Workload Placement on Heterogeneous CPU-GPU Systems

Marcos N. L. Carvalho
UPC, NKUA, Athena RC
marcos.nogueira@upc.edu

Alkis Simitsis
Athena Research Center
alkis@athenarc.gr

Anna Queralt
UPC, BSC
anna.queralt@upc.edu

Oscar Romero
UPC
oscar.romero@upc.edu

ABSTRACT

The popularity of heterogeneous CPU-GPU processing has increased considerably in recent years. To efficiently utilize heterogeneous resources, data processing systems depend on an appropriate workload placement strategy to assign the right amount of compute to the right processor. However, finding an optimal placement strategy is not trivial due to various complex and conflicting trade-offs related to the characteristics of processors, the nature of the workload, and data locality. In addition, placement decisions impact workload runtime and performance cost, and also depend on the availability of potentially different implementations for CPUs and GPUs, which adds extra complexity in such heterogeneous environments. In this tutorial, we review and compare state-of-the-art strategies for workload placement on heterogeneous CPU-GPU architectures, along with runtime prediction techniques and methods to support multi-device code. We also discuss open issues and identify potentially promising future research directions.

PVLDB Reference Format:
Marcos N. L. Carvalho, Alkis Simitsis, Anna Queralt, and Oscar Romero. Workload Placement on Heterogeneous CPU-GPU Systems . PVLDB, 17(12): 4241 - 4244, 2024.
doi:10.14778/3685800.3685845

1 INTRODUCTION AND MOTIVATION

In the last decade, graphic processing units (GPUs) have gained popularity in applications across diverse research domains, such as Data Science, Artificial Intelligence, or HPC. GPUs provide a high level of parallel processing power to accelerate applications, but they also present three core limitations: *(i) Data transfer bottleneck:* data must be moved between CPUs and GPUs, typically, over a low-bandwidth system bus which adds a communication overhead *(ii) Limited memory size:* Although the GPU memory has a considerable greater bandwidth, its size is much smaller than the main memory available for CPUs, preventing the processing of high volumes of data on GPUs; and *(iii) CPUs and GPUs are designed for different purposes:* CPUs excel at processing low-latency operations with low degree of thread parallelism, while GPUs process best operations with a high degree of thread parallelism.

Hence, heterogeneous CPU-GPU architectures have emerged as a strategy to alleviate the drawbacks of GPUs and exploit the benefits of both CPUs and GPUs. Optimizing the utilization of heterogeneous CPU-GPU systems is a challenging research problem that requires (a) finding the optimal workload placement strategy (i.e., on what processor, CPU or GPU, to run part of the computation of an application), (b) estimating runtime costs over different hardware architectures, and (c) managing multi-device code [25].

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685845

Determining the most efficient placement strategy is a complex task, which requires balancing the trade-off between optimal execution (i.e., assign a workload to a processor) and mitigated data transfers (i.e., employ a single processor to avoid data move) [29]. However, the large space of non-linear execution parameters (a.k.a. factors) involved in the design process and the lack of concrete guidelines often result in subpar placement decisions that cause load imbalance, waste resources, and amplify the data transfer bottleneck. For estimating runtime costs, current works consider a blend of techniques based on heuristics, cost models, and learn models. For managing heterogeneous CPU-GPU code, kernel templates, compilers, and raw kernels are the typical alternatives.

Although a significant body of research has been accumulated in recent years, a comprehensive study of the related work and current best practices is missing. Past surveys provide an excellent overview of the problem, but do not delve into the specifics of the CPU-GPU workload placement challenges and techniques [32, 37]. As this is a truly interdisciplinary topic, we believe it would be of interest to the research community to be informed about the advances on this hot problem in areas such as systems, HPC, and data management.

2 TUTORIAL SCOPE AND COVERAGE

In this tutorial, we present and compare the state-of-the-art solutions for workload placement on heterogeneous CPU-GPU systems, and answer the following questions: (a) what have we learned about workload placement on such architectures and what is still missing, (b) how placement strategies balance choosing processors and data locality, (c) what are the techniques used to estimate costs and support placement decisions, and (d) what are the alternatives to manage heterogeneous CPU-GPU code. Here, the term ‘workload’ covers a variety of data processing functions, including database queries, data flows/pipelines, or general apps and programs.

Duration. We propose a 90-min tutorial structured as follows:

(1) Introduction and motivation [∼10’]
(2) A taxonomy for CPU-GPU workload placement [∼10’]
(3) Workload placement strategies [∼24’]
(4) Estimating costs for placement decisions [∼18’]
(5) Managing heterogeneous CPU-GPU code [∼18’]
(6) Open issues and research directions [∼10’]

Tutorial scope. The literature about heterogeneous CPU-GPU processing is wide and several approaches have explored different *GPU usage*, i.e., primary processor, accelerator, or heterogeneous CPU-GPU processing, and *GPU integration*, i.e., integrated GPU (iGPU) or dedicated GPU (dGPU, or hereafter, simply referred as GPU) [18]. In this tutorial, we focus on heterogeneous CPU-dGPU processing solution approaches as this is the most popular setting in both server and high performance computing infrastructures [35].

3 THE CPU-GPU LANDSCAPE

GPU usage. Applications using GPUs, either as a primary processor or as an accelerator, typically assume that the workload placement decision is done beforehand (static placement) [e.g., 4, 6, 23]. Most practical applications, however, require dynamic (and often, adaptive) workload placement, which is a challenging problem [32].
GPU integration and data transfer bottleneck. iGPUs have limited memory bandwidth (they share the main memory with CPUs) and reduced processing capacity compared to dGPUs. However, iGPUs are more energy efficient and do not face a data transfer bottleneck. Hence, iGPUs are popular in fine-grained workloads (e.g., stream processing or processing in edge devices) [32]. On the other hand, dGPUs have limited memory size and deal with a data transfer bottleneck as their memory is decoupled from the main memory, hence, requiring data transfer over a low-bandwidth interconnect. However, their high bandwidth memory and processing capacity make dGPUs ideal to process coarse-grained and compute-intensive workloads (e.g., machine learning, numeric algorithms) [32]. The size limitation of dGPUs memory is being gradually mitigated (e.g., NVIDIA H200 offers 141GB memory [38]); still, it remains much smaller than the system main memory, which can be in the order of terabytes in modern servers. Price-wise, GPU memory (GDDR6) current $/GB ratio (100$/GB) resembles the price per GB of DRAM a decade ago (60$/GB) and it is steadily improving [24].
Data transfer techniques and workload placement. Several software/hardware solutions aim at mitigating the CPU-GPU data transfer bottleneck [32]. For example, data locality-based solutions avoid data transfer by placing workloads on processors where the input data is already located [8, 12, 13, 26, 34]. Such solutions use off-chip caching policies to avoid slow disk accesses, and are typically used in placement strategies that consider data locality [32]. Other placement strategies ignore data locality and focus on placing workloads on the processor that results in faster runtime, potentially at the cost of most expensive CPU-GPU data transfer.

3.1 A Taxonomy of Solutions

We studied 77 papers on workload placement on heterogeneous CPU-GPU systems, spanning a period of 15 years (2009-2024). In our analysis, a number of factors consistently stand out in all papers. Based on these, we classify the state of the art using the following dimensions: (a) placement strategy, (b) off-chip cache policy, (c) placement granularity, (d) placement time, (e) placement decision, (f) placement prediction model, (g) monitored metrics, and (h) GPU programming method. For space considerations, we do not list all 77 papers here. Figure 1 shows an abridged taxonomy indicating representative works[1] for each dimension and also how many papers consider the said dimension (the #papers row –e.g., 9 papers employ function shipping).

3.1.1 *Workload placement strategies.*

Workload placement and scheduling are similar concepts, but they have different design purposes. While placement strategies decide where to run a workload (CPU or GPU processor), scheduling strategies decide where (specific CPU core or GPU device) and when (the execution order considering resource utilization) to run a workload. In general, placement strategies work on an optimization layer providing hints to schedulers about the ideal processor type to run workloads.
Placement strategy. Early approaches focused on *data shipping* [e.g., 18, 19, 28, 30], where functions (i.e., applications or fractions of an application) are first assigned to the most suitable processors (i.e., those that result in smallest estimated costs) and then data is placed on the processors where functions have been assigned at the cost of increasing the data movement overhead. Conversely, *function shipping* [e.g., 8, 13, 26] places functions where data is already located to reduce data transfer at the cost of not always placing functions on the most fit processor. Selecting the best approach is not trivial and highly depends on the application and resources.
Off-chip cache policy. Function shipping approaches typically employ cache to keep a subset of data on off-chip memory (i.e., host main memory for CPU or device global memory for GPU) and reduce data movement costs [8, 26]. GPU cache has higher bandwidth but smaller size than CPU cache. Thus, besides finding a cache policy that selects a subset of data to fit into the limited cache size, choosing a cache type is also a challenge. Popular cache policies include *frequency-based* [e.g., 10, 24, 26], *greedy-based* [e.g., 12, 15, 16], and *semantic-aware* [e.g., 1, 8, 13] that attempt to cache the data with the most impact on task execution. *No off-chip caching* [e.g., 5, 20, 28, 40] is commonly used in data shipping approaches.
Placement granularity. *Job-level* granularity [e.g., 31] comprises a set of applications to be placed together on CPUs or GPUs, whereas *application-level* granularity [e.g., 4, 5, 24] involves the placement of a single application. Applications can be partitioned logically and/or physically into *tasks* [e.g., 12, 18, 20, 33]. In logical partitioning, portions of code of the application are identified as tasks and placed on CPUs or GPUs. In physical partitioning the input data of an application is divided into blocks and processed as multiple independent tasks. Tasks can be grouped and placed in *pipelines* [e.g., 13, 21, 23], logically partitioned on *functions* [e.g., 14, 29] or physically partitioned on segments (i.e., groups of task input data) [e.g., 8] or bits (i.e., the finest granularity of task input data) [39]. Coarser granularity allows placement decisions to generalize to multiple applications, whilst finer granularity involves application-specific factors (e.g., kernel-level). Most approaches however opt for a balanced granularity, considering task-level placement.
Placement time. *Runtime* (a.k.a. dynamic or local) placement [e.g., 7, 17, 26, 33] performs placement decision at workload execution based on runtime factors (e.g., current load, memory usage). This strategy deals better with unforeseen events (e.g., out-of-memory errors, heap contention). However, the choice of factors to consider is limited, which often leads to sub-optimal placement [28]. *Compilation time* (a.k.a. static or global) placement [e.g., 2, 4, 5, 23] decides placement before workload execution and typically, it does not handle well unforeseen events. This strategy relies on complex placement algorithms that achieve more robust performance at the cost of a higher implementation effort and computational overhead. A typical limitation includes the estimation of runtime factors (e.g., query cardinality), which oftentimes is not accurate [28]. *Hybrid placement strategies* combining runtime and compilation time placement have also been considered [e.g., 14, 21, 22, 30, 34].

[1]'Representative' stems from a blend of -sometimes conflicting- criteria including most cited, most recent, first to introduce, etc. Our aim is to highlight the techniques through the papers, so although a different presentation of papers would be possible, it would not change the gist of the analysis presented.

Dimension Group	Workload Placement Strategies																			Cost Estimatition							Het. Code Mgmt.		
Dimensions	Plmt. Strategy		Off-chip Cache Policy				Plmt. Granularity							Plmt. Time			Plmt. Decision			Plmt. Pred. Model			Monitored Metrics				GPU Prog. Method		
Features	Data shipping	Function shipping	Semantic-aware	Frequency-based	Greedy	No caching	Job	Application	Pipeline	Task	Function	Segment	Bit	Runtime	Compilation time	Hybrid	Automatic	Semi-automatic	Manual	Cost model	Heuristic model	Learn model	Latency	Throughput	Energy consumption	Monetary price	Kernel template	Compiler	Raw kernel
#papers	68	9	5	9	13	50	3	9	7	49	2	5	2	26	35	20	25	42	19	11	46	26	75	21	7	1	30	14	41
Karnagel et al. [28]	**x**					x				x						x		x				x	x				x		x
Robust CoGaDB [26]		**x**		x						x				x				x				x	x				x		x
HetCache [13]		x	**x**						x					x			x				x		x	x				x	
RateupDB [24]	x			**x**				x								x	x				x		x	x					x
Parla [12]	x				**x**					x				x	x		x		x		x		x				x	x	
Wen and O'Boyle [40]	x					**x**				x						x		x				x	x					x	x
Ravi et al. [31]	x					x	**x**								x			x			x		x	x			x		
Compressed Crystal [5]	x					x		**x**							x				x	x	x		x						x
HetExchange [21]	x					x			**x**							x	x				x		x	x				x	
Carvalho et al. [18]	x					x				**x**					x				x		x		x				x		
HERO [29]		x		x							**x**					x		x				x	x						x
Mordred [8]		x	x									**x**				x		x		x	x		x				x		
Pirk [39]	x					x							**x**		x			x			x		x						x
Li et al. [33]	x					x				x				**x**				x				x	x				x		
MCL Schedulers [2]		x			x					x				x	**x**		x		x		x		x					x	x
CGgraph [9]		x	x							x						**x**	x				x		x						x
DBD [30]	x					x				x						x	**x**				x		x						x
CoTrain [34]	x				x					x						x		**x**		x	x		x	x			x		
TensorFlow [16]	x				x					x					x		x		**x**		x		x	x			x		
Gowanlock et al. [17]	x					x				x				x				x		**x**			x				x		x
GaccO [1]		x	x					x								x		x			**x**			x					x
READYS [20]	x					x				x				x				x				**x**	x				x		x
Placeto [23]	x					x			x						x			x				x	**x**				x		
Lutz et al. [11]	x				x					x						x	x				x		x	**x**					x
Sigmoid [7]	x					x				x				x			x			x			x		**x**				x
Crystal [4]	x					x		x							x				x	x	x		x			**x**			x
Lee and Park [36]	x					x				x					x				x		x		x				**x**		
Xekalaki et al. [19]	x					x				x					x		x				x		x					**x**	
GFlink [10]	x			x						x					x				x				x						**x**

Figure 1: Abridged taxonomy for workload placement on CPU-GPU systems (representative papers are marked with x)

Placement decision. Automatic approaches [e.g., 7, 13, 21, 24, 30] usually rely on cost models or heuristics. Semi-automatic approaches [e.g., 8, 23, 34] rely on human interaction for tuning, but automate the placement decision. Manual placement [e.g., 10, 16, 18, 36] relies on the developer's guidance based on extensive profiling.

3.1.2 *Estimating costs for placement decisions.*

Several metrics have been proposed to estimate the costs of placement decisions.

Placement prediction model. *Cost-based placement* [e.g., 4, 7, 8, 17] relies on performance cost estimates (e.g., latency, throughput, power, energy consumption). This strategy builds a quantitative performance formulation considering factors such as CPU-GPU architectural differences, workload characteristics, and CPU-GPU data transfers. Oftentimes, cost-based placement results in ad-hoc designs tailored for specific settings, and hence non-transferable to different architectures and configurations. *Heuristic-based* approaches [e.g., 1, 13, 30, 39] are based on pre-defined rules. A common heuristic is to offload workloads with a low (high) degree of thread parallelism to CPUs (GPUs). *Learn-based* approaches [e.g., 14, 20, 23, 26] perform placement based on cost prediction at runtime, using historical execution logs for training. Such techniques exploit transfer learning to generalize placement decisions to other settings. The usual drawbacks include expensive training (e.g., training data size, training time, energy footprint) and extra care to avoid catastrophic forgetting. Most approaches though are heuristic-based (see Figure 1).

Monitored metrics. Various metrics have been proposed to influence CPU-GPU placement. Latency (workload execution time) is the most popular [e.g., 23, 26, 30, 40]. Other metrics include throughput [e.g., 1, 11, 16, 34], energy consumption [7], and monetary price [4]. Comparing placement strategies opting for optimizing such different metrics is an additional design challenge.

3.1.3 *Managing heterogeneous CPU-GPU code.*

Specialized programming abstractions and coding paradigms are needed to enable routinely placing workload on mixed CPU and GPU environments.

GPU programming method. Some approaches use *kernel templates* [e.g., 8, 17, 18, 36] that offer pre-defined GPU-accelerated algorithms via public libraries. These offer a high degree of GPU programming abstraction, but they lack capabilities for fine-tuned optimization. Typical options include NVIDIA CUDA-X, CuPy, RAPIDS (e.g., cuML, cuDF, cuGraph), Intel HDK, Crystal [4], etc. Other approaches employ *compilers* [e.g., 13, 19, 21, 40] that enable the implementation of user-defined function kernels using high-level sequential code. Since the kernel is customizable, it allows some flexibility for performance optimization. Popular compilers include Numba, CUDA Python, PyCUDA, PyOpenCL, NVPTX, AMD Aparapi, DaCe [27], etc. Finally, another option is to use *raw kernels* [e.g., 9–11, 30] that constitute the most fine-grained method to program GPUs and hence, to achieve high performance optimization on GPUs. Efficient programming with raw kernels requires insights about the GPU hardware internals. Typical options include CUDA (NVIDIA), HIP (AMD), SYCL (Intel), and the general purpose OpenCL and OpenACC. Choosing the right method is a fine balance between productivity (programming abstraction) and performance (fine-grained GPU optimization) [3].

The tutorial will cover these technologies with code examples and provide a comparison (w.r.t. effort, usability, expressiveness) with appropriate references to the relevant papers.

4 OPEN ISSUES AND RESEARCH DIRECTIONS

Open issues. Pain points include: (1) high complexity to predict performance metrics, (2) non-linear, conflicting correlation of factors, (3) load imbalance due to a plethora of relevant factors, (4) expensive training for learn-based models, (5) expensive and energy-intensive infrastructure, (6) lack of automation in optimization and fine-tuning strategies, and (7) different software and hardware offerings requiring specialized treatment.

Research Directions. There are several research directions worth pursuing: (1) automated placement strategies to optimize the trade-off between processor choice and data transfer, (2) multi-objective placement strategies to account for contradicting cost metrics (e.g., latency and energy), (3) multi-dimensional placement strategies (most current works consider only a few of the dimensions/features listed in Figure 1), (4) automated fine-tuning of execution parameters, (5) adaptive cache-aware workload placement using learning, (6) suitable benchmarks (data, workloads) and simulators to provide a standard means for comparing different strategies.

5 TARGET AUDIENCE, LEARNING OUTPUT

Material. The tutorial will be example-driven showcasing the strengths and limitations of the state of the art. The tutorial material will be publicly available.

Audience. The tutorial targets students, academics, researchers, and practitioners interested in developing efficient data analysis workflows taking advantage of heterogeneous CPU-GPU systems. No prior knowledge is needed on GPU programming, but we assume understanding of basic concepts about parallel processing.

Output. The learning output includes: (a) An overview of the state-of-the-art practices and techniques for efficient workload placement on heterogeneous CPU-GPU architectures. (b) Identification of the most critical factors related to the performance implications of workload placement. (c) Understanding the technical limitations and the trade-offs between design choices and achieved goals. And (d) exposure to challenges and opportunities for the new generation of heterogeneous CPU-GPU systems.

6 PRESENTERS

Marcos Carvalho [url] is currently a Marie Skłodowska-Curie early stage researcher, pursuing a joint PhD degree at UPC, NKUA, and Athena RC. His research is related to data processing optimization, heterogeneous hardware, and machine learning.

Alkis Simitsis [url] is a Research Director at Athena RC. He has 20+ years of corporate and academic experience on solutions for scalable big data infrastructure, data analytics, distributed databases, and systems optimization. He holds 45 patents, has published 120+ papers, and frequently serves in the PC's of top-tier conferences.

Anna Queralt [url] is a Serra Húnter fellow at UPC and Established Researcher at BSC, with 10+ years experience in efficient and scalable data management and processing. She has published 30+ papers in top venues and serves as PC in top-tier conferences.

Oscar Romero [url] is a Full Professor at UPC. His research interests span data management and governance. He has published 100+ papers on top venues and serves as PC in top-tier conferences.

ACKNOWLEDGMENTS

This work has been partially supported by the H2020-MSCAITN-2020 DEDS (GA.955895), and the EU-HORIZON programmes CREXDATA (GA.101092749) and FAIR-CORE4EOSC (GA.101057264) and the Spanish MICIU DOGO4ML (PID2020-117191RB-I00).

REFERENCES

[1] N. Boeschen et al. 2022. Gacco-a gpu-accelerated oltp dbms. In *ACM SIGMOD.*
[2] A. Kamatar et al. 2020. Locality-aware scheduling for scalable heterogeneous environments. In *IEEE/ACM ROSS.*
[3] A. K. Ziogas et al. 2021. NPBench: A benchmarking suite for high-performance NumPy. In *ACM ICS.*
[4] A. Shanbhag et al. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *ACM SIGMOD.*
[5] A. Shanbhag et al. 2022. Tile-based lightweight integer compression in GPU. In *ACM SIGMOD.*
[6] B. He et al. 2009. Relational query coprocessing on graphics processors. *ACM TODS* 34, 4 (2009).
[7] B. Pérez et al. 2021. Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *Elsevier JPDC* 157 (2021).
[8] B. Yogatama et al. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *PVLDB* 15, 11 (2022).
[9] Cui et al. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *PVLDB* (2024).
[10] C. Chen et al. 2018. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data. *IEEE TPDS* 29, 6 (2018).
[11] C. Lutz et al. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *ACM SIGMOD.*
[12] H. Lee et al. 2022. Parla: A python orchestration system for heterogeneous architectures. In *IEEE SC.*
[13] H. Nicholson et al. 2023. HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics. *CIDR* (2023).
[14] H. Zhang et al. 2020. Learning-driven interference-aware workload parallelization for streaming applications in heterogeneous cluster. *IEEE TPDS* 32, 1 (2020).
[15] J. Ren et al. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *USENIX ATC.*
[16] M. Abadi et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv* (2016).
[17] M. Gowanlock et al. 2019. Accelerating the unacceleratable: Hybrid CPU/GPU algorithms for memory-bound database primitives. In *ACM DaMoN.*
[18] M. NL Carvalho et al. 2024. Performance Analysis of Distributed GPU-Accelerated Task-Based Workflows. *EDBT* (2024).
[19] M. Xekalaki et al. 2022. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware. *PVLDB* 15, 13 (2022).
[20] N. Grinsztajn et al. 2021. Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. In *IEEE CLUSTER.*
[21] P. Chrysogelos et al. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB* 12, 5 (2019).
[22] R. Appuswamy et al. 2017. The case for heterogeneous HTAP. In *CIDR.*
[23] R. Addanki et al. 2018. Placeto: Efficient progressive device placement optimization. In *NeurIPS.*
[24] R. Lee et al. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *PVLDB* 14, 12 (2021).
[25] S. Breß et al. 2014. Ocelot/hype: Optimized data processing on heterogeneous hardware. *PVLDB* 7, 13 (2014).
[26] S. Breß et al. 2016. Robust query processing in co-processor-accelerated databases. In *ACM SIGMOD.*
[27] T. Ben-Nun et al. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *IEEE SC.*
[28] T. Karnagel et al. 2015. Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments.. In *EDBT Workshops.*
[29] T. Karnagel et al. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB* 10, 7 (2017).
[30] T. Park et al. 2023. Orchestrating Large-Scale SpGEMMs using Dynamic Block Distribution and Data Transfer Minimization on Heterogeneous Systems. In *ICDE.*
[31] V. Ravi et al. 2012. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *IEEE/ACM CCGRID.*
[32] V. Rosenfeld et al. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM CSUR* 55, 1 (2022).
[33] Z. Li et al. 2021. Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time. *Elsevier J. Syst. Arch.* 114 (2021).
[34] Z. Li et al. 2023. CoTrain: Efficient Scheduling for Large-Model Training upon GPU and CPU in Parallel. In *ACM ICPP.*
[35] D. Grewe and MFP O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *Springer CC.*
[36] S. Lee and S. Park. 2021. Performance analysis of big data ETL process over CPU-GPU heterogeneous architectures. In *IEEE ICDE Workshops.*
[37] S. Mittal and J. S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM CSUR* 47, 4 (2015).
[38] NVIDIA. 2024. *NVIDIA H200.* https://www.nvidia.com/en-eu/data-center/h200/
[39] H. Pirk. 2012. Efficient cross-device query processing. In *VLDB PhD Workshop* .
[40] Y. Wen and M. FP O'Boyle. 2017. Merge or separate? Multi-job scheduling for OpenCL kernels on CPU/GPU platforms. In *ACM GPGPU.*