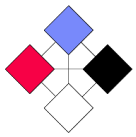
D7.1 Design of integration HW accelerators



DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.5

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

In D7.1, the DAPHNE project team reports on the planned overall design of integration HW accelerators as well as details on accelerated operations and primitives, as well as its compiler and runtime support.

D7.1 Design of integration HW accelerators			
WP7 – HW Accelerator Integration			
Type of document	Report	Version	[1.5]
Dissemination level	PU		
Lead partner	Technische Universität Dresden		
Author(s)	Dirk Habich		
Contributors	KNOW		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial write-up of the report.	Dirk Habich
V1.1	Incorporated comments by Patrick Damme and Mark Dokter; additional details, figures and explanations.	Dirk Habich
V1.2	Incorporated comments by Patrick Damme; added additional details and explanations in Sections 4 and 5.	Dirk Habich
V1.3	Revised version by incorporating Mark Dokter's comments.	Dirk Habich
V1.4	Incorporated comments by Patrick Damme; enhanced details and explanations in Section 4 and 5.	Dirk Habich
V1.5	Revised version by incorporating Mark Dokter's comments.	Dirk Habich

Abbreviations

Abbreviation	Term
KNOW	Know-Center GmbH
WP	Workpackage
T	Task

1 Introduction

Modern data-driven applications have to deal with increasingly large and heterogeneous data collections as well as a variety of machine learning (ML) models for cost-effective automation and improved analysis results [A17, A+19b, P+21, Z+19]. This creates a trend towards integrated data analysis (IDA) pipelines that jointly utilize data management (DM), high-performance computing (HPC), and ML systems. As described in [D+22], developing and deploying such IDA pipelines is, however, still a painful process of integrating different systems and related developers, programming paradigms, resource managers, and data representations. Integrating DM+ML, HPC+ML, DM+HPC for improving productivity and/or performance are old problems though [BKS20, S+00, S+11]. However, an open system infrastructure for seamlessly developing, deploying and running IDA pipelines is still missing, and at the same time, new challenges related to hardware, productivity, and utilization emerge.

To overcome that, the DAPHNE project sets out to build an open and extensible system infrastructure for integrated data analysis pipelines. To achieve that goal, our envisioned infrastructure is based on MLIR [L+20] as a multi-level, LLVM-based intermediate representation backed by multiple organizations and communities. This approach allows a seamless integration with existing applications and runtime libraries while also enabling extensibility for specialized data types, hardware-accelerated kernels, hardware-specific compilation chains, and custom scheduling algorithms. While the DAPHNE report *D2.1 - Initial System Architecture* has described the overall DAPHNE system architecture, this report focuses on presenting the motivation and design for hardware accelerator integration.

Interestingly, DM, HPC, and ML systems share many optimization techniques and together stress every hardware aspect of storage, computation, and networking. Accordingly, these systems are strongly impacted by hardware challenges such as the end of Dennard scaling and the end of Moore's law, which ultimately lead to dark silicon [BC11, JP13] and increasing specialization at device level (CPUs, GPUs, FPGAs, APUs) and storage level (computational memory/storage, storage hierarchies). This hardware specialization, in turn, leads to increasing heterogeneity and thus, even larger productivity and utilization challenges for pipelines across DM, HPC, and ML systems [Ca+18, D+22, P+21b]. In particular, the challenges are (i) developing as well as generating operators - hereinafter also called computation kernels or kernels for short - which can be efficiently executed on accelerators such as CPUs, GPUs or FPGAs, (ii) integrating these accelerator-specific operators in the whole DAPHNE compilation and runtime infrastructure in a seamless way, and (iii) selecting the best-fitting accelerator for efficient execution depending on the specific IDA pipeline and hardware environment.

The remainder of this report is structured as follows:

- In Section 2, we give a short overview about hardware accelerator heterogeneity.
- Based on this overview, we detail the overall design approach of DAPHNE for hardware accelerator integration in Section 3.
- Then, Section 4 describes our approach for handcrafting as well as generating of hardware-accelerated kernels.

- Section 5 presents our concepts for an efficient and effective employment of the hardware-accelerated kernels in IDA pipelines.

2 Overview of Hardware-Accelerator Landscape

Recent development on the hardware landscape has a massive impact on the software stack for data processing software systems [BC11, S05]. While the software side enjoyed a 'free ride' with ever increasing clock cycles for many decades, the time has finally come to adjust principles of software architectures to exploit the opportunities provided by modern hardware components [S05]. To better understand this aspect, we give a short overview of the hardware development in recent years. As described below, the heterogeneity of hardware at device and storage levels is growing and thus poses a great challenge.

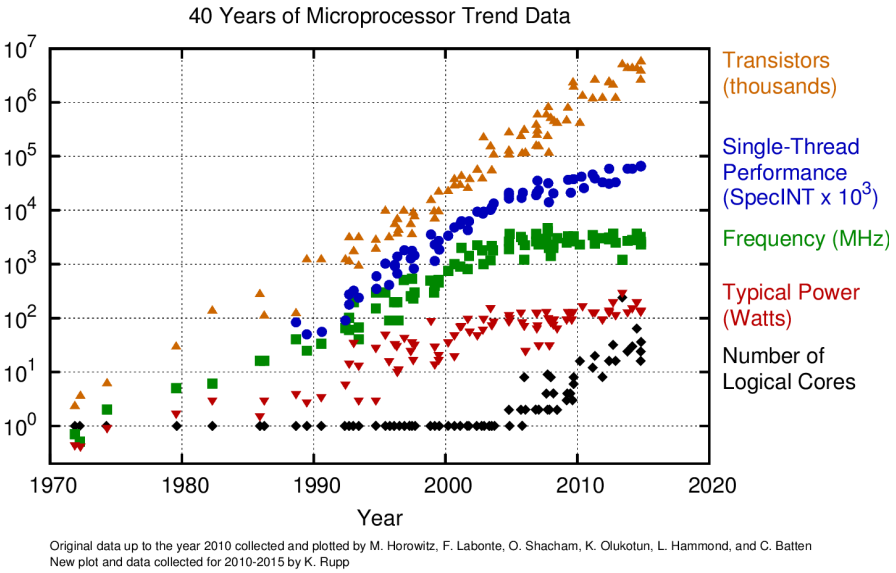


Figure 1: Trend of the main processor characteristics.

2.1 Device Level Heterogeneity

At device or processing level, the early multi-core with double-digit numbers of cores per system has passed [BC11, S05]. Nowadays, multi-socket CPU systems with up to 1,000 cores have become economically feasible. In addition, GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays) have made significant progress in providing general-purpose processing capabilities.

2.1.1 CPU Developments

The first wave of a disruption in the area of processing units was the switch from increasing to maintaining the frequency in combination with increasing the number of cores of a single CPU [BC11, S05]. Current general-purpose server processors like for example the Intel Xeon Gold Processor 6240R provides 24 cores resulting in a direct support of 48 threads using hyper-threading for a single socket. As Figure 1 shows, this trend may continue by keeping the frequency at the current level, but still increasing the number of cores on the same die using the growing number of transistors. Moreover, this allows to build scale-up hardware systems with more than 1,000 cores by combining multiple CPUs on one die, e.g., HPE SGI UV 300 including up to 48 TByte of DRAM. These scale-up hardware systems are also called shared-

memory multiprocessor systems and can be further classified in systems with uniform memory access (UMA) and systems with non-uniform memory access (NUMA) [BC11, W04]. In both cases, all processes can access the complete memory, but they differ in the way they realize the connection of the processors to the memory. The issues in NUMA are increased latency and decreased bandwidth when accessing data in remote main memory locations [K+13, P+10].

Besides the increasing number of cores according to the *Multiple-Instruction Multiple-Data (MIMD) parallel paradigm*, there is also a growing potential of support for data-level parallelism as second kind of distribution. SIMD extensions according to the *Single-Instruction Multiple-Data parallel paradigm (SIMD)* allow to process multiple data items with a single instruction, which looks extremely promising for data-intensive applications [H15]. While almost all modern CPU architectures have been providing SIMD extensions for many CPU generations known as SSE or AVX on the Intel and AMD cores as well as NEON or SVE for ARM cores, a significant step forward has been made in the recent past. On the one hand, larger registers can now be found on most modern CPUs. Having initially started with 128 bit-wide SIMD registers, the current Intel AVX-512 extensions provide 512 bit-wide operations. In contrast, ARM SVE supports up to 2,048 bit-wide SIMD registers [S+17]. On the other hand, the instruction set was extended to better support blending, comparing, and permuting operations in addition to conflict detection and support for floating-point arithmetic. While SIMD extensions are heavily used in core database algorithms like scans [W+09] or compression algorithms [D+20] in column-stores, the general design of efficient data structures fully leveraging the potential of SIMD extensions is still a challenging research task.

In addition to SIMD extensions, recent developments also suggest extending the instruction set of traditional CPU cores, e.g., with a database-specific instruction set and corresponding additional elements like additional registers and larger memory interfaces. For example, a *dpCore* as part of a specialized Data Processing Unit (DPU) originally proposed within the Oracle Rapid project [A+17] offers single-cycle instructions like bit-vector load (BVLD), filter (FILT), and CRC32 hash-code generation to accelerate query operations like filters and joins. A similar approach is pursued by the Titan3D project [H+17] extending a general-purpose Xtensa LX5 core with instructions to support efficient bitwise decompression, hash code generation or efficient operators for bitmap index utilization.

Thus, these CPU extensions already provide accelerator functionality within general-purpose CPUs, and they will get more diverse in the future. For example, the number of SIMD instructions differs between Intel architectures, because not all available SIMD extensions from Intel are meant to be supported by all architectures.

2.1.2 Domain-Specific Processing Devices

With CPUs as general-purpose processing devices on the one side, domain-specific processing units may be considered the extreme on the other side of the diversity spectrum with GPGPUs (general-purpose computing on graphics processing units) probably serving as the most widely known representative of domain-specific processing units. GPUs are originally designed to process graphic pipelines including coordinate transformations and rasterization as well as hosting additional components, e.g., a display controller. While most of the elements were

hard-wired and thus only a limited part of this functionality was programmable, the growing interest in GPUs has caused significant changes in the GPU-internal architectures. The graphics pipeline has become fully programmable, controlled by a user-defined sequence of so-called shaders allowing the GPU cores to be used for general-purpose computations. For example, GPUs are now commonly used for HPC applications.

Acceleration Processing Units (APUs): GPUs or CPUs with specific instruction set extensions are representatives of attempts to support domain-specific requirements. Pushing the envelope by supporting specific application requirements exclusively leads into the field of domain-specific acceleration processing units (APUs) usually deployed within a coprocessor model. While there exists -- by design -- a wide variety of different systems, APUs for efficiently executing vector operations have a long tradition and are gaining -- with machine learning as the driving application area -- significant attention. For example, NEC provides a card-mounted vector processor SX-ACE [NEC] with a C++-based middleware to accelerate sparse matrix computations. Google introduced a Tensor Processing Unit (TPU), which aims at accelerating neural network computations [J+17]. Since the main task of a TPU is to multiply matrices, more than half of the chip is covered by the matrix multiplication unit including a 24MB scratchpad buffer. The remaining area is only used for the control, several interfaces, and other auxiliary components. The loss of general-purpose flexibility and the focus on specialized application tasks results -- as expected -- in a significant performance gain of up to 30X compared to a corresponding implementation based on recent CPUs or GPUs [J+17].

While APUs were developed to support compute-intensive tasks, APUs also play a crucial role to support data-intensive tasks, which are directly relevant for efficient data systems. An example can be seen in the HARP ASIC to be found in systems of the HPE SGI UltraViolet family. The HARP APU is responsible for connecting individual rack units (IRUs) to a cache-coherent NUMA system by maintaining cache coherency and providing a common address space. A HARP also hosts a Global Reference Unit (GRU) offering an API to offload memory operations within a NUMA-based system, e.g., functionality to asynchronously copy memory between processors and to accelerate atomic memory operations. As shown in [D+17] in greater detail, significant performance improvements can be achieved by using the specialized implementation of low-level memory management functions compared to variants offered by the operating system.

Reconfigurable Hardware: APUs are usually developed to enhance the performance and to reduce power consumption for a specific task. However, they are hard-wired in a sense that once they are manufactured, they cannot be changed anymore. For situations with a higher degree of flexibility (e.g., changing application requirements) or for rapid-prototyping, reconfigurable hardware, especially FPGAs, lend themselves to providing a viable alternative. The core concept of FPGAs is based on an array of configurable logic blocks each implementing a binary function, which can be changed by modifying the truth table of the particular function [T17]. This table is written to the SRAM cells of the logic block and can be overwritten, thus yielding a reconfigurable processing unit. The number of these logic blocks as well as other specifications and the price vary heavily between different FPGAs models. The ability to be reconfigurable obviously comes with some limitations compared to ASICs. First, the resources on an FPGA are limited, i.e., the specific task to be supported by an FPGA-based

implementation has to be carefully selected; second, building takes some time. For example, the synthesis of an application for an FPGA can take between several minutes and multiple hours, depending on the complexity of the application and the available computing power.

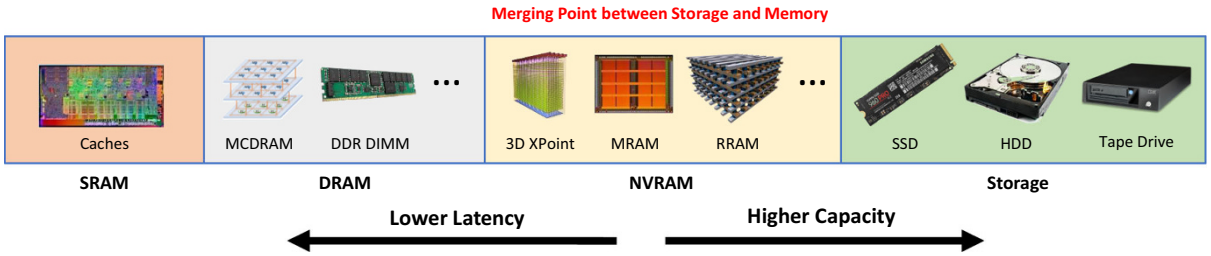


Figure 2: Illustration of the diversity of memory and storage technologies (taken from [OL18]).

2.2 Storage Level Heterogeneity

In addition to the increasing device level heterogeneity, we also have seen tremendous developments in memory and storage technologies as illustrated in Figure 2. For a long time, the traditional storage hierarchy has comprised several layers of memory technologies, ordered from the fastest and least dense to the slowest and most dense: CPU caches (SRAM), main memory (DRAM), secondary memory (HDD), and potentially tertiary memory (Tape Drive) [OL18]. The rise of Flash memory, manufactured in the Solid-State Drive (SSD) form, has pushed HDDs another level down the storage hierarchy: SSDs have successfully superseded HDDs. However, the rise of novel memory technologies, such as Storage-Class Memories (SCM), and substantial hardware and software advances in existing technologies, such as Open-Channel SSDs [B18], force us to reconsider how we conceive storage hierarchies as described in Deliverable D6.1 of the DAPHNE project. Indeed, storage system designers are faced with an unprecedented diversity of memory and storage technologies, as illustrated in Figure 2.

2.3 Challenges

The emerging hardware, in turn, leads to increasing heterogeneity and thus, even larger productivity and utilization challenges for IDA pipelines across DM, HPC, and ML systems. This applies to CPU instruction extensions such as SIMD as well as to the multitude of domain-specific accelerators such as GPUs or FPGAs. Thus, major challenges tackled by WP7 of the DAPHNE project are: (i) developing as well as generating operators - also called computation kernels -, which can be efficiently executed on accelerators such CPUs, GPUs, or FPGAs, (ii) integrating these accelerator-specific operations in the whole DAPHNE compilation and runtime infrastructure in a seamless way, and (iii) selecting the best-fitting accelerator for efficient execution depending on the specific IDA pipeline and hardware environment.

3 DAPHNE System Architecture

In cooperation with the other WPs of DAPHNE, we designed a system architecture to efficiently address our challenges for HW accelerator integration. The resulting DAPHNE system architecture is shown in Figure 3 and explained in more detail in Deliverable D2.1 of the DAPHNE project. The DAPHNE prototype is built from scratch in C++ but utilizes MLIR [L+20] as a multi-level, LLVM-based intermediate representation (IR) as well as existing runtime libraries such as BLAS, LAPACK, and DNN kernels and collective operations. These libraries are augmented with more specialized, custom kernel implementations. Users specify their IDA

pipelines in the DaphneDSL (a language similar to Julia, PyTorch, or R) or DaphneLib (a high-level Python API with lazy evaluation that internally compiles DaphneDSL scripts as well). These scripts are then compiled – via a multi-level compilation chain – into executable runtime plans.

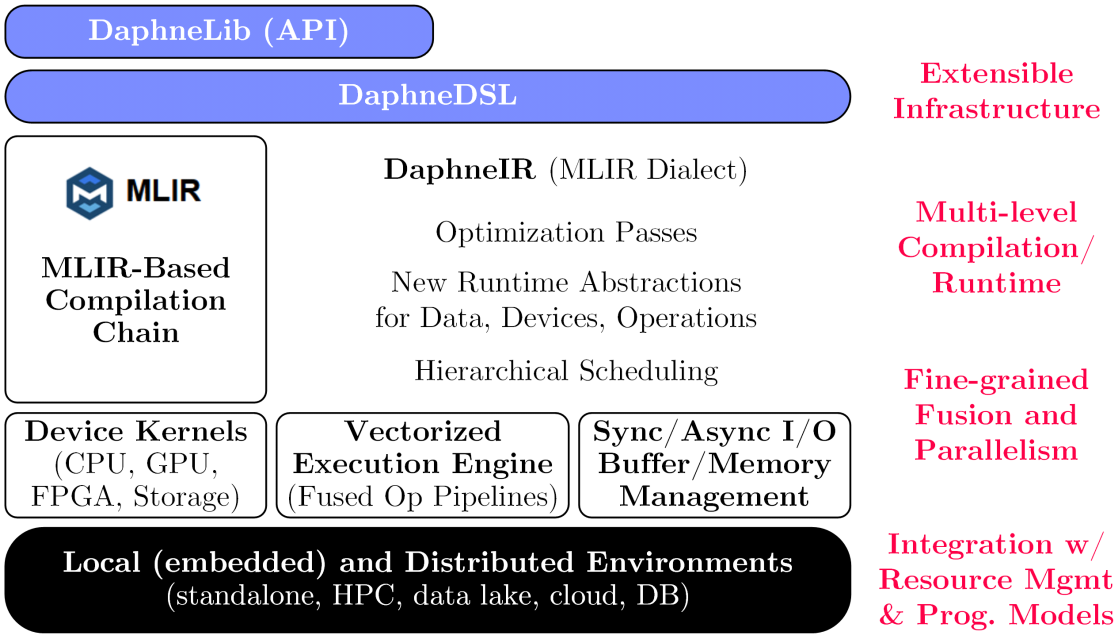


Figure 3: DAPHNE System Architecture.

A major design decision with respect to HW acceleration integration is the focus on an extensible infrastructure allowing to register new data types, kernels, and scheduling algorithms in predefined extension hooks. Thus, extensibility goes beyond recent work on combining variants (variability) of communication primitives [G+21]. Based on this extensibility concept, we can add a great variety of kernels for HW accelerators in a flexible way without focusing on a specific hardware accelerator. Moreover, we explicitly decided to implement these kernels in C/C++ (as opposed to lowering them to low-level MLIR/LLVM operations), because latest generation HW accelerators can usually be programmed that way, while we would have to wait for the respective MLIR/LLVM extensions for the lowering approach. More details on how we implement kernels for HW accelerators can be found in Section 4.

Based on that foundation, we lower DaphneDSL's frame and matrix operations to calls to precompiled C++ hardware-accelerated kernels and only use LLVM for control flow and scalar operations. This is very different to other MLIR dialects but allows a more flexible HW acceleration integration. Moreover, this design still allows the implementation of kernels in LLVM, or other MLIR dialects as well as the utilization of code generation approaches, if beneficial. The mapping to the C++ kernel calls is realized via corresponding MLIR compiler passes. This can be used to implement different strategies - different compiler passes - to select the best-fitting HW accelerator for efficient IDA execution depending on the specific pipeline properties and hardware environment (in cooperation with WP5 and WP6). More details on how we (plan to) select the best-fitting kernel can be found in Section 5.

Coordinating the system architecture with WP2-6 makes it easier to collaborate within the consortium and allows all partners to focus on their respective WPs. Thus, the result is a

comprehensive system infrastructure with many facets that are coordinated with each other. To demonstrate the advantages of this system architecture, we are jointly working on a vectorized execution engine for compiled operator pipelines of frames and matrices; heterogeneous HW devices, and computational storage as described in Deliverables D2.1 and D6.1. This engine is also the essential driver for WP7 and provides the corresponding functionality in terms of orchestrating hardware-accelerated kernels. The set of kernels includes key operations of linear algebra (e.g., matrix multiplication [BA+20]), data management (e.g., joins and group-by), HPC libraries (e.g., signal processing) as well as data access primitives such as scans, sorting, as well as lossless and lossy (de)compression [D+20, E+19].

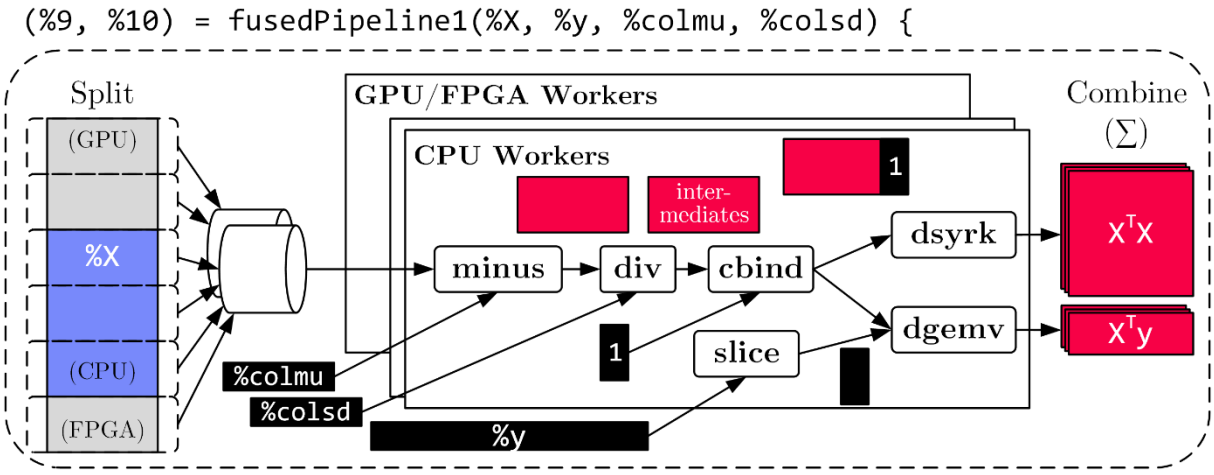


Figure 4: Vectorized Execution of Compiled Operator Pipelines (with multi-device data and operator placement).

To better understand the vectorized execution, Figure 4 depicts the basic integration of vectorized operator pipelines into an execution plan. Similar to LLVM loops, such a vectorized pipeline has multiple inputs, multiple outputs, and an IR body (operation). Additionally, we specify split (e.g., row slicing) and combine (e.g., row-bind concatenation or aggregate) functions. In this example, we perform matrix standardization (`(X-colMeans(X))/colSds(X)`), append a column of ones for intercept computation, and compute X^TX and X^Ty as part of a close-form linear regression algorithm. The input matrix X is federated across CPU, GPU, and FPGA memory, and vectorized execution creates tasks for aligned row partitions (similar to morsels [L+14]) and appends them to one or multiple (device-specific) task queues. A task comprises its input data, an operator pipeline (graph) with a specific input data binding (scalar, row, or tile), outputs, and a combine. Then, worker threads read from the queues, execute the tasks by calling the precompiled hardware-accelerated C++ kernels, and combine the results.

4 Kernels for Heterogeneous Hardware Accelerators

As described above, it is an explicit design decision to implement kernels in C/C++ with respect to the HW accelerator integration. In this section, we give an overview of the state-of-the-art and describe our approach in more detail.

4.1 State-of-the-Art

With the new opportunities that open up the emerging hardware landscape as described in Section 2, there also arises a plethora of new challenges when considering how to program kernels on these hardware platforms. To solve these new challenges, state-of-the-art

programming approaches are either based on general-purpose or on domain-specific programming frameworks.

4.1.1 General-Purpose Programming Frameworks

One approach for implementing kernels on different HW accelerators are general heterogeneous programming frameworks, such as OpenMP [DM98], OpenCL [M+11], CUDA [SK10], SYCL [RL16], and oneAPI [R+21]. These models are not specialized for a single domain or class of applications, but rather define abstraction mechanisms and language extensions for common computational patterns, such as parallel loops or concurrently executing tasks. Generally, they are tightly integrated with an existing serial programming language, such as C, C++, or Fortran and thus allow at least partial re-use of an existing serial code base.

The advantage of the generic nature of these programming models is that they can be used to accelerate a wide range of applications from very different domains, as they focus on fundamental computational patterns rather than domain-specific abstractions. Improvements of the implementation of a programming model, e.g., in the compiler or runtime library, benefit many applications, and hardware vendors only need to implement the abstractions of the programming model once to open up their hardware to many users and applications.

On the other hand, due to their general design, such programming frameworks fail to capture the domain-specific, high-level semantics of HW accelerators, and still operate on a comparably low level of abstraction, e.g., reasoning about individual loops. Memory management is typically also explicit and requires careful attention by the programmer. Because the language mechanisms (e.g., pragmas) are directly integrated into program code, applications implemented in such general frameworks also mix what is computed with how it is computed, with negative effects on code readability and maintainability.

As the abstractions of general programming frameworks still operate on a comparably low level, implementations using such a framework are often specific to a single class of devices. One example for this problem can be found when comparing vendor recommendations for programming FPGAs or GPUs using OpenCL: While FPGA vendors usually suggest a single work-item implementation, with FPGA-specific optimizations such as pipelining applied inside that single work-item, GPU vendors usually suggest parallelizing an application across many work-items and, if applicable, also concurrently active kernels. Consequently, OpenCL implementations optimized for FPGAs usually do not deliver good performance on GPUs and vice versa.

4.1.2 Domain-Specific Programming Frameworks

In contrast, domain-specific programming frameworks can be used to overcome the limitations of general-purpose programming models. They are specialized for a single domain and provide programming language abstractions for operations specific to this domain. Because these domain-specific operations have well-defined semantics and the memory access pattern is known beforehand, it is possible to separate what is computed from how it is computed, by moving the mapping of domain-specific operations to hardware accelerator facilities into the implementation of the programming framework. This also allows to vary the mapping depending on available hardware and the desired performance independently of the actual source code.

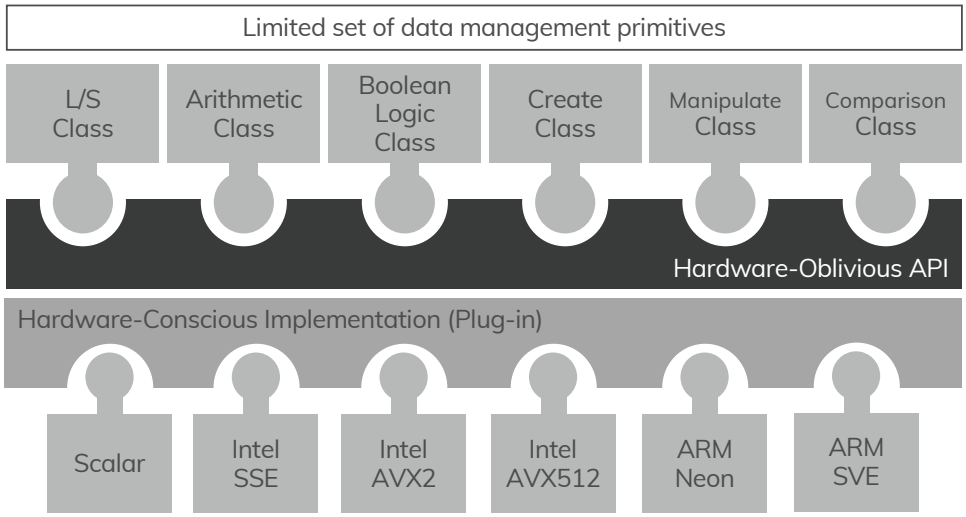


Figure 5: Template Vector Library.

A recent example from the data management domain is the Template Vector Library (TVL) [U+20]. TVL offers hardware-oblivious, but column-store specific primitives as generic functions. This explicitly enables database systems programmers to implement each database operator in a SIMDified, but hardware-independent fashion, on the one hand. On the other hand, the TVL is also responsible for mapping the provided hardware-oblivious primitives to different SIMD hardware as illustrated in Figure 5. For this mapping, the TVL includes a plug-in concept, where each plug-in has to implement each provided hardware-oblivious primitive for a specific SIMD hardware in a hardware-conscious manner. TVL is realized in C++ with the help of template metaprogramming.

Other similar approaches are Weld [P+18], Voodoo [P+16], or Sierra [Le+14]. The downside of domain-specific frameworks is the additional implementation effort that initially has to be spent to define and implement the framework itself. For general-purpose frameworks, the implementation effort can be amortized over many different applications and domains that can use the framework. For domain-specific frameworks, on the other hand, this effort can be justified only by a large number of users or applications, or significant gains in expressiveness or performance. To reduce the initial implementation effort, compiler frameworks such as LLVM or MLIR can facilitate the implementation of domain-specific frameworks.

4.2 Hand-Crafted HW-Accelerated Kernels

To show the broad applicability of our HW accelerator integration concept, we will implement various kernels for our vectorized execution engine using different frameworks (general-purpose as well as domain-specific). In particular, we will use CUDA [SK10] for NVIDIA GPUs, oneAPI [R+21] and T2S [S+19] for FPGAs and TVL [U+20] for SIMD extensions on CPUs. Through these extensive implementations, we can finally compare the individual approaches with each other in terms of development time and performance of the implemented kernels. This aspect is especially important regarding FPGAs.

In general, these hardware-accelerated kernels are initially device-specific, whereby the vectorized execution engine, in combination with the scheduling mechanisms provided by WP5, is responsible for the orchestration regarding an efficient and effective IDA pipeline

execution. If required to further optimize the execution of specific operations over multiple hardware accelerators, it will also possible to implement a set of hand-crafted HW-accelerated kernels for various combinations of multiple devices. Despite some existing libraries, multi-device support for important operations of integrated data analysis pipelines is still very limited. Examples of existing libraries are cuBLAS-XT [W+16] and some academic prototypes for matrix multiplication [C+20] or hash-joins [L+20b]. To overcome that and to achieve the full potential of multiple device execution, we aim to adopt strategies from distributed query processing and distributed linear algebra to the multi-device setting and adapt these operations according to specific characteristics of the underlying devices. Furthermore, we will also adopt distribution primitives from WP4 such as ring-reduce to provide the tools for composing efficient data analysis pipelines on multiple, potentially heterogeneous, accelerators. Moreover, the DAPHNE compiler will be extended accordingly, so that the compiler is able to support single-device (default) as well as multi-device hardware-accelerated kernels.

These hand-crafted hardware-accelerated kernels are pre-compiled as reusable building blocks so that they can be called from a variety of DaphneDSL-scripts. This significantly increases the usefulness and applicability of these hand-crafted kernels.

4.3 Code Generation

Besides the implementation of hand-crafted hardware-accelerated kernels, we also dedicate ourselves to the following described code generation aspects to improve our approach.

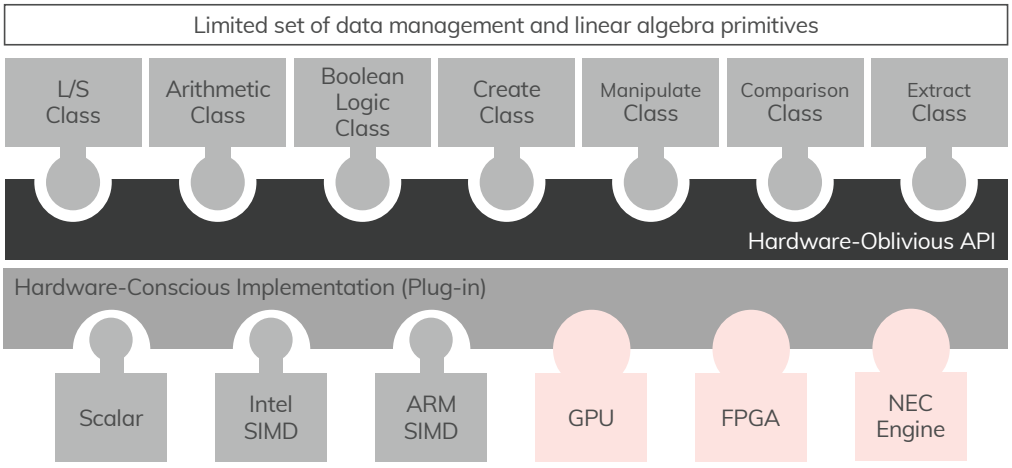


Figure 6: Enhanced Template Vector Library (Virtual Vector Concept).

4.3.1 Virtual Vector Concept

Supporting multiple hardware accelerators with hand-crafted kernels implies a high implementation effort. To overcome that, we aim for extending the domain-specific programming framework TVL to a virtual vector framework as shown in Figure 5. The extension mainly concerns two aspects. On the one hand, the current hardware-oblivious interface is limited to primitives of the data management domain. This interface will be enhanced with hardware-oblivious primitives required for linear algebra operations. On the other hand, the TVL-backends are currently limited to SIMD extensions as illustrated in Figure 4. Besides SIMD extensions, GPUs and FPGAs also provide data-level parallelism but with a different execution model. To integrate these accelerators, we virtualize GPUs and FPGAs as virtual vector engines

with specific HW-accelerated software-defined SIMD instructions - implemented in CUDA for GPUs and oneAPI for FPGA. This virtual vector concept leaves the SIMD register length as an implementation and optimization choice. A first approach with promising results is published in [F+21]. Based on this domain-specific framework, we can implement data-level parallel operators in C++ in a hardware-oblivious way. From every single source operator implementation, we are then able to automatically derive multiple variants for various HW accelerators.

4.3.2 Fused Pipeline Operators

To optimize the execution of IDA pipelines, operator fusion is a very important technique, which fuses sub-pipelines - a set of contiguous kernels - into a single operator [B+18]. Advantages are avoiding allocation of intermediates, reduced memory bandwidth requirements, and specialization according to queries and operator parameterizations. One more advantage of fuses pipelines: allows to process out of core/in batches. With the DAPHNE compiler, we are able to fuse certain operations, forming the concept of a fused pipeline operator. This means that operators, if they meet certain criteria, qualify to be put together into a container operator - a fused pipeline. The focus of these criteria lies on how the input data to operations can be split and how the output data has to be combined. In MLIR we can express these constraints with the provided compiler infrastructure and have one of the compiler's optimization passes gather chains of operations to put into a "VectorizedPipeline" operator. For the input of an operation, we indicate if the data can be processed if split in a row- or column-wise fashion while for the output characteristics, we specify a combine property of row-wise, column-wise or aggregation. If the split and combine characteristics of subsequent operations match up, a fused pipeline can be created. These chains of operations can then work on slices of data in parallel without materializing full-sized intermediates. The sliced input is wrapped in task items to be queued in the vectorized execution engine that employs runtime scheduling decisions. Worker threads execute the fused pipelines according to the given schedule (WP5). By controlling the task size, we can ensure bounded memory requirements and fit intermediates into the device caches. That way, the entire operator pipeline behaves like a dedicated, hand-crafted kernel. A task is the unit of scheduling with potential worker contention on shared task queues and outputs, and random access to the start of the task data. The more tasks (or the smaller the task size), the higher the overhead but the better for load balancing (see Deliverable D5.1). Separating task size from data binding provides additional flexibility. Furthermore, this wrapping into variably sized tasks allows us to adapt to sparse input patterns. Sparsity exploitation across chains of operations can avoid huge dense intermediates and thus, change the asymptotic runtime behavior, but these opportunities are only leveraged for a limited set of operations and CPUs yet. Beyond this state-of-the-art, we aim to generate sparsity-exploiting fused operators for GPUs, which is challenging due to the irregularities of sparse matrices. Additional focus areas are operator fusion (both compilation and runtime) for multiple, heterogeneous hardware devices, as well as improved optimizer support for generating fused operators for partial data placement on different devices.

4.3.3 Sparsity Exploiting Code Generation for GPU

While it may not be feasible to compile source code at run-time for all types of accelerators (e.g., it takes hours to compile even a simple vector addition sample program for FPGAs), we certainly will engage in code generation for GPU devices using the CUDA language and SDK. Not only does the language allow to express kernels in an elegant and human readable fashion due to CUDA supporting most concepts of the C++ language (up to C++17 at the time of writing). There is also a dedicated runtime compiler, named NVRTC, included in the SDK. This allows to compile binaries from in-memory source code - which will be generated by the Daphne compiler based on the operators that it encounters during execution of the multitude of optimization passes. Previous work has shown the benefit of decoupling data bindings from source code generation [B+18]. By using templates that model the way of certain access patterns, the complexity of generating high performance operators is reduced and can focus on the computation and not become bloated with code that deals with all kinds of problems when dealing with feeding the data to the generated kernel. This complexity is partially "exported" to the DAG (directed acyclic graph) analysis stage, that determines if a certain code template can be applied. These code templates facilitate sparsity exploitation as they can be geared towards being invoked on non zeros, not needing to fall back to densifying the sparse inputs to do computations. And with chains of operations contained in the generated code only being triggered by non zeros, a lot of unnecessary computation can be avoided, thus not only increasing performance but also making certain computations possible. While an implementation of this approach for GPU has its novelty to a certain extent, it is also a challenge yet to be solved to gear dynamically created operators to the irregular input patterns that need to be dealt with during IDA pipeline execution. GPUs and most SIMD style hardware excels at processing regular patterns, e.g., dense inputs, because the hardware and the data are laied out like a grid, fitting together naturally. Thus, processing sparse data will almost certainly be not utilizing the available resources as efficiently. But shaping the way inputs are fed to the processing elements can bring efficiency to tolerable or even favorable levels and brings along other benefits like reduced memory bandwidth requirements. Two examples of methods of adapting sparse inputs to accelerators are row binning [A+14] and adaptive chunking [W+19]. The former processes rows of a sparse matrix that have similar length with similar kernel launch configurations. The latter tries to form uniformly sized chunks of work for the GPU. Both approaches have been applied primarily to matrix-matrix and matrix-vector multiplication.

4.4 Summary

In DAPHNE, we explicitly decided to integrate HW accelerators by implementing kernels in C/C++ on these HW accelerators. Thus, this implementation scope includes the following aspects: (i) implementing key operations on different HW accelerators (T 7.1), (ii) supporting multi-device/multi-accelerator kernels (T 7.4), and (iii) code generation for HW accelerators (T 7.5).

5 Employment of HW Accelerators in IDA Pipelines

While the previous section dealt with the implementation of hardware-accelerated kernels, this section describes our concepts for an efficient and effective employment of these kernels in IDA pipelines. The basic execution model of DAPHNE supports both (1) sequentially invoking (hardware-accelerated) kernels of an IDA pipeline and producing materialized intermediates in

memory with copy-on-write semantics and operator-level synchronization barriers as well as (2) fusing operators to a vectorized pipeline as pursued with a vectorized execution engine. Additionally, we adopt hierarchical scheduling mechanism as well as task-parallel loops and operations.

From a HW accelerator perspective, two aspects play an important role for an efficient and effective execution: operator and data placement. Existing work for both aspects mainly relies on manual or heuristic placement, but there is also work on using reinforcement learning for operator placement onto multiple HW accelerators [L+20b, M+17, R+20]. Recent work further applies self-scheduling schemes across devices [D+19] or partitions the data accordingly to expected device performance [G+19], for utilizing all available devices jointly.

To advance the state-of-the-art and to develop a comprehensive DAPHNE approach, we focus on three important components: (i) memory management, (ii) scheduling, and (iii) cost modeling.

(i) Memory Management: As described in Deliverable D2.1, DAPHNE's core data structures as part of the memory management are dense or sparse matrix formats. Both use row-major representations: a dense linearized one-dimensional array, and a compressed sparse row (CSR) [S94] format of row offsets, column-index and value arrays. While matrices are homogeneous arrays, frames have a schema and thus require the handling of value types. Given common analytic workload characteristics, our frames rely on a column-oriented storage implemented via a dense matrix per column or column group. This composition allows the reuse of matrix operations as frame operations. In hybrid runtime IDA execution plans that utilize HW accelerators, a data object might be partitioned or replicated across devices. For maximum flexibility – for example in programs with conditional control flow – we keep this data-location information in runtime data structures. Specifically, matrices and frames reference the host data (which can be a `nullptr`) and/or data on HW accelerators, computational storage devices [BD21, LB21], and distributed workers.

This meta-data enrichment allows the memory management to track the data placement and to orchestrate the data transfer at any time for each IDA pipeline execution. For example, a compiled GPU kernel is called through its host kernel, which first invokes primitives to make the inputs available in GPU memory. If the data is already on the GPU, there is no additional transfer, and otherwise the primitive utilizes implicit (stream and discard) or explicit (copy and retain) means of data transfer. The decision is made based on the meta-data. In a next step, we will allow the compiler to inject prefetch and broadcast directives to overlay anticipated data transfer with other operators to mask latencies. These distribution primitives nicely generalize to HW accelerators, the distributed runtime, and computational storage. Moreover, these primitives expand the technical possibilities for optimizing the data transfer between HW accelerators.

(ii) Scheduling: Based on the memory management foundation, we will explore different scheduling strategies from WP5 – see Deliverable D5.1 -- with respect to operator and data placements for HW acceleration (T7.2). In particular, we will investigate placement decisions at compile-time as well as runtime. Based on this investigation, we want to derive new placement heuristics for the different kinds of HW accelerators. For example, for GPUs, it is already known

that the data must not be too small, since overhead for the data transfer is then too large. The question is whether this heuristic also applies to FPGA or not, or what other influencing factors, such as prefetching, may exist.

(iii) Cost Modeling: Finally, we aim at developing a cost-based approach for operator and data placement decisions (T7.3 and T7.6). For these cost-based decisions, accurate performance models of the HW accelerators are a key ingredient. Thus, we aim at creating a wide spectrum of performance models from simple analytical models to more complex profiling and learning-based models (requiring profiling for each HW accelerator). Based on that, we further aim to model data access costs along the data path of IDA pipelines under different access characteristics, accelerator capabilities, and accelerator-centric models of parallelism. Then, these models are used during the compile-time of an IDA pipeline, to determine the best-fitting hardware accelerator in combination with the best-fitting data placements. For this purpose, it must be possible to estimate the sizes and properties (e.g., the sparsity of matrices [SB+19]) of the intermediate data accordingly [BH+07, MNS09].

6 Conclusions

This report presented the motivation and the design for hardware accelerator integration in DAPHNE. In particular, we described our approach for hand-crafting as well as generating of hardware-accelerated kernels. Based on that, we presented our concepts for an efficient and effective employment of the hardware-accelerated kernels in IDA pipelines.

7 References

[A+19b] S. Agrawal et al. Machine Learning for Precipitation Nowcasting from Radar Images. CoRR, abs/1912.12132, 2019.

[A+17] S.R. Agrawal et al. A many-core architecture for in-memory data processing. In: MICRO, pp 245–258, 2017

[A17] S. N. M. Albarqouni. Machine Learning for Biomedical Applications: From Crowdsourcing to Deep Learning. PhD thesis, Technische Universitaet Muenchen, 2017.

[A+14] A. Ashari et al. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. ICS 2014: 273-282, 2014.

[BD21] A. Barbalace and J. Do. Computational Storage: Where Are We Today? In CIDR, 2021.

[BH+07] K. S. Beyer et al.: On synopses for distinct-value estimation under multiset operations. SIGMOD 2007.

[BKS20] S. Blanas et al. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. In CIDR, 2020.

[BA+20] M. Boehm et al.: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. CIDR 2020.

[B+18] M. Boehm et al. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. PVLDB, 11(12), 2018.

[BC11] S. Borkar and A. A. Chien. The future of microprocessors. Commun. ACM, 54(5):67–77, 2011.

[B18] M. Bjørling. Open-Channel Solid State Drives. https://openchannelssd.readthedocs.io/en/latest/. 2018, 2022-05-30

[Ca+18] J. Castrillon et al. A Hardware/Software Stack for Heterogeneous Systems. IEEE Trans. Multi Scale Comput. Syst. 4(3): 243-259, 2018

[C+20] Y. R. Choi et al. Matrix-matrix multiplication using multiple GPUS connected by Nvlink. GloSIC, pp. 354-361, 2020.

[D+19] K. Dursun et al. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. PVLDB, 12(12), 2019.

[D+20] P. Damme et al. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. PVLDB 13(11), 2020.

[D+22] P. Damme et al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In CIDR, 2022.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering, 5(1), pp.46-55., 1998

[D+17] M. Dreseler et al. Hardware-accelerated memory operations on large-scale numa systems. ADMS@VLDB, 2017.

[E+19] A. Elgohary et al. Compressed linear algebra for declarative large-scale machine learning. PVLDB 9(12), 2017

[F+21] J. Fett et al. The Case for SIMDified Analytical Query Processing on GPUs. DaMoN@SIGMOD, 14:1-14:5, 2021

[G+21] S. Gan et al. BAGUA: Scaling up Distributed Learning with System Relaxations. CoRR, abs/2107.01499, 2021.

[G+19] M. Gowanlock et al. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In DaMoN@SIGMOD, 2019.

[H+17] S. Haas et al. Application-specific architectures for energy-efficient database query processing and optimization. Microprocess Microsyst. 55:119–130, 2017

[H15] C. J. Hughes. Single-Instruction Multiple-Data Execution. Morgan & Claypool Publishers. 2015

[JP13] R. Johnson and I. Pandis. The bionic DBMS is coming, but what will it look like? In CIDR, 2013.

[J+17] N.P. Jouppi NP et al. In-datacenter performance analysis of a tensor processing unit. https://arxiv.org/pdf/1704.04760.pdf. 2017, Accessed 2022-05-31

[K+13] T. Kiefer et al. Experimental evalu- ation of NUMA effects on database management systems. BTW, pp. 185–204, 2013.

[L+20] C. Lattner et al. MLIR: A Compiler Infrastructure for the End of Moore's Law. CoRR, abs/2002.11054, 2020.

[L+14] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In SIGMOD, 2014.

[Le+14] R. Leißa et al. Sierra: a SIMD extension for C++. WPMVP@PPoPP, 2014.

[LB21] A. Lerner and P. Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In SIGMOD, 2021.

[L+20b] C. Lutz et al. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In SIGMOD 2020.

[M+17] A. Mirhoseini et al. Device Placement Optimization with Reinforcement Learning. In ICML, 2017.

[MNS09] G. Moerkotte et al. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. PVLDB 2(1) 2009.

[M+11] A. Munshi et al. OpenCL programming guide. Pearson Education, 2011

[NEC] NEC. Nec accelerates machine learning for vector computers. http://www.nec.com/en/press/201707/global_20170703_02.html. Accessed 30 May 2022.

[OL18] Ismail Oukid and Lucas Lersch: On the Diversity of Memory and Storage Technologies. Datenbank-Spektrum 18(2): 121-127, 2018.

[P+18] S. Palkar et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. PVLDB 11(9), 2018

[P+10] I. Pandis et al. Data-oriented transaction execution. PVLDB, 3(1):928–939, 2010.

[P+16] H. Pirk et al. Voodoo – A Vector Algebra for Portable Database Performance on Modern Hardware. PVLDB 9(14), 2016

[P+21] T. Pfaff et al. Learning Mesh-Based Simulation with Graph Networks. ICLR, 2021.

[P+21b] C. Pilato et al. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In DATE, 2021.

[R+20] A. Raza et al. GPU-accelerated data management under the test of time. In CIDR, 2020.

[R+21] J. Reinders et al. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL (p. 548). Springer Nature, 2021

[RL16] R. Reyes and V. Lomüller. SYCL: Single-source C++ accelerator programming. In Parallel Computing: On the Road to Exascale (pp. 673-682). IOS Press, 2016

[SK10] J. Sanders and E. Kandrot. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[SB+19] J. Sommer: MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. SIGMOD 2019.

[S+19] N. K. Srivastava et al. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In FCCM, 2019.

[S+17] N. Stephens et al. The ARM scalable vector extension. IEEE Micro, 37(2):26–39, 2017.

[S+11] M. Stonebraker et al. The Architecture of SciDB. In SSDBM, 2011.

[S05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's journal, 30(3):202–210, 2005.

[S+00] A. S. Szalay et al. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In SIGMOD, 2000.

[T17] J. Teubner. Fpgas for data processing: current state. it Inf Technol 59(3):125–131, 2017.

[U+20] A. Ungethuem et al. Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. In CIDR, 2020.

[W+16] L. Wang et al. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. SC. pp. 1-11, 2016

[W+09] T. Willhalm et al. Simd-scan: ultra fast in-memory table scan using on- chip vector processing units. PVLDB Endowment 2(1):385–394, 2009.

[W+19] M. Winter et al. Adaptive sparse matrix-matrix multiplication on the GPU. PPoPP 2019: 68-81, 2019.

[W04] W. H. Wolf. The future of multiprocessor systems-on-chips. DAC, pp- 681–685, 2004.

[Z+19] A. Zamuda et al. Forecasting Cryptocurrency Value by Sentiment Analysis: An HPC-Oriented Survey of the State-of-the-Art in the Cloud Era. In High-Performance Modelling and Simulation for Big Data Applications. 2019.