X}^T \mathbf{D} \mathbf{X}, \quad (6.15)$$

where \mathbf{I} is the identity matrix, $\mathbf{X} = [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}]^T$ is the data matrix, and \mathbf{D} is diagonal with entries $D_{ii} = \ell''_{\text{SH}}(s_i \mathbf{x}^{(i)T} \mathbf{w})$.

The Hessian matrix has size $N \times N$, and thus would be far too large to be stored in memory. Fortunately, Equation 6.13 can be solved using a conjugate-gradient procedure, which requires only Hessian-vector products. These can be calculated efficiently through

$$\nabla^2 R_D[\mathbf{w}] \boldsymbol{\delta} = 2\boldsymbol{\delta} + c \mathbf{X}^T \mathbf{D} \mathbf{X} \boldsymbol{\delta}, \quad (6.16)$$

because \mathbf{X} is a very sparse matrix. In practice, Equation 6.13 is only solved approximately, drastically reducing the number of conjugate-gradient iterations, and thus Hessian-vector products, that need to be calculated.

Determining the Step-Size The resulting step vector $\boldsymbol{\delta}$ might be outside the region in which the quadratic approximation (see Equation 6.13) accurately models the true risk landscape $R_D[\mathbf{w}]$. Therefore, a step-size mechanism is needed usually either by using a trust region or by doing a line search.

Due to the linear nature of the ranking function $r(\mathbf{x}; \mathbf{w}) = \mathbf{x}^\top \mathbf{w}$, the line search can be implemented efficiently by using

$$r(\mathbf{x}; \mathbf{w} + \lambda \boldsymbol{\delta}) = \mathbf{x}^\top \mathbf{w} + \lambda \mathbf{w}^\top \boldsymbol{\delta} \quad (6.17)$$

$$\|\mathbf{w} + \lambda \boldsymbol{\delta}\|_2^2 = \|\mathbf{w}\|_2^2 + 2\lambda \mathbf{w}^\top \boldsymbol{\delta} + \lambda^2 \boldsymbol{\delta}^\top \boldsymbol{\delta}. \quad (6.18)$$

By caching the values of $\|\mathbf{w}\|_2^2$, $\mathbf{w}^\top \boldsymbol{\delta}$, $\boldsymbol{\delta}^\top \boldsymbol{\delta}$, $\mathbf{x}^\top \mathbf{w}$ and $\mathbf{x}^\top \boldsymbol{\delta}$, the cost of evaluating the loss for any value of λ after the first evaluation drops to $O(N)$ evaluations of ℓ_{SH} and the corresponding additions and multiplications of the cached values in Equations 6.17 and 6.18.

Implicit Hard-Instance Mining in Hinge Losses When using the squared hinge loss (see Equation 6.11) for ℓ_{SH} , the loss and all its derivatives become zero once the sample is classified correctly with a sufficient margin³ $\mathbf{s} \mathbf{x}^\top \mathbf{w} > 1$. Consequently, the corresponding entries in the diagonal matrix \mathbf{D} in Equation 6.16 become zero. This means that in the product $\mathbf{X}^\top \mathbf{D} \mathbf{X}$, the feature matrix \mathbf{X} can be replaced with a much smaller matrix $\tilde{\mathbf{X}}$ that contains only the examples that are not classified correctly with a margin. Denote with $\mathcal{E} := \{i \in [N] : \mathbf{s}_i \mathbf{x}^{(i)\top} \mathbf{w} \leq 1\}$ the set of indices of examples with nonzero loss (the *hard* instances), then $\tilde{\mathbf{X}} = [\mathbf{x}^{(i)} : i \in \mathcal{E}]^\top$.

As a consequence, the full feature matrix \mathbf{X} is only needed once per step to determine the gradient $\nabla R_D[\mathbf{w}]$, \mathbf{D} , and the hard examples \mathcal{E} . Afterwards, each CG iteration only requires the reduced matrix $\tilde{\mathbf{X}}$. This can be interpreted as an implicit hard instance mining step that is performed at the beginning of each step. As the weight vector \mathbf{w} approaches the optimal weights \mathbf{w}_* , most instances will have sufficient margin, and only few hard instances remain, $|\mathcal{E}| \ll N$ (cf. Figure 6.16). Therefore, later iterations require significantly less computation time than earlier ones. In fact, by using an initial vector \mathbf{w}_0 for which the hard-example set \mathcal{A} is already small can speed up the overall computation time tremendously, as discussed below.

6.3.4 Further Performance Improvements

Mean-Separating Initialization A simple attempt to improve the initial weight vector is to choose a hyperplane that separates the means of the positive and negative instances for that label. Denote $\mathcal{P} := \{\mathbf{x}^{(i)} : i \in [N], y^{(i)} = 1\}$ and $\bar{\mathbf{x}} := N^{-1} \sum_{i=1}^N \mathbf{x}^{(i)}$, then the means are

$$\bar{\mathbf{p}} := \frac{1}{|\mathcal{P}|} \sum_{\mathbf{x} \in \mathcal{P}} \mathbf{x}, \quad \bar{\mathbf{n}} := \frac{N\bar{\mathbf{x}} - |\mathcal{P}|\bar{\mathbf{p}}}{N - |\mathcal{P}|}. \quad (6.19)$$

³ The margin of an instance denotes how far its score is from the classification boundary. An instance with a margin of 0 is classified correctly, but the slightest perturbation of its features or the classifier's weights could change the classification.

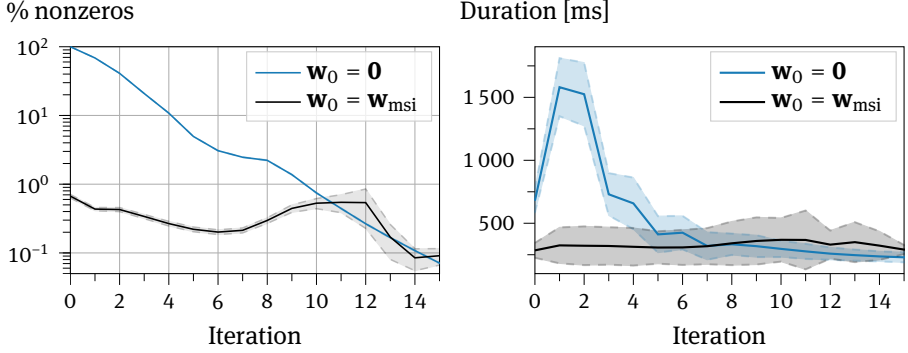


Fig. 6.16: Sparsity of the Hessian calculation $|\mathcal{E}|/N$ (left) and average duration of each optimization iteration (right) over the index of the iteration for zero and mean-separating initialization.

As $\bar{\mathbf{x}}$ only need be calculated once for the dataset, and $\bar{\mathbf{p}}$ can be computed quickly due to the data imbalance $|\mathcal{P}| \ll N$, these values can be computed efficiently.

This procedure can be viewed as an extreme case of data summarization (cf. Chapter 3), in which the entire dataset is reduced to just two instances. The idea is now to solve this very small classification problem, and use its solution as the starting point for solving the full problem. This will be particularly useful if the simple solution already classifies many of the easy negative instances correctly, as the set of hard examples \mathcal{E} will be small in such a case.

For two data points, the classification problem can be solved explicitly, and we call its result the *mean-separating initialization* (msi) vector \mathbf{w}_{msi} . This vector is the minimum-norm vector that attains pre-specified margins μ_+ for classifying the two data points. As a consequence, it lies in the plane spanned by $\bar{\mathbf{x}}$ and $\bar{\mathbf{p}}$, and is characterized by the following equations:

$$\mathbf{w}_{\text{msi}} = \alpha \bar{\mathbf{x}} + \beta \bar{\mathbf{p}}, \quad (6.20)$$

$$\bar{\mathbf{p}}^T \mathbf{w}_{\text{msi}} = \mu_+, \quad \bar{\mathbf{n}}^T \mathbf{w}_{\text{msi}} = \mu_-. \quad (6.21)$$

Heuristically, values $\mu_+ = +1$, $\mu_- = -2$ work well, and are based on the rationale that negative samples cover a larger volume in the instance space, and thus the initial decision boundary should be closer to the mean of the positives than to the mean of the negatives.

The efficacy of this method can be seen from Figure 6.16, which evaluates the two initialization strategies on the AmazonCat-13k [478] dataset using an AMD Rome 7H12 CPU.⁴ The data shows that

- starting from a zero initial vector, the fraction of nonzeros starts at 100 % and decreases as the training progresses;

⁴ Computational resources provided by CSC – IT Center for Science, Finland.

- starting from a mean-separating initial vector, the sparsity is already high in the beginning; and
- increased sparsity translates to significant reductions in computation time and corresponding energy savings.

In terms of wall-clock training duration t_{wc} , the speedup that can be achieved by switching from $\mathbf{0}$ to \mathbf{w}_{msi} , defined as $t_{wc}(\mathbf{0})/t_{wc}(\mathbf{w}_{msi})$, lies between 150 % and 500 % as shown in Table 6.5.

Feature-Sorting A large portion of the computation time is spent on calculating the initial margins $\mathbf{X}^T \mathbf{w}$ at the beginning of each iteration. Because \mathbf{X} is a sparse matrix, this computation has low arithmetic density, and because the feature dimension is typically very large, the vector \mathbf{w} does not fit into the L2-cache. These two properties mean that this operation is severely memory-bound.

The caching behavior of \mathbf{w} can be improved significantly by making use of the dataset characteristic – in particular the fact that typical XMC tf-idf datasets have a long-tailed distribution in the feature vector, meaning that some features have a large amount of non-zero entries, but most features have few non-zeros[29]. By sorting the feature indices according to the frequency of their occurrence, the corresponding entries in \mathbf{w} are brought closer together in the address space, thus improving the caching behaviour.

While this has no effect on the scaling of the performance with the thread count, it does induce an absolute speedup, as indicated by the dashed lines in Figure 6.17.

The Memory-Bottleneck On machines with many cores, the performance of the computations presented here is memory-bound. This can be seen in Figure 6.17, where despite the embarrassingly parallel nature of the computations, the performance scales sublinearly once a certain core count is exceeded.

In the specific case of running the computations on a 2-socket AMD Rome 7H12 (64 cores per CPU, cf. Figure 6.15) machine, Figure 6.17 shows almost perfect scaling from 8 threads, corresponding to 1 thread per NUMA node, up to 32 threads, corresponding to one thread per L3 cache. For higher thread counts, the speedup saturates and in some cases more threads may even be disadvantageous to performance.

In addition to the memory bottleneck, there will be a thermal/power bottleneck involved. The used CPU has a base clock of 2.6 GHz, but if only a few cores are used the clock frequency may be increased up to 3.3 GHz.⁵ This indicates that even without the memory bottleneck, the expected performance increase of 128 cores over 16 cores would be less than 16×. This shows that the sublinear scaling cannot be explained by

⁵ Given that the computations are memory-bound, even when running with 128 cores the execution ports of the CPU will be idle for a significant amount of time. Only moderate downclocking to ≈ 3.18 GHz occurred in our setup.

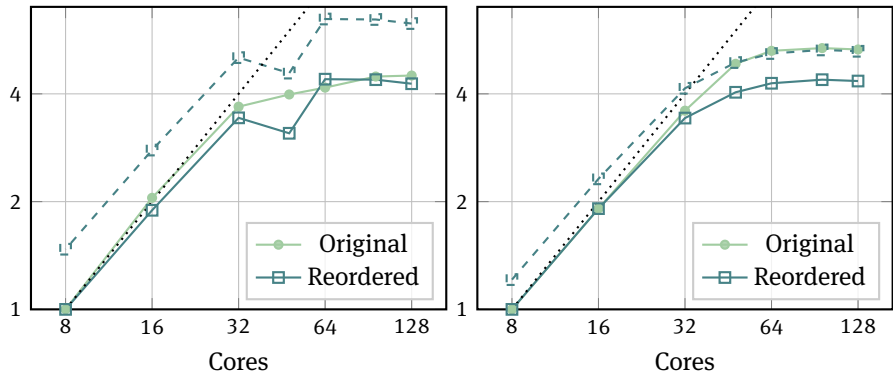


Fig. 6.17: Relative speedup for increasing thread counts using both original and reordered features for the first 10,000 labels of the Wikipedia-500k [50] dataset (left) and for the AmazonCat-13 [478] dataset (right). The dashed line shows the speedup of reordered features, normalized to the computation time with original features. The dotted line indicates perfect scaling. The (non-parallel) portion of the program run-time that is spent parsing the input dataset has been subtracted from the timings presented here.

Tab. 6.5: Training time (in hours) for zero and mean-separating initialization, as well as the number of non-zero weights (NNZ) after pruning (in millions) and their fraction. The experiments were run on a two-socket AMD Rome 7H12 machine.

Dataset	Zero	MSI	Speedup	NNZ	Fraction
Wiki10-31k	0.05	0.03	164 %	114	3.62 %
Amazoncat-13k	0.33	0.09	353 %	75	2.78 %
Amazoncat-14k	1.31	0.39	339 %	110	1.26 %
WikiTitles-500k	5.34	1.19	451 %	83	0.09 %
Amazon-670k	6.82	1.36	503 %	405	0.44 %
Delicious-200k	7.84	4.73	166 %	1175	0.73 %
WikiLSHTC-350k	18.52	5.17	358 %	434	0.08 %

Tab. 6.6: Results of DiSMEC in comparison with the state-of-the-art results as reported in March 2022 in Bhatia, Dahiya, Jain, Prabhu, and Varma [50], for selected XMC datasets.

Dataset/Metric	DiSMEC	SOTA Method	SOTA
Amazon-670K	—	—	—
P@1	44.7	LightXML	49.1
P@3	39.7	LightXML	43.8
P@5	36.1	LightXML	39.9
AmazonCat-13K	—	—	—
P@1	93.4	LightXML	96.8
P@3	79.1	LightXML	84.0
P@5	64.1	LightXML	68.7
Wikipedia-500K	—	—	—
P@1	70.2	AttentionXML	82.7
P@3	50.6	AttentionXML	63.8
P@5	39.7	AttentionXML	50.4
EURLex-4K	—	—	—
P@1	82.4	APLC-XLNet	87.7
P@3	68.5	APLC-XLNet	74.6
P@5	57.7	APLC-XLNet	62.3

reduced clock frequencies alone; rather, another resource, such as memory bandwidth, is also limiting performance.

6.3.4.1 Comparison With Deep Learning Methods

As shown in Table 6.6, the DiSMEC instantiation of the embarrassingly parallelizable one-vs-rest framework described in Algorithm 4 can be a competitive baseline. Its performance is not significantly worse in comparison to the state-of-the-art deep learning methods, which typically employ transformers encoders [174]. Unlike the deep learning models that require careful hyper-parameter tuning, the linear binary classification underlying DiSMEC is more readily interpretable and well-understood from a theoretical viewpoint. For sparse tf-idf data representation, linear XMC classifiers also work at par with tree-based approaches [371, 584] and those involving dense low-dimensional label embeddings [51, 279].

6.3.5 Summary and Outlook

In this section we presented linear classification algorithms for extreme multi-label classification. The linear model makes the training parallelize perfectly across different labels, though in practice the scaling levels out with too many cores in a single node. This is because even though the training itself does not require any communication

or synchronization between the threads, the different cores within a machine still compete for resources such as memory access. By placing a copy of the feature matrix in each NUMA domain, it can be ensured that each CPU can read its data from a part of the memory that it has fastest access to, and the load on the memory interface is spread out across the different NUMA domains. Additionally, by reordering the columns in the sparse feature matrix, data locality and, accordingly, cache efficacy can be improved. An implementation that combines techniques can be found at <https://doi.org/10.5281/zenodo.6699587>. For further discussion on the interaction between machine learning and the memory hierarchy, see Chapter 7.

The amount of work required to train the linear classifier for the highly imbalanced data typical for XMC can be drastically reduced by starting the weights from a good initialization. One way to find such a weight vector is to reduce the dataset to just two training instances, the centers of masses of the positive and negative training points in the original dataset, which can be calculated efficiently. By initializing the full training procedure with a weight vector that separates this summarized data, one can capitalize on the speedup of the conjugate-gradient optimizer due to implicit hard-instance mining of the hinge loss. This procedure can be seen as a variation of the *sketch-and-solve* principle introduced in Section 3.2. The main difference is that here the solution based on the sketch is used to initialize the full training, and thus no compromise in accuracy is made.

The presentation in the book is focused on the computational and implementation challenges of XMC problems. However, the scale of the label space also leads to interesting statistical consequences such as a long-tailed label distribution and corresponding data-scarcity for tail labels, as well as incomplete training data with missing labels. For a discussion of these issues, see the works of Babbar and Schölkopf [29], Jain, Prabhu, and Varma [336], and Qaraei, Schultheis, Gupta, and Babbar [587].

6.4 Optimization of ML on Modern Multicore Systems

Helena Kotthaus

Peter Marwedel

Abstract: This section demonstrates how the integration of knowledge about underlying hardware platforms and learning algorithms can provide results that would not be feasible by using the knowledge of only one type. In particular, this section presents the optimization of ML algorithms on multicore systems, and in this way addresses the same type of architectures as in Section 6.3. The optimization is based on resource-aware scheduling strategies for parallel machine learning algorithms. The focus is on Model-Based Optimization (MBO), also known as Bayesian optimization, which is an ML algorithm with huge resource demands, including a large number of computational jobs. Execution times of these jobs are estimated in order to enable their scheduling on parallel processors. The section demonstrates that this scheduling enables the processing of larger problem sizes within a given time budget and reduces the end-to-end wall-clock time for a constant problem size.

6.4.1 Motivation

The notion of resource-constrained systems is typically associated with small, integrated, and special-purpose devices exhibiting limitations with respect to, say, computational power, size, or battery life in embedded and cyber-physical systems. However, reducing the understanding of resource restriction to systems of this kind is not sensible. In fact, even high-performance computers and clusters can suffer from resource constraints when solving highly challenging problems that require massive amounts of resources [166, 666]. Therefore, it makes sense to consider resource constraints also for applications typically executed on larger systems.

Here, this is shown for the case of *parallel* MBO. MBO is a state-of-the-art global optimization method for black-box functions that are expensive to evaluate. To reduce the number of necessary evaluations of the black-box function, conventional MBO uses an iteratively refined regression model on a set of already evaluated configurations to approximate the objective function. However, such approaches neglect the heterogeneous resource requirements for evaluating different configurations in the model space, which often leads to inefficient resource utilization. This calls for new resource-aware scheduling strategies to efficiently map configurations to the underlying parallel architecture in accordance with their resource demands. In contrast to classical scheduling problems, the scheduling for MBO needs to interact with the configura-

tion proposal mechanism to select configurations with suitable resource demands for parallel evaluation.

The fundamentals and related approaches of parallel MBO are presented in Section 6.4.2. An overview of the RAMBO (Resource-Aware MBO) framework including the resource-aware scheduling strategies, as well as the corresponding evaluation results on homogeneous multiprocessor cluster systems, is given in Section 6.4.3. Section 6.4.4 proposes a concept for resource-aware scheduling strategies on heterogeneous embedded systems. The results are shown in Section 6.4.5.

6.4.2 Fundamentals and State of the Art for Parallel MBO

In machine learning, selecting the best algorithms for a given optimization problem and simultaneously tuning the corresponding hyperparameters of these algorithms can be very computationally intensive. Many strategies for hyperparameter optimization have been developed. (For an overview see, e.g., [55]). Hyperparameter optimization refers to finding the best configuration θ of a model, e.g., for a prediction problem a model with high predictive performance on an independent test set. When the evaluation of a single configuration already requires high resources, e.g., a very long runtime, then very wasteful optimization methods like evolutionary algorithms are not applicable. A popular approach for algorithm selection is F-racing [448], where a population of configurations is racing against each other and underperforming candidates are iteratively eliminated. This approach also requires many evaluations, at least in the early stage of the algorithm.

An established alternative in the situation of expensive time constraints is Model-Based Optimization (MBO), also known as Bayesian optimization, a state-of-the-art technique for expensive black-box optimization. In this optimization process, an unknown function, say, a machine learning algorithm, is evaluated to find the parameter configuration with the highest quality of the output measured by a given performance criterion within a limited time budget. This process is computationally challenging due to the huge parameter space that needs to be contemplated, and can result in extremely long response times. For this reason it is desirable to reduce the optimization time while maintaining the prediction quality, i.e., $\theta^* := \operatorname{argmin}_{\theta \in \Theta} f(\theta)$ for a search space Θ and an evaluation $f(\theta)$ of the black-box with input $\theta \in \Theta$ [348]. Aiming to reduce the number of evaluations of f required to find the best configuration θ^* , we used an iteratively refined and updated regression model (surrogate model), which attempts to approximate the black-box function by predicting $f(\theta)$ based on previous evaluations of f . During each iteration, a so-called *infill criterion* (acquisition function) proposes new promising configurations for evaluation.

In its original formulation, the MBO algorithm operates purely sequentially, proposing one configuration to be evaluated after the other [348]. For applications such as hyperparameter tuning for machine learning algorithms or computer simulations, the

parallelization of MBO has become of an increasingly interesting approach to reduce the overall execution time [288]. In order to propose multiple points (configurations) simultaneously in a parallel MBO setting, several modifications to the infill criteria or the general technique have been suggested. The modifications result in multiple configurations being proposed in each iteration [54, 254, 328]. The number of simultaneously proposed configurations is typically chosen to match the number of available CPU cores. However, these modifications in general neglect the heterogeneous resource requirements for evaluating different configurations in parallel. Depending on the parameter configuration of the applied machine learning algorithm, resource requirements such as CPU utilization or memory footprint usage can vary heavily [666].

The most important parallel extensions of MBO update the regression model either synchronously or asynchronously. Both variants are based on different infill criteria and have different advantages and drawbacks.

Synchronous Execution To allow for parallelization with a synchronous model update, infill criteria and techniques that propose multiple configurations in each iteration (constant liar, Kriging believer, qEI [254], qLCB [328], MOI-MBO [54]) have been suggested. *Multi-point proposals* are able to derive q configuration proposals $\mathbf{x}_1^*, \dots, \mathbf{x}_q^*$ simultaneously instead of only proposing one configuration \mathbf{x}^* from a *surrogate model*. Here, the model is updated after all evaluations within one iteration are finished. Hutter et al. [328] introduced the qLCB criterion, which is an extension of the single-point LCB criterion using an exponentially distributed random variable to generate q different candidate proposals by drawing random values of $\lambda_j \sim \text{Exp}(\lambda)$ ($j = 1, \dots, q$) from the exponential distribution:

$$\text{qLCB}(\mathbf{x}, \lambda_j) = \hat{\mu}(\mathbf{x}) - \lambda_j \hat{\sigma}(\mathbf{x}) \text{ with } \lambda_j \sim \text{Exp}(\lambda). \quad (6.22)$$

The λ variable guides the exploration-exploitation trade-off. Sampling multiple different λ_j might result in different “good” configurations by varying the impact of the standard deviation term.

Another popular multi-point infill criterion is the qEI criterion [254], which directly optimizes the single-point EI criterion over q points. As the computation of EI uses Monte Carlo sampling, it is quite expensive [136]. Therefore, a less expensive alternative, the *Kriging believer* approach [254], is often chosen. Here, the first configuration is proposed based on the standard single-point EI criterion. Its posterior mean value is treated as a real value of f to refit the surrogate, penalizing the surrounding region with a lower standard deviation for the next point proposal using EI again. This is repeated until q proposals are generated.

The above mentioned multi-point infill criteria can cause inefficient resource utilization when the parallel executed evaluations have heterogeneous execution times. Before new configurations are proposed, the results of all evaluations within one iteration are gathered to update the model. Thus the slowest evaluation becomes the

bottleneck, and all other parallel worker processes idle after finishing their evaluation before a new MBO iteration can start. However, performing the model updates only once per MBO iteration also leads to less computation overhead. Varying the execution times of parallel evaluations have already been addressed by Snoek et al. [635], who suggest that these be modelled with an additional surrogate, leading to an “expected improvement per second” favoring less expensive configurations. The resource-aware scheduling strategies for parallel MBO presented in this section also use regression models to estimate resource requirements, but instead of adapting the infill criterion, they use them to guide the scheduling of parallel evaluations. The goal is to guide MBO to interesting regions in a faster and resource-efficient way without directly favoring less expensive configurations.

Asynchronous Execution To avoid CPU idling, asynchronous execution replaces the evaluation of multiple configurations in batches, and the synchronous refitting of the model by refitting the model after each evaluation. Here, the number of worker processes equals the number of available CPU cores, but each worker proposes the next point for evaluation independently, even if configurations \mathbf{x}_{busy} are currently under evaluation on other CPU cores. The main challenge is to avoid evaluations of very similar configurations by modifying the infill criterion to deal with points that are currently under evaluation. The fast Kriging believer approach [254], which is based on EI (also used for multi-point proposals), can be applied to block these regions.

Another approach assessing pending values is the *Expected* EI (EEI) [253, 339, 635]. Here, the unknown value of $f(\mathbf{x}_{\text{busy}})$ is integrated out by calculating the expected value of y_{busy} via Monte Carlo sampling, which is, similar to qEI, computationally demanding. For each Monte Carlo iteration, values $y_{1,\text{busy}}, \dots, y_{\mu,\text{busy}}$ are drawn from the posterior distribution of the surrogate regression model at $\mathbf{x}_{1,\text{busy}}, \dots, \mathbf{x}_{\mu,\text{busy}}$, with μ denoting the number of pending evaluations. These values are combined with the set of already known evaluations and used to fit the surrogate model. The EEI can then simply be calculated by averaging the individual expected improvement values, which are formed by each Monte Carlo sample (n_{sim} denotes the number of Monte Carlo iterations):

$$\widehat{\text{EEI}}(\mathbf{x}) = \frac{1}{n_{\text{sim}}} \sum_{i=1}^{n_{\text{sim}}} \text{EI}_i(\mathbf{x}) \quad (6.23)$$

Besides the advantage of an increased CPU utilization, asynchronous execution can also cause additional runtime overhead due to the higher number of model updates and the computational costs for new point proposals, especially when the number of available CPU cores increases. Furthermore, the heterogeneous execution times of job configurations can lead to very similar point proposals due to model updates that are based on similar histories. Instead of using asynchronous execution to efficiently utilize parallel computer architectures, the new approach presented in this section uses the synchronous execution combined with resource-aware scheduling. The next

section includes a comparison of this approach (RAMBO) [385, 389, 390, 666] with the synchronous and asynchronous parallel variants of MBO described above.

6.4.3 Resource-Aware Scheduling Strategies

To enable the interaction between resource-aware scheduling strategies and the general MBO process, the RAMBO framework is proposed. Its foundations are based on the `m1rMBO` library [53]. The framework shown in Figure 6.18 aims to resource-efficiently reduce the end-to-end wall clock time needed by parallel MBO, and thus converge to the optimal configuration more rapidly. RAMBO consists of three main steps:

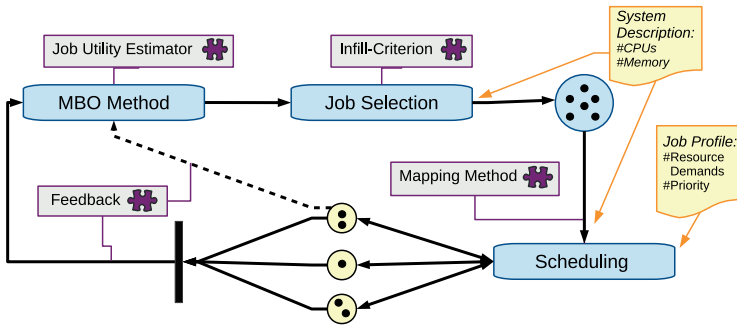


Fig. 6.18: Key steps (shown in blue) in the Resource-Aware Model-Based Optimization Framework [385]: building a regression model, selection of evaluation jobs, and job scheduling. Asynchronous execution (dashed line) and synchronous execution (solid lines) are possible.

First, a previously initialized regression model is built by the *MBO method*. Simultaneously, a *job utility estimator* creates profiles for the evaluations of configurations (*jobs*) by means of an additional regression model. These *job profiles* include runtime estimates, which are used as an input for the respective scheduling strategy later on. In conventional synchronous MBO approaches, such runtime estimates are not available. Hence, the slowest evaluation becomes a bottleneck within one MBO iteration, and the already finished parallel worker processes remain idle. As a consequence, the feedback of all idling processes and hence the model update is delayed.

The second step, i.e., the *job selection*, follows the MBO principles for configuration proposals. Typically, an *infill criterion* such as qLCB in equation (6.22) is used to propose configurations offering a proper compromise between the predicted outputs (*exploit*) and the uncertainty about the search space region (*explore*), i.e., that have a high potential to optimize the quality of the regression model. To this end, RAMBO provides mechanisms to interact with the job proposal mechanism by postponing or skipping suggested configurations that are deemed to be insufficiently promising or exhibit

unsuitable job profiles. As part of this process, a knapsack-based heuristic can be applied to select the most promising and suitable configurations.

Finally, a configurable *scheduling strategy* allocates the jobs to the available resources (system description) according to their particular resource demands. In addition, an execution priority based on the infill criterion is required to ensure that the model is updated i) with the most promising configurations and ii) as soon as possible. This model update follows the synchronous approach, i.e., it is performed when the results of all jobs executed within one MBO iteration are gathered. In a nutshell, the regression model is iteratively updated based on the results of all previous iterations until the runtime budget is exhausted.

Priorities for Job Selection To model the usefulness of a candidate for the objective function, Kriging is used as a surrogate regression model, and qLCB (6.22) is used as a multi-point infill criterion to generate a set of job proposals. Compared with the multi-point proposal qEI [254], the qLCB criterion is more suitable since it is able to propose a set of independent candidates. qLCB can simultaneously generate q candidates by drawing q random values of $\lambda_j \sim \text{Exp}(\lambda)$ ($j = 1, \dots, q$) from the exponential distribution. Each λ_j results in a different trade-off between exploitation ($\lambda_j \downarrow$) and exploration ($\lambda_j \uparrow$), and thus leads to a different optimal configuration \mathbf{x}_j^* after solving

$$\mathbf{x}_j^* := \underset{\mathbf{x}}{\operatorname{argmin}} [\text{LCB}(\mathbf{x}, \lambda_j)] = \underset{\mathbf{x}}{\operatorname{argmin}} [\hat{y}(\mathbf{x}) - \lambda_j \hat{s}(\mathbf{x})], \quad (6.24)$$

where $\hat{y}(\mathbf{x})$ denotes the posterior mean and $\hat{s}(\mathbf{x})$ denotes the root of the posterior standard deviation of the surrogate model at point \mathbf{x} .

Since the set of proposed candidates \mathbf{x}_j^* cannot be directly ordered by how promising a candidate is, an additional order is introduced to guide the search for the best candidate towards more promising areas. Therefore, the highest priority is given to the candidate \mathbf{x}_j that was proposed using the smallest value of λ_j and is thus closest to the optimum (exploitation). The priority for each job is defined as $p_j := -\lambda_j$.

However, qLCB does not include a penalty for the proximity of selected configurations, which might become a problem if the number of parallel evaluations is high. Therefore, the Euclidean distance is used to reprioritize p_j to \tilde{p}_j , encouraging the selection of configurations that are more scattered in the domain space.

First, a set of $q > m$ configurations is sampled from the qLCB criterion. These configurations are then hierarchically clustered by their distance in the domain space of the objective function using the complete linkage method. The procedure starts with the configuration that has previously been assigned the highest priority and assigns it to the first position in the list of selected jobs \tilde{j} . For each following step $i \geq 2$, all candidates are split into i clusters according to the hierarchical clustering. Of these i clusters the $i - 1$ clusters that already contain candidates with assigned positions are discarded, leaving one cluster. The position i in \tilde{j} is assigned to the job with the highest priority within this cluster. This goes on until all q candidates have assigned positions.

Thereby an ordering following the hierarchy induced by the clustering is generated. Finally, new priorities \tilde{p}_j are assigned based on the order of \tilde{j} , i.e. the first job in \tilde{j} gets the highest priority q and the last job gets the priority 1.

As a result, the set of candidates contains batches of jobs with similar priority, which are spread in the domain space. The priorities serve as input for the scheduling, which groups the q jobs to m CPU cores using the runtime estimates \hat{t} .

Resource Utility Estimation The runtime estimates of the set of jobs proposed in each MBO iteration are needed for the scheduling to avoid the execution of jobs with high runtime variances and thus to reduce idling worker processes. This is accomplished by using an additional regression model. As for the MBO algorithm itself, the runtime of a job is predicted in each iteration based on the runtimes of all previously evaluated jobs to build the runtime model of the black-box function. For the model, Kriging is used for homogeneous CPU systems since the runtime is expected to be a continuous function. For parallel architectures with heterogeneous CPUs, Random Forest is used for the model instead. Here, the runtime of a job is estimated for different CPU types (as described in Section 6.4.4). The accuracy of the runtime estimation also influences the scheduling decision. Therefore the runtime estimation quality is also included in the evaluation results.

Knapsack-Based Scheduling Strategy The goal of the knapsack-based scheduling strategy is also to reduce the CPU idle time on the workers while acquiring the feedback of the workers in the shortest possible time to avoid model update delay. Here the qLCB multi-point infill criterion is used to form a set of jobs $J = \{1, \dots, q\}$ that should be executed on the available CPU cores $K = \{1, \dots, m\}$. The estimated runtime is given by \hat{t}_j and the corresponding priority by p_j for each job proposal. The time bound for each MBO iteration (deadline) is defined by the runtime of the highest prioritized job. The goal is to maximize the profit, given by the priorities, of parallel job executions within each MBO iteration. To solve this problem, we apply the 0 – 1 multiple knapsack algorithm for global optimization routines [62]. Here, the knapsacks are the available CPU cores and their capacity is the maximally allowed computing time, defined by the runtime of the job with the highest priority. The items are the jobs J , their weights are the estimated runtimes \hat{t}_j , and their values are the priorities p_j . Accordingly, the capacity for each CPU core is \hat{t}_{j^*} , with $j^* := \operatorname{argmax}_j p_j$. To select the best subset of jobs, the algorithm maximizes the profit Q :

$$Q = \sum_{j \in J} \sum_{k \in K} p_j c_{kj}, \quad (6.25)$$

It is the sum of priorities of the selected jobs, under the restriction of the capacity

$$\hat{t}_{j^*} \geq \sum_{j \in J} \hat{t}_j c_{kj} \quad \forall k \in K \quad (6.26)$$

per CPU. The restriction with the decision variable $c_{kj} \in \{0, 1\}$ s.t.

$$1 \geq \sum_{k \in K} c_{kj} \forall j \in J, c_{kj} \in \{0, 1\} \quad (6.27)$$

ensures that a job j is at most mapped to one CPU.

The job with the highest priority defines the time bound (deadline) \hat{t}_j , and is thus mapped to the first CPU core $k = 1$ exclusively, while single jobs with higher execution times are directly discarded (discarded jobs will be proposed again in the next MBO iteration if they are promising enough). Then, the knapsack algorithm is applied to assign the remaining candidates in J to the remaining $m - 1$ CPU cores. This leads to the best subset of J that can be run in parallel, minimizing the delay of the model update. If a CPU core is left without a job, the regression model can be optionally queried for a job with an estimated runtime smaller or equal to \hat{t}_j to fill the gaps. Jobs having an estimated runtime *shorter* than the deadline, however, can lead to idle times if no other job can be executed within the time remaining until the next model update. The idle time resulting from suboptimal resource usage can be additionally exploited by enabling preemption and migration. More precisely, allowing jobs to be preempted and migrated to other cores provides the opportunity to fill unused time slots within an MBO iteration with high-priority jobs that would be skipped otherwise. Thus a larger set of jobs can be executed. The details of RAMBO's flexible migration mechanisms are described in [389].

Evaluation To evaluate the resource-aware MBO scheduling strategies included in the RAMBO framework, a comparison with different synchronous and asynchronous parallel MBO approaches was performed. The comparison included two asynchronously executed MBO strategies [253, 338] aiming to use all available CPU time to solve the optimization problem in parallel. Both of them used Kriging as a surrogate, with the EEI criterion 6.23 [339] and the Kriging believer [254] criterion. In Kotthaus et al. [390], RAMBO was also compared with a third asynchronous execution strategy, which is included in the SMAC (Sequential Model-based Algorithm Configuration) tool [329], using a random forest surrogate. The results showed that RAMBO and the two other asynchronous execution strategies always converged faster to the optimum compared to SMAC, which is why SMAC is not included in the following presentation. Besides the comparison with the asynchronous strategies, the following presentation also includes two synchronously executed MBO approaches. One of them used the qLCB multi-point infill criterion 6.22 and the other used the qEI criterion [254]. All parallel MBO approaches including the new RAMBO approach were evaluated on a set of established continuous synthetic functions combined with simulated execution times to ensure a fair and disturbance-free environment.

The usage of synthetic functions ruled out technical problems emerging on multi-user systems (swapping, network congestion, CPU cycle stealing, other users occupying

fast caches, etc.). Furthermore, synthetic functions eased the evaluation of MBO approaches on different difficulty levels. Two different categories of objective functions (implemented in the R library `smoof` [66]) were considered:

1. Functions with a smooth surface: `rosenbrock(d)` and `bohachevsky(d)` with dimension $d = 2, 5, 10$, which are likely to be fitted well by MBO.
2. Highly multimodal functions: `ackley(d)` and `rastrigin(d)` ($d = 2, 5, 10$), for which MBO is expected to have problems achieving good results.

For each objective function, a 2-, 5- and 10-dimensional version were used, each of which was optimized using 4 and 16 CPU cores in parallel to investigate scalability. Figure 6.19 visualizes the synthetic test functions for $d = 2$ [385].

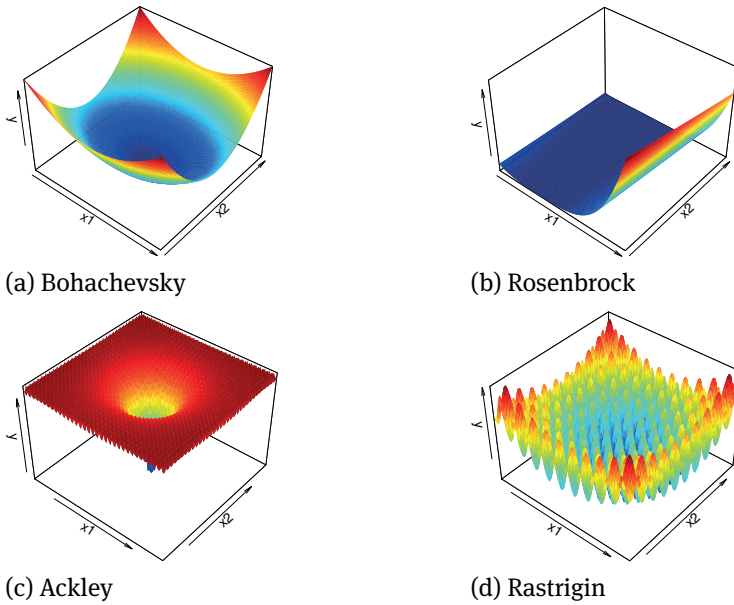


Fig. 6.19: Synthetic test functions used for the evaluation for $d = 2$. (a) and (b) show a smooth surface; (c) and (d) are highly multimodal [385].

Since synthetic functions are illustrative test functions, they have no significant runtime. Therefore, these functions were also used to simulate different runtime behaviors. For each benchmark two different synthetic functions were combined. One determines the number of seconds it would take to calculate the objective value of the other function. For example, for the combination `rastrigin(2).rosenbrock(2)` it would require `rosenbrock(2)(x)` seconds to retrieve the desired objective value `rastrigin(2)(x)` for an arbitrary proposed configuration \mathbf{x} . Technically, the benchmark

sleeps $\text{rosenbrock}(2)(\mathbf{x})$ seconds before returning the objective value. The runtime was simulated with either $\text{rosenbrock}(d)$ or $\text{rastrigin}(d)$, and all combinations of the four objective functions were analyzed except where the objective and the time function were identical. For the unification of the input space, values from the input space of the objective function were mapped to the input space of the function that simulated the runtime behavior. The output of the runtime functions were scaled to return values between 5 minutes to 60 minutes.

To examine how fast the parallel approaches converge to the optima of the benchmark functions within a limited time budget, the distance between the best found configuration at time t and a predefined target value (optimal configuration) was measured. This measurement reflects the accuracy of the receptive MBO approach within the given time budget. To make this measurement comparable for all objective functions, the function values were scaled to $[0, 1]$. Here, 0 is the target value, defined as the best configuration y reached by any optimization approach within the given time budget. The upper bound 1 is the best y found in the initial set of already evaluated configurations, and is identical for all approaches per given benchmark. Both values were averaged over 10 repetitions. If an optimization needs 2 hours to reach an accuracy of 0.5, this means that within 2 hours half of the way to the best configuration 0 has been accomplished, after starting at 1. The differences between the approaches were compared at the three accuracy levels 0.5, 0.1, and 0.01. The optimizations were repeated 10 times and conducted on $m = 4$ and $m = 16$ CPUs to examine scalability. Time budgets were 4 hours for 4 CPU cores and 2 hours for 16 CPU cores in total, including all computational overhead and CPU idling. All experiments were executed on a Docker Swarm cluster using the R library `batchtools` [415]. The initial set was generated by latin hypercube sampling [481] with $n = 4 \cdot d$ configurations, and all of the following optimizations start with the same initial set in all 10 repetitions:

- `rs`: Random search, served as a base-line.
- `qLCB`: Synchronously executed MBO using qLCB where in each MBO iteration $q = m$ configurations were proposed.
- `ei.bel`: Synchronously executed approach using Kriging believer where in each MBO iteration m configuration were proposed.
- `asyn.ei.bel`: Asynchronously executed MBO using Kriging believer.
- `asyn.eei`: Asynchronously executed MBO using EEI (100 Monte Carlo iterations).
- `rambo`: New synchronously executed MBO using qLCB with job priority refinement and the knapsack-based resource-aware scheduling strategy, $q = 8 \cdot m$ candidates proposed in each iteration.

Optimizations `qLCB` and `ei.bel` are implemented in the R library `m1rMBO` [53]. Optimizations `asyn.eei`, `asyn.ei.bel` and `rambo` are also based on `m1rMBO`. For all MBO approaches, a Kriging model was used from the library `DiceKriging` [605] with a Matern $\frac{5}{2}$ -kernel [474] and a nugget effect of $10^{-8} \cdot \text{Var}(\mathbf{y})$, where \mathbf{y} denotes the vector of all observed function outcomes.

The quality of resource-aware scheduling depends on the accuracy of the resource estimation. Without reliable runtime predictions, the scheduler is unable to optimize for efficient utilization. The runtime for all benchmarks was simulated with either `rosenbrock(d)` or `rastrigin(d)`. Figure 6.20 shows an example where the runtime estimation for the `rosenbrock(5)` time function works well (left part). Here, the residual values for the runtime estimation of the evaluated configurations decrease over time. However, the runtime prediction for `rastrigin(5)` (right part) is imprecise. For the 2- and 10-dimensional versions, the results are similar.

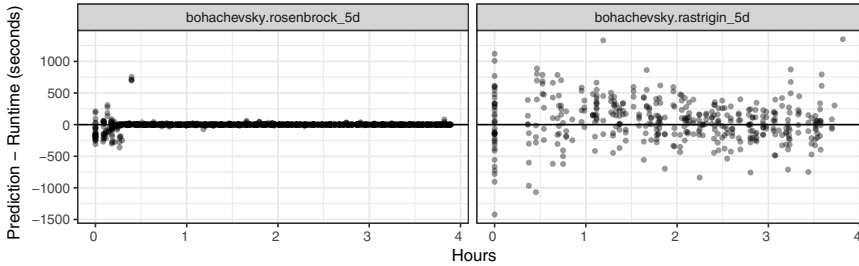


Fig. 6.20: Residuals of the runtime estimation over time for the `rosenbrock(5)` and `rastrigin(5)` time functions on 4 CPU cores combined with `bohachevsky(5)` as the objective function. Positive values indicate an overestimated runtime and negative values indicate an underestimation [385].

This encourages us to consider separate scenarios where runtime estimation has a high quality (`rosenbrock(·)`), and scenarios where runtime estimation is usually error-prone (`rastrigin(·)`s). In the following, we will focus on the scenario with high resource estimation quality. The evaluation results of the scenario with low runtime estimation quality can be found in [385] and are further optimized by a flexible scheduling mechanism [389].

Box plots for the time required to reach the three different accuracy levels in 10 repetitions within a budget of 4 hours on 4 CPU cores are shown in Figure 6.21, and within a budget of 2 hours on 16 CPU cores in Figure 6.22. The faster an approach reaches the desired accuracy level, the lower the box and the better the approach. If an approach was unable to reach an accuracy level within the given time budget, the respective time budget plus a penalty of 1 000 s is entered. Table 6.7 lists the aggregated ranks over all objective functions, grouped by approach, accuracy level, and number of CPU cores. For this computation, the approaches are ranked with regard to their performance for each repetition and benchmark before they are aggregated with the mean. If there are ties in Figures 6.21 and 6.21 (e.g., if an accuracy level was not reached), all values are assigned the worst possible rank. The benchmarks indicate an overall advantage of the new resource-aware MBO algorithm `rambo`. On average, `rambo` is always fastest. `rambo`

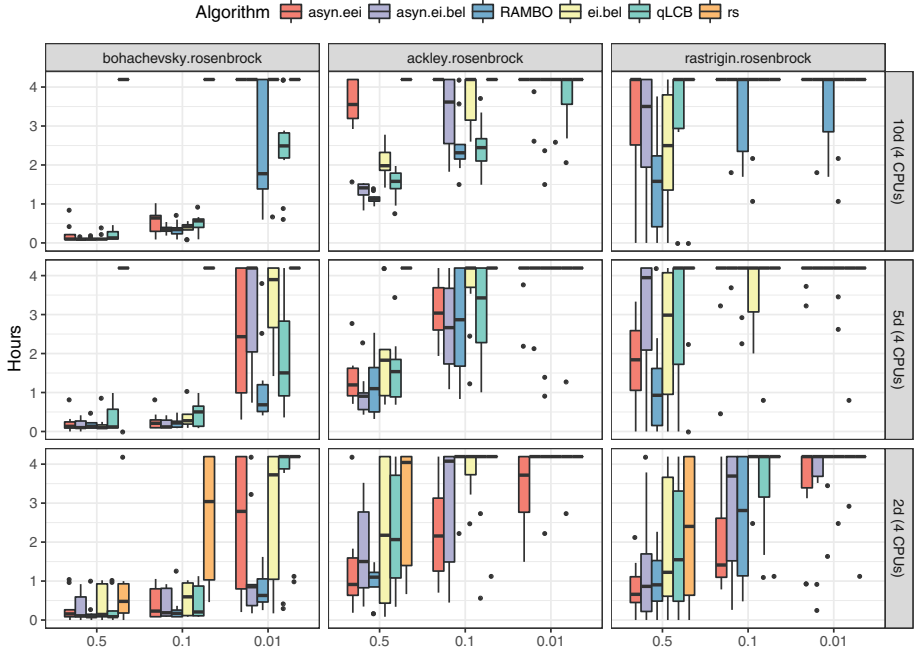


Fig. 6.21: Execution times on 4 cores as a function of the accuracy level for different objective functions using the time function `rosenbrock(.)` [385]. Execution times are low for moderate accuracy levels and favourable for RAMBO (shown in blue).

is closely followed by the asynchronous MBO variant `asyn.ei.bel` for accuracy levels 0.5 and 0.1 on 4 CPU cores but the lead becomes more clear on 16 CPU cores, especially for the highest accuracy level 0.01.

In comparison with the conventional synchronous MBO approaches `ei.bel` and `qLCB`, `rambo`, `asyn.eei`, and `asyn.ei.bel` reach the given accuracy levels in shorter time on 16 CPU cores. This is especially true for objective functions that are highly multimodal and thus hard to model (`ackley(.)`, `rastrigin(.)`) by the surrogate, as seen in Figure 6.22.

Table 6.7 shows that the less expensive `asyn.ei.bel` approach performs better than the computationally demanding `asyn.eei` on 16 CPUs. On 4 CPUs the synchronous `qLCB` approach is faster than the asynchronous approaches for the highest accuracy level 0.01. This result is influenced by the good performance of `qLCB` on functions with a smooth surface, as can be seen in Figure 6.21 in the 5- and 10-dimensional version of the `bohachevsky(.)` benchmark. When comparing the performance of the approaches for the 2-dimensional versus the 10-dimensional versions of the benchmarks, Figure 6.22 shows that the `rambo` approach outperforms all other approaches at higher dimensional problems compared with lower dimensions.

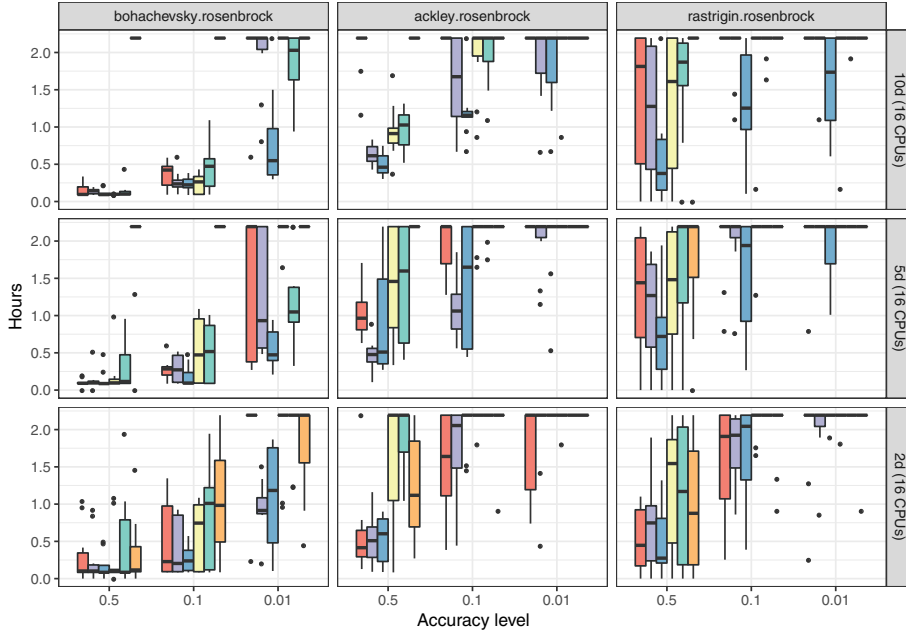


Fig. 6.22: Execution times on 16 cores as a function of the accuracy level for different objective functions using the time function `rosenbrock(.)` [385].

Figure 6.23 exemplarily visualizes the mapping of the parallel configuration evaluations (jobs) for all MBO approaches on 16 CPU cores for the 5d versions of the benchmarks. Each gray box represents the execution time of a job on the respective CPU. The gaps represent CPU idle time. For the synchronously executed MBO approaches `rambo`, `qLCB`, and `ei.be1`, the vertical lines represent the end of an MBO iteration. Red boxes indicate that the CPU is busy with a point proposal.

The necessity of a resource estimation for jobs with varying runtimes is obvious because the synchronous variants `qLCB` and `ei.be1` can cause long idle times by queuing jobs together with large runtime differences. The scheduling in `rambo` manages to reduce this idle time. This effect of efficient resource utilization increases with the number of CPUs. `rambo` reaches nearly the same effective resource utilization as the asynchronous approaches and at the same time reaches the accuracy level fastest. The Monte Carlo approach `asyn.eei` generates a high computational overhead as indicated by the red boxes, which reduces the effective number of evaluations. Here, the overhead for a new point proposal sometimes needs the same amount of time as the job evaluation. Idling occurs because the calculation of the EEI is encouraged to wait for ongoing EEI calculations to include their proposals. This overhead also increases with the number of evaluated points. By contrast, `asyn.ei.be1` has comparably low overhead and thus basically no idle time. This seems to be an advantage for `asyn.ei.be1` on 16 CPU cores,

Tab. 6.7: Execution times for accuracy levels 0.5, 0.1, 0.01 averaged over all benchmarks with the rosenbrock(·) time function on 4 and 16 CPU cores with a time budget of 4 hours and 2 hours, respectively [385]. Relative ranks within a column are included in parentheses.

Algorithm	4 CPUs			16 CPUs		
	0.5	0.1	0.01	0.5	0.1	0.01
asyn.eei	3.53 (3)	3.91 (3)	4.91 (3)	3.64 (3)	4.30 (3)	5.30 (3)
asyn.ei.bel	3.21 (2)	3.66 (2)	5.04 (4)	2.93 (2)	3.31 (2)	4.48 (2)
rambo	2.47 (1)	3.40 (1)	4.23 (1)	2.54 (1)	2.98 (1)	3.72 (1)
ei.bel	3.64 (4)	4.36 (5)	5.31 (5)	3.81 (4)	4.57 (4)	5.70 (5)
qLCB	4.02 (5)	4.24 (4)	4.83 (2)	4.27 (5)	5.04 (5)	5.40 (4)
rs	5.57 (6)	5.89 (6)	5.89 (6)	5.17 (6)	5.71 (6)	5.82 (6)

where on average it performs better on all accuracy levels than the computationally demanding asyn.eei, especially for higher dimensional problems.

Observations rambo outperforms the conventional synchronous MBO. The resource utilization obtained by the scheduling in rambo leads to faster and better results, especially when it comes to increasing problem dimensions (configurable parameters) and increasing numbers of available CPU cores. On average, rambo converges faster to the optimum than all considered asynchronous approaches. This indicates that the resource utilization obtained by the RAMBO approach improves MBO, especially when the number of available CPU cores increases. Predictable runtimes can be assumed for real applications like hyperparameter optimization for machine learning methods, even if the runtime estimation quality is difficult to determine in advance. The results also suggest that, on some setups, the choice of the infill criterion determines the parallelization strategy for better performance.

6.4.4 Scheduling Strategies for Heterogeneous Architectures

As described in Section 6.4.3, the resource-aware scheduling for MBO uses two inputs: the estimated resource utilization and the priority of the proposed candidates. While the priority of a candidate is computed as described above, the estimation of the resource utilization needs to be enhanced for heterogeneous systems.

Resource Estimation for Heterogeneous Systems The regression model used to estimate the execution times of the candidates was previously based on Kriging; now Random Forest is applied instead. Random Forest is more suitable for heterogeneous systems since the job execution times build up a discontinuous model due to the additional categorical variable that represents the processor type. The regression model now needs to estimate the runtime \hat{t}_j for each candidate in the proposed set of jobs $J =$

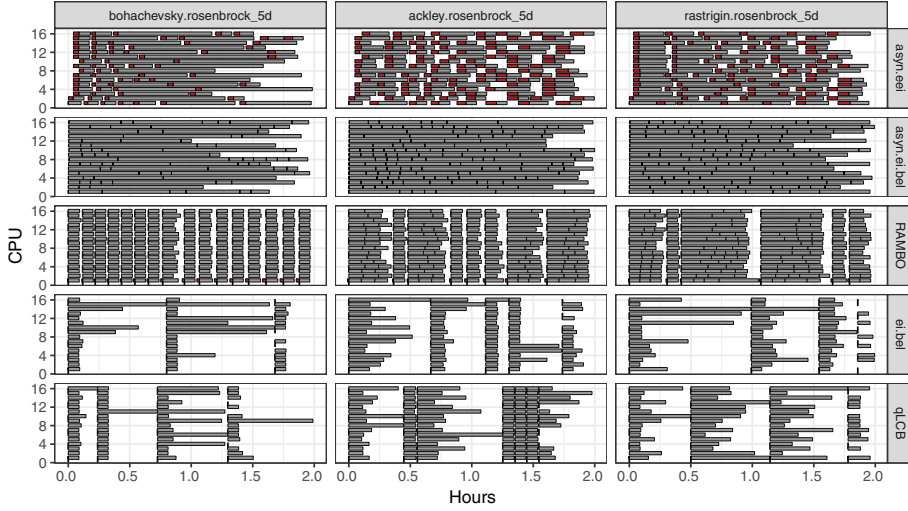


Fig. 6.23: Scheduling of MBO algorithms: Time shown on x -axis and mapping of candidates to $m = 16$ CPU cores on y -axis. Each gray box is a job. Each red box represents the overhead of the point proposal. The gaps represent CPU idle time [385].

$\{1, \dots, q\}$ and for each available CPU core $K = \{1, \dots, m\}$. This is required since the execution of a job is processor-dependent. If the underlying heterogeneous architecture is known, the number of runtime estimates per job can be reduced to the number of different processor types. Thus the runtime of a job $j \in J$ is predicted for each available processor type $k \in K$ in each MBO iteration based on the runtime of all previously evaluated jobs to build the runtime model of the black-box function and is therefore denoted as \hat{t}_{kj} .

Knapsack-Based Scheduling To apply the 0 – 1 multiple knapsack algorithm for scheduling on heterogeneous architectures, the original formulation from Section 6.4.3 needs to be extended. Now the items representing the jobs J have different weights, represented by the different runtime estimates \hat{t}_{jk} per processor type k . Since the capacity of the CPU cores is now heterogeneous, a reformulation is needed. For this purpose, a ratio variable representing an approximated ratio of the runtime differences produced by the different processor types is introduced.

To minimize the delay of the model update with the results of the most promising candidate, the job with the highest priority $j^* := \operatorname{argmax}_j p_j$ is now always placed on the CPU core $k^* := \operatorname{argmin}_k \hat{t}_{kj^*}$ leading to the shortest estimated runtime for j^* . The capacity for the remaining CPUs, and thus the time bound for each MBO iteration, is accordingly defined by the shortest estimated runtime of the highest prioritized job \hat{t}_{k^*,j^*} . We introduce the ratio variable $\hat{t}_{k^*,j^*}/\hat{t}_{kj^*}$ representing the runtime difference of the highest prioritized job on the remaining k CPU cores.

The assumption that runtimes on different CPU types differ by a constant factor goes back to the uniform processor model described by Pinedo, which is a simplified model of real hardware [579]. For example, one CPU might offer vector instructions that some jobs benefit from, whereas others make no use of them. Instead of relying on statically precomputed ratios (such as those derived from the ratio of CPU frequencies), the selected job j^* is used as the “benchmark” for comparing CPU speeds in a given MBO iteration, under the assumption that in this iteration the speed on CPU k differs from k^* by a factor of $\hat{t}_{k^*j^*}/\hat{t}_{kj^*}$. The formulation of the restriction of the capacities for the remaining CPU cores is thus as follows, while the rest of the knapsack algorithm remains as described in Section 6.4.3:

$$\hat{t}_{k^*j^*} \frac{\hat{t}_{k^*j^*}}{\hat{t}_{kj^*}} \geq \sum_{j \in J} \hat{t}_{kj} c_{kj} \quad \forall k \in K. \quad (6.28)$$

Here, the estimated execution times of the remaining candidates on the fastest CPU core \hat{t}_{k^*j} on the right-hand side of equation 6.28 is expected to be approximately similar to the estimated runtime of a job on the remaining CPU cores \hat{t}_{kj} , multiplied by the ratio variable:

$$\hat{t}_{k^*j} \approx \hat{t}_{kj} \frac{\hat{t}_{k^*j^*}}{\hat{t}_{kj^*}} \quad \forall k \in K, \forall j \in J. \quad (6.29)$$

This formulation is needed to reduce the number of weights (number of runtime estimates per CPU type) per item j to a single weight \hat{t}_{k^*j} in order to apply the original knapsack algorithm.

Evaluation The effectiveness of the heterogeneous RAMBO approach is evaluated by targeting the ARM big.LITTLE architecture⁶ of the Odroid-XU3 platform,⁷ which is also commonly found in mobile devices. This platform is equipped with four “big” Cortex A15 CPUs (quad-core) with a frequency that can be scaled up to 2.0 GHz and four “little” Cortex A7 CPUs that have about half the processor speed (1.4 GHz). The Odroid-XU3 platform also includes a Mali-T628 GPU (not considered here) and 2 GB of main memory. For the evaluation of RAMBO on heterogeneous processing architectures, not only the runtime that is needed to find the best possible configuration is examined but also the energy consumption. This is accomplished by reading from the power measurement sensors INA231 offered by the Odroid-XU3 platform, which report energy consumption for both processor types as well as for the RAM and the GPU. To measure the energy and power consumption of the resource-aware scheduling strategy and its competing MBO approaches, a so-called Relay Reader [526] is used to read out the sensor data in regular

⁶ ARM big.LITTLE Technology: <https://developer.arm.com/technologies/big-little> (accessed Feb. 22nd, 2022).

⁷ Odroid-XU3: <https://developer.arm.com/graphics/development-platforms/odroid-xu3> (accessed Feb. 22nd, 2022).

intervals of approximately one second via threads for both CPU types. These threads are executed on separate CPUs and do not influence the runtime measurements.

The experimental setup consists of a subset of the setup described in Section 6.4.3. RAMBO is compared with the conventional synchronous MBO approach using the qLCB multi-point infill criterion and with the asynchronous MBO approach, which aims to exploit all available CPU time to solve the optimization problem in parallel and using the Kriging believer criterion [254]. All MBO approaches are evaluated on the 2-dimensional versions of the synthetic functions and executed on 4 CPU cores. The runtime of the objective functions was previously simulated by sleeping for a given time, determined via an additional synthetic function that represented the runtime behavior of the respective objective function. For the power consumption measurements, a real computation is needed. This is accomplished by repeatedly executing a function that draws random numbers. The runtime of this real computation is still controlled via an additional synthetic function that defines the number of repetitions and simulates the time that is needed for calculating the objective value. For the synthetic function that simulates the runtime of the objective functions, the `rosenbrock(d)` function is used, since it delivers a more reliable runtime estimation than `rastrigin(d)` (see Figure 6.20). The output of the `rosenbrock(2)` function is scaled to return values from 5 min to 50 min. The MBO approaches run for 2 hours on $m = 4$ CPU cores, and include all computation overhead and CPU idling. The initial set is generated as with the homogeneous experiments by using the latin hypercube sampling [481] with $n = 4 * d$ configurations. All approaches start with the same initial set in all 10 repetitions.

Tab. 6.8: Ranking for accuracy levels 0.5, 0.1, 0.01 averaged over all problems with `rosenbrock(2)` time function on 4 CPU cores with a time budget of 2 hours [385].

Algorithm	0.5	0.1	0.01
<code>rambo</code>	1.90 (1)	1.77 (1)	1.90 (1)
<code>asyn.ei.bel</code>	2.07 (2)	2.43 (2)	2.63 (2)
<code>qLCB</code>	2.67 (3)	2.63 (3)	2.70 (3)

Table 6.8 lists the aggregated ranks over all 2-dimensional objective functions, grouped by accuracy level. As described in Section 6.4.3, the approaches are ranked with regard to their performance for each of the 10 repetitions and for each benchmark before they are aggregated into the mean. Figure 6.24 shows the corresponding box plots for the time required to reach the three different accuracy levels, as described in Section 6.4.3. The faster an approach reaches the desired accuracy level, the lower the displayed box and the better the approach.

The benchmarks indicate an overall advantage of the new knapsack-based algorithm for heterogeneous systems, especially for the highest accuracy level 0.01. On

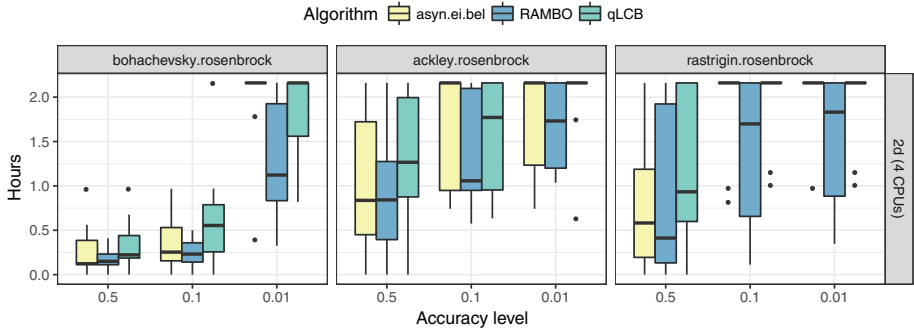


Fig. 6.24: Execution time as a function of the accuracy level for the 2-dimensional objective functions using time function rosenbrock(2) (lower is better) [385].

average, rambo is always fastest in reaching each of the three accuracy levels, and thus converges faster to the optimum in the time budget of 2 hours. In comparison with rambo, the conventional synchronous MBO approach qLCB is unable to reach the accuracy level 0.01 for the rastrigin(2) and ackley(2) functions in all 10 repetitions (see Figure 6.24). The same can be said about the asynchronous MBO approach asyn.ei.bel for the bohachevsky(2) and rastrigin(2) functions.

Figure 6.25 shows the box plots for the energy consumption over all 10 repetitions for each benchmark on each CPU type (upper part, Cortex A7, and Cortex A15) and over all CPUs (lower part, combined). Low boxes indicate a small energy consumption. The results indicate that rambo consumes more energy than the default qLCB approach on the “slow” Cortex A7 CPU cores, while it consumes less energy on the “fast” Cortex A15 CPU cores. In comparison with the asyn.ei.bel approach, rambo manages to consume less energy on the “slow” Cortex A7 CPU cores. The reason for the higher energy consumption of rambo compared with the synchronous qLCB approach on the “slow” Cortex A7 cores (see upper part of Figure 6.25) lies in the resource-aware scheduling strategy, which is able to utilize the less energy consuming A7 CPU cores more efficiently by mapping jobs to specific cores. Furthermore, only jobs with a runtime smaller or equal to the job with the highest priority are executed within one MBO iteration. Accordingly, longer running jobs with a lower optimization potential are discarded and more MBO iterations can be performed in the given time budget. By contrast, qLCB is not able to map jobs to specific CPU cores; it starts four jobs on the 4 available CPU cores that were proposed by the infill criterion in each MBO iteration, without respect to the heterogeneity of the underlying architecture and the job execution times.

Another contributing factor to the higher energy consumption of the qLCB approach is that it executes more jobs on the more energy-consuming A15 CPU cores due to the OS scheduling. Within one MBO iteration, the OS scheduler migrates jobs from a “slow” A7 CPU to a “fast” A15 CPU for cases where a job on a fast CPU finishes earlier than a job on a slow CPU. This speeds up computation and thus executes more MBO-

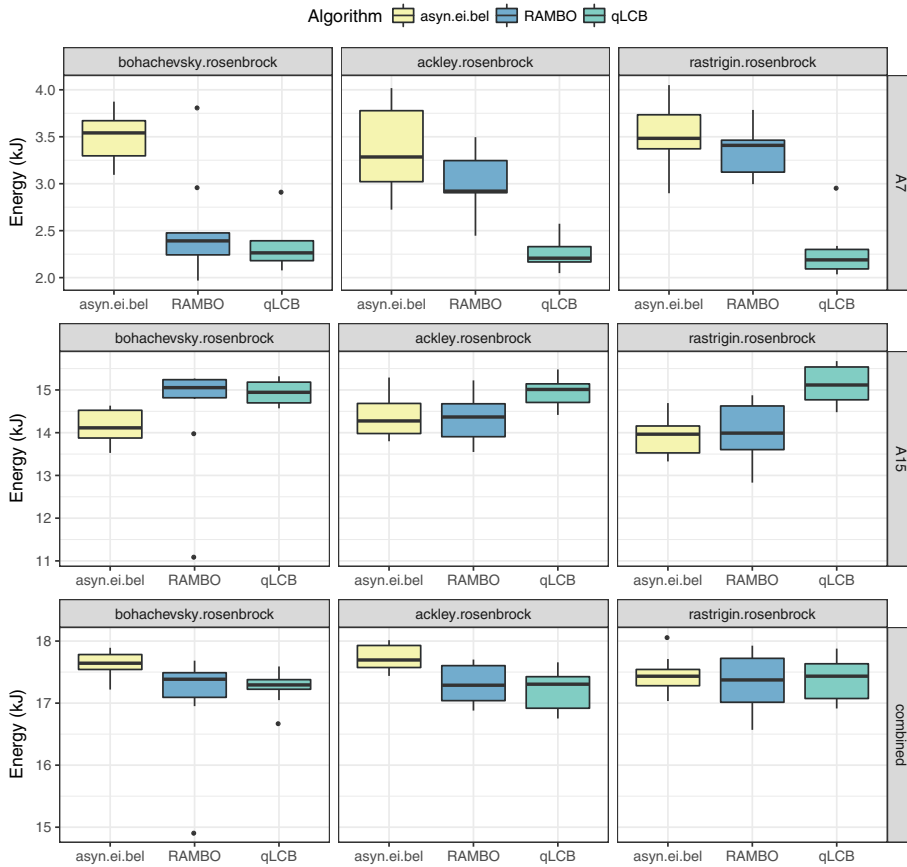


Fig. 6.25: Energy consumption in kJ on the two A15 CPUs (2.0 GHz), the two A7 CPUs (1.4 GHz), and combined consumption on both CPU types across all 10 repetitions for each objective function, with rosenbrock(2) time function and a time budget of 2 hours (lower is better) [385].

iterations. Hence, qLCB has nearly no idle time on the A15 CPU cores. However, the conventional synchronous approach only performs approximately half as many MBO iterations as rambo. In general, rambo executed more job evaluations in the given time than both competing MBO approaches. However, the combined energy consumption on all four CPU cores depicted in the lower part of Figure 6.25 shows that rambo consumes approximately the same amount of energy as qLCB, while it consumes less energy than asyn.ei.bel for the bohachevsky(2) and ackley(2) benchmark functions.

The asynchronous asyn.ei.bel approach in most cases consumes more energy than rambo since it has nearly no CPU idle time. However, it still converges more slowly to the optimum. The reason for this is that rambo selects more promising candidates with shorter runtimes since it executes only jobs with a runtime shorter than or equal

to the most promising candidate, and thus aims to find the cheapest way of evaluations through the model.

Overall, the results show that the resource utilization obtained by the scheduling for heterogeneous architectures in `rambo` enables MBO to converge faster to the optimum without consuming more energy resources than the competing approaches.

6.4.5 Summary: Resource-Aware Scheduling for ML on Multicores

We presented resource-aware scheduling strategies for parallel machine learning algorithms on multicore systems. The resource-aware model-based optimization framework RAMBO was introduced and evaluated. RAMBO can fully use the potential of parallel architectures. This was accomplished with an estimation model for the runtimes of each evaluation of a black-box function to guide the scheduling of configurations to available resources. In addition, an execution priority reflecting the estimated profit of a black-box evaluation was used to guide MBO to interesting regions in a faster, resource-efficient way without directly favoring less expensive configurations. The evaluation results showed that RAMBO converged faster to the optimum than the existing parallel approaches. RAMBO was especially efficient for complex high-dimensional problems, and strongly improved upon the existing approaches in terms of scalability when the number of available CPU cores was increased. Overall, it was shown that the integration of knowledge from the theory of using the underlying hardware (like scheduling) with knowledge about machine learning algorithms achieved results that would not have been feasible without crossing the boundaries of traditional knowledge areas.

6.4.6 Conclusion

The advantage of linking information on underlying hardware platforms with algorithmic knowledge is assumed to exist not only in this particular case. Lowering the walls between disciplines is likely to provide benefits in other cases as well.