D7.4 Prototype and overview multi-device operations and placement

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

Version 1.0

PUBLIC

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

Document Description

Previous deliverables already shared the overall design of the integration of hardware (HW) accelerators as well as an initial approach how to develop HW-accelerated kernels and how to anchor these HW-accelerated kernels in the entire DAPHNE infrastructure. This document presents our developed extended concepts for multi-device operations and placement. Moreover, this document shares a snapshot of the developed prototype and describes three examples in more detail.

D7.4 Prototype and overview multi-device operations and placement			
WP7 – Hardware Accelerators			
Type of document	D	Version	[1.0]
Dissemination level	PU		
Lead partner	TUD		
Author(s) Reviewer(s)	Dirk Habich, Johannes Fett, Mark Dokter Lennart Schmidt		
Contributors	DAPHNE Team		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Initial structure and write-up of the introduction	Dirk Habich
V0.2	Initial write-up of Section 3.1	Dirk Habich
V0.3	Finalized Section 3.1 and Section 3.2	Dirk Habich, Johannes Fett
V0.4	Review Comments from Lennart Schmidt incorporated	Dirk Habich

V0.5	Initial write-up of section 3.3	Mark Dokter
V0.6	Incorporated feedback	Mark Dokter
V1.0	Finalization	Dirk Habich

Abbreviations

Abbreviation	Definition
DM	Data Management
DSL	Domain Specific Language
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HBM	High-Bandwitdh Memory
HPC	High-Performance Computing
IDA	Integrated Data Analysis
MIMD	Multiple Instruction Multiple Data
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation
SIMD	Single Instruction Multiple Data

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. This requirement 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 is an old problem. 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 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 reports *D2.1 - Initial System Architecture* [D2.1] and *D2.2 - Refined System Architecture* [D2.2] have described the overall DAPHNE system architecture, report *D7.1 - Design of integration hardware (HW) accelerators* [D7.1] has presented the overall design of the integration of HW accelerators as well as has detailed on accelerated operations and primitives.

As introduced in DAPHNE report D7.1 [D7.1], the challenges for the integration of HW accelerators are (i) developing as well as generating operators - hereafter 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 [D7.1]. While challenge (i) is addressed by *Task 7.1 - Accelerated Key Operations and Data Access Primitives*, *Task 7.4 - Multi-Device Operation Kernels*, and *Task 7.5 - Code Generation for HW Accelerators*, challenge (ii) is considered in *Task T7.2 - Compiler and Runtime Support for HW Accelerators*. Selecting the best-fitting accelerator for efficient execution - challenge (iii) - is part of *Task 7.2 - Compiler and Runtime Support for HW Accelerators* as well as *Task 7.3 - Performance Models and Cost Estimation*.

In the subsequent DAPHNE report D7.2 [D7.2], we demonstrated an initial approach how to develop hardware-accelerated kernels and how to anchor these hardware-accelerated kernels in the entire DAPHNE infrastructure. Additionally, we gave an overview on the devised performance models for a cost-based approach for hardware-accelerated kernels and data placement decisions in a heterogeneous hardware environment. The third DAPHNE report D7.3 [D7.3] of work package 7 enhanced the work presented in D7.2 by introducing extended concepts for code generation of hardware-accelerated kernels and the integration into the

entire DAPHNE infrastructure. On the one hand, we focused on accelerating relational data processing with the usage of Single-Instruction Multiple-Data (SIMD) extensions of general-purpose CPUs as well as FPGAs. In particular, we demonstrated a concept that allows to execute single-source SIMD-oblivious code on both CPU and FPGA without having to consider FPGAs in isolation. On the other hand, we presented how our efforts regarding code generation for sparsity exploitation on GPU are improving end to end performance of selected algorithms available in the DAPHNE source repository.

This fourth and final report summarizes our work and achieved results with regard to multi-device operations and data placement on multiple homogeneous and heterogeneous devices. The remainder of this deliverable is structured as follows:

- In Section 2, we detail the access to our prototype artifacts for this written deliverable document.
- Then, we introduce our prototypes by describing the underlying demonstration scenarios in Section 3.
- Afterwards, we explain the folder structure of our prototype artifacts in Section 4.

2 Artifact Access

The extended prototype is publicly accessible under the following link:

- **Link:** https://tinyurl.com/daphne-D74 (tgz archive > 50 MB)

This is a snapshot of the branch D7.4 of the DAPHNE open-source repository at https://github.com/daphne-eu/daphne (commit as indicated in the file githash.txt in the tgz archive).
The directories contained in the artifact are described in more detail in section 4.

3 Demonstration scenario

In this deliverable, we present three different scenarios, each focusing on a different kinds of multi-device settings:

- [Homogeneous Multi-Core CPU]: The first scenario focuses on accelerating relational data processing with the combined usage of thread-level and data-level parallelism on general-purpose CPUs. The description is based on the already accepted paper [S+25].
- [Heterogeneous CPU/GPU]: The second scenario extends the homogeneous multi-core CPU scenario by adding a GPU for co-processing.
- [Multi-Device GPUs]: The third scenario describes the multi-GPU capabilities of DAPHNE's vectorized engine.

3.1 Homogeneous Multi-Core CPU

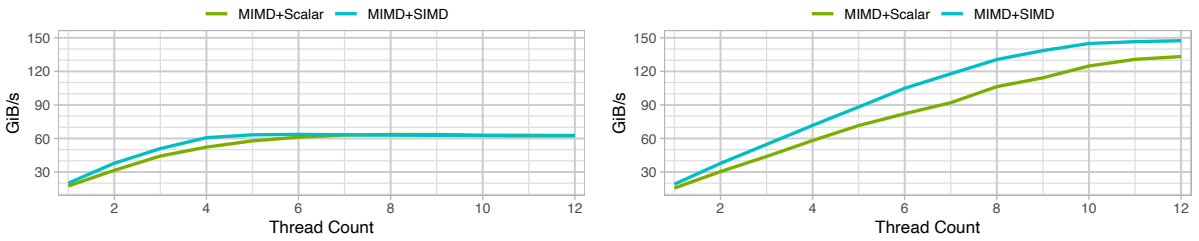
Despite the continuous evolution of computing units with GPUs, FPGAs etc. in mind, the enduring dominance of general-purpose CPUs for an efficient data processing remains a significant fact. The reason for this is that modern general-purpose CPUs still offer high-computational power. This computational power is achieved through three different sources of parallelism: *thread-level parallelism* (based on the Multiple Instruction Multiple Data -- MIMD -- parallel paradigm), *data-level parallelism* (based on the Single Instruction Multiple Data -- SIMD -- parallel paradigm), and *instruction-level parallelism*. State-of-the-art data processing engines usually leverage all three sources of parallelism in a more or less common way to reduce the processing latency:

- **Thread-level parallelism** is usually applied on the level of individual query operators or pipelines (intra-operator/intra-pipeline parallelism) by distributing the input data equally among threads (physical cores). That means the same operator/pipeline is simultaneously executed on multiple (logically) disjoint data partitions (see also [D5.1] for decision on thread-level parallelism).

- **Data-level parallelism** is achieved by explicitly using SIMD instructions within query operators/pipelines. A single SIMD instruction processes multiple data elements simultaneously, increasing the single-thread performance. The SIMD instructions are usually applied to contiguous data elements in main memory.

- **Instruction-level parallelism** is achieved by applying the same operation to a vector of elements or by compiling operations into intertwined pipeline machine code (vectorized processing).

While instruction-level parallelism is about executing several instructions in sequence on loaded or cached data, thread-level and data-level parallelism are used to process data in parallel. Moreover, thread-level and data-level parallelism complement each other and should usually be combined to exploit the full potential of modern CPUs. Nevertheless, thread-level often has a higher priority than data-level parallelism since the main memory bandwidth is already fully saturated with the use of thread-level parallelism. This aspect is clearly shown in the diagram in Figure 1(a) illustrating the measured memory throughput for calculating an aggregating sum of a large array of integer values with increasing thread-level parallelism (increasing number of physical cores) as well as with (+SIMD) and without (+Scalar) using data-level parallelism. As depicted, the use of data-level parallelism only reduces the number of required threads to achieve the maximum throughput of roughly 60GiB/s. The hardware foundation for these experiments was a recent Intel Xeon Max 9648 (Sapphire Rapids architecture) with 48 physical cores on a socket and the experiment was conducted on a single socket with 12 physical cores.



(a) DDR5 main memory. **(b) HBM main memory.**

Figure 1: Comparing multi-threaded throughput for aggregating sum with data in different main memory technologies.

Since our hardware foundation does not only have regular DRAM, we also repeated the same experiment with the whole data in High-Bandwidth Memory (HBM) as well. Interestingly, when executing the same operation on data in HBM (cf. Figure 1(b)), exploiting data-level parallelism becomes mandatory to utilize the interconnect fully. This experimental result clearly shows that the joint utilization of MIMD and SIMD is gaining in importance with developments such as of HBM. However, due to diverging granularities of parallelism and scopes of data accesses, the state-of-the-art interplay of MIMD and SIMD execution paradigms requires different approaches, which creates algorithmic overhead when combining them. Therefore, we argue that it is time to fundamentally rethink the MIMD-SIMD interplay on modern general-purpose CPUs through a unified memory access approach. Before we discuss our novel approach in Section 3.1.2, we will first summarize the state-of-the-art in Section 3.1.1. Then, we describe our prototypical implementation in Section 3.1.3 and present selected evaluation results in Section 3.1.4.

3.1.1 Preliminaries

As mentioned above, parallelism is the name of the game for an efficient processing of analytical queries on general-purpose CPUs. According to Flynn's classification, modern CPUs offer the following hardware parallelization opportunities: (i) Multiple Instruction Multiple Data (MIMD) and (ii) Single Instruction Multiple Data (SIMD). As next, we will discuss the state-of-the-art approaches for both. As a representative running analytical example, we compute the aggregating sum over a large column or array of integer data (contiguous memory area). A scalar implementation of this aggregating sum sequentially iterates over the array and adds up the values one after the other.

MIMD: CPUs offering MIMD have a number of homogeneous processing elements (PE) -- cores -- that operate asynchronously and independently. That means individual PEs may be executing different instructions on different pieces of data at any time. This opportunity -- also called *thread-level parallelism* -- is a heavily used optimization technique in columnar database engines. In detail, MIMD is used to realize a data-partitioned intra-operator parallelism. Here, all data objects, e.g., columns, are logically partitioned and partitions are exclusively accessed by the assigned worker thread that is pinned to a specific PE. With this so-called data-oriented

```
1  uint64_t const * data; /* previously initialized */
2  uint64_t sum = 0;
3  #pragma omp parallel for num_threads(T_CNT) reduction(+:sum)
4  for (size_t i = 0; i < total_elements; ++i) {
5      const auto val = data[i];
6      sum += val;
7  }
```

Listing 1: Basic OpenMP aggregation.

```
1  #pragma omp declare reduction(simd_add :...)
2  uint64_t const * data; /* previously initialized */
3  uint64_t sum = 0;
4  __m512i vec_sum = _mm512_set1_epi64(0);
5  #pragma omp parallel for num_threads(T_CNT) reduction(simd_add : vec_sum)
6  for (size_t i = 0; i < total_elements / VEC_SZ; ++i) {
7      const auto vals = _mm512_loadu_epi64(data + (i * VEC_CNT));
8      vec_sum = _mm512_add_epi64(vec_sum, vals);
9  }
```

Listing 2: Basic OpenMP aggregation with explicit SIMD.

architecture, the same operator or query pipeline is logically simultaneously executed on disjoint data partitions. Data partitions are usually equally sized, so every PE processes the same amount of data, thus minimizing overall runtime.

Listing 1 illustrates our running example implemented as a partitionable loop. The #pragma (*omp parallel)* instructs the compiler to partition the following *for* loop into *T_CNT* partitions (if specified) and consequently assigns a set of loop iterations to a distinct thread. Iterations can be assigned en bloc or interleaved based on a static or dynamic OpenMP scheduler. However, a specific iteration is only processed by a single thread. To avoid data loss or even a segmentation fault, the global *sum* variable must not be written by multiple threads simultaneously Therefore, it could be either made atomic or a thread-local partial sum is calculated, e.g., through the custom *reduction(+:sum)* operation, which only adds up the partial sums at the end of each block.

Thus, this state-of-the-art approach naturally extends the scalar processing to a thread-level parallelism and is widely used. Especially for query operators with a sequential memory access pattern, no sophisticated control mechanisms are required, as there are usually no data or control flow dependencies between the processed partitions. But it can also be used to realize more complex operators. We can conclude that MIMD is used to realize a logically synchronous -- but physically asynchronous – and independent processing of logically partitioned data for an efficient analytical query processing.

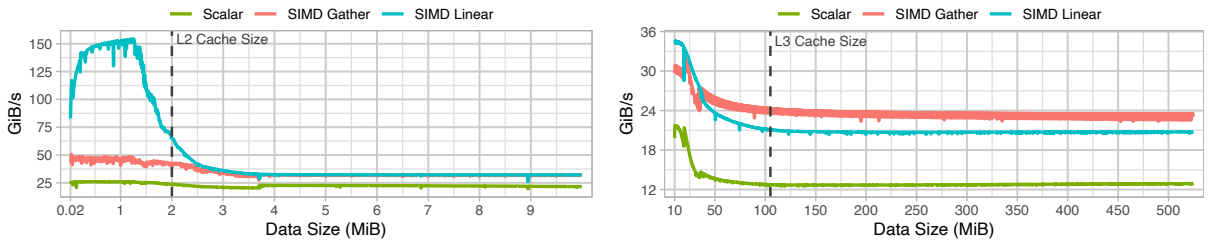
SIMD: Contrary to MIMD, SIMD describes computing units with multiple PEs -- SIMD register lanes -- that perform the same instruction on multiple data elements parallel in lockstep. That means SIMD exploits data-level parallelism, but not concurrency: there are parallel computations, but each PE performs the exact same instruction on different pieces of data at any time. On general-purpose CPUs, each core offers SIMD capabilities through specific SIMD instruction set extensions with varying capabilities (e.g., Intel SSE, AVX2, AVX512, or ARM Neon/SVE). From an abstract point of view, the state-of-the-art SIMD processing for an efficient analytical query processing resembles its scalar counterpart. Data has to be loaded sequentially into a (SIMD-)register, which can then be further processed through explicit intrinsics rather than higher-level abstract operators such as + or -. Lines 7 and 8 in Listing 2 show the exact translation of the scalar variant via AVX512 intrinsics. Based on the SIMD properties and their standard interpretation, we can conclude that SIMD is used to realize a synchronous and dependent processing of consecutive data elements for efficient analytical

processing. This is an entirely different approach compared to the thread-level parallelism discussed above.

Interplay MIMD-SIMD: Listing 2 also shows that linear SIMD processing can be easily enriched with OpenMP to combine both approaches, i.e., thread-level parallelism and data-level parallelism happen in the same operator. On the outer scope, data is divided into a set of logical partitions, which are processed by an individual thread. However, the SIMD approach processes multiple elements simultaneously in the inner scope per partition. This amalgamation employs two orthogonal algorithmic approaches to realize an intra-operator parallelism, although both want to execute the same operator code for different data elements. The difference can be explained in more detail from the data access perspective, in this case, from the underlying array of our running example. While non-contiguous array elements are accessed across all thread-level PEs, only contiguous array elements are accessed by data-level PEs. Moreover, the non-contiguous access is logically equivalent to a strided access, where array elements are accessed equidistantly because the partitions are equally sized. Therefore, the stride distance (or stride size) equals the partition size. However, this strided access is implicitly executed by the thread-level parallelism, as each thread is assigned its own start position in the array and runs through the array from this start position to the end position of the respective data partition.
This interleaving of strided and linear access naturally makes implementing parallel query operators more difficult, as different approaches have to be considered. To overcome that, we argue that the strided access can also be applied to data-level parallelism (SIMD), which thus allows for a unified parallelization concept as follows: On the outer thread-level scope, data is divided into a set of coarse-grained logical partitions, which are processed by an individual thread. On the inner data-level scope, coarse-grained logical partitions are further subdivided into fine-grained partitions, which are processed by individual SIMD register lanes. To supplement our claim, we developed a set of carefully designed microbenchmarks, which are explained in the remainder of the paper. Our microbenchmarks tackle different corner cases for combinations of data access patterns, applied relational query operators, and the underlying storage format.

3.1.2 Unified Data Access for MIMD-SIMD Interplay

SIMD extensions of modern general-purpose CPUs consist of two main building blocks: (i) SIMD registers, which are larger than traditional scalar registers, and (ii) SIMD instructions working on those SIMD registers. Contrary to scalar processing, SIMD registers must be explicitly populated with data elements from main memory using a load or a gather instruction. On the one hand, load is applied, whenever a linear data access pattern is conducted as done in the state-of-the-art for analytical query processing using SIMD capabilities. Linear implies that the accessed data elements are organized as a contiguous sequence like an array. On the



(a) Array Size between 0.02 and 10 MiB. **(b) Array Size between 10 and 500 MiB.**

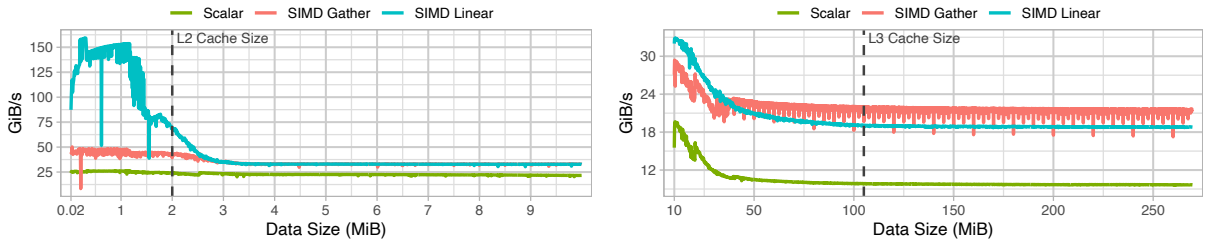
Figure 2 :Comparing single-threaded throughput for aggregating sum with all data in DDR5 main memory and with different access patterns: scalar, linear, and data-partitioned SIMD.

other hand, gather is used when a random-access pattern -- data elements from non-consecutive memory locations -- is required. The general guideline has been that gather should be avoided as far as possible due to the considerable performance loss.

In [H+22], we already have shown that the gather instruction can achieve the same performance as the load instruction with a so-called block-strided access pattern. A memory access pattern is called strided when memory fields accessed are equally distant and the distance is usually denoted as stride size. The particular property of our proposed block-strided access pattern is that the input data, e.g., an array, is logically divided into blocks. In the simplest case, each block consists of k consecutive pages, where k is the number of SIMD lanes of the underlying SIMD register. The blocks are successively processed and for each block, the SIMD processing works as follows: Each SIMD lane is assigned a page from the block and each lane is further responsible for processing the assigned page. To achieve that, a strided access with the page size as the stride size is performed on the block using the gather instruction.

However, the SIMD processing with this block-strided access pattern is quite complicated and requires a two-stage partitioning into blocks as well as pages. To overcome that shortcoming, we investigated the conducted simple partitioning approach of thread-level parallelism for SIMD processing. In this case, data is logically divided into k equally sized partitions in a straightforward way and partitions are exclusively pinned to a specific SIMD lane. To load the corresponding data elements of the k data partitions into the k lanes of a SIMD register, we issue a gather instruction conducting a strided access, whereby the stride size now equals the partition size. Then, the same processing functionality through a SIMD instruction is simultaneously executed on the loaded data elements. Subsequently, the consecutive data elements within the k partitions are loaded until all data elements of the partitions are processed.

In the case of our aggregating sum, each SIMD lane computes a partial sum per data partition like each thread but synchronously. The advantage now is that we apply the same instruction independently for each lane and can, therefore, rely on element-wise SIMD instructions. In the final step, we have to add up the individual partial sums, which we can do through the use of horizontal addition. Figure 3(a) compares the single-threaded throughput results for the aggregating sum operation with all data in DDR5 main memory on our hardware system using



(a) Array Size between 0.02 and 10 MiB.

(b) Array Size between 10 and 500 MiB.

Figure 3: Comparing single-threaded throughput for aggregating sum with all data in HBM main memory and with different access patterns: scalar, linear, and data-partitioned SIMD.

(i) purely scalar processing (Scalar), (ii) AVX-512 SIMD processing with a linear access pattern (SIMD Linear), and (iii) AVX-512 SIMD processing with the described simple data-partitioned approach (SIMD Gather). In the experiments, we varied the array size with randomly generated uint64_t elements in the range from 0.02 to 500 MiB as depicted on the x-axes of both diagrams in Figure 3. As we can see, the linear SIMD variant offers significant throughput advantages if the data fits into the L2 cache (2 MiB on our CPU). However, if the total amount of data exceeds the size of the L3 cache, then our proposed data-partitioned SIMD approach outperforms the linear variant. This effect also applies for AVX2 with 32-bit and 64-bit data types but not for AXV512 with 32-bit data types, which is in line with the results presented in [H+22]. In addition, we repeated the same experiments with all data in HBM2 main memory on our Intel Xeon Max hardware system and Figure 3(b) shows the measured throughput values. The results confirm the DRAM findings for HBM as well.

3.1.3 Implementation and Execution

Traditionally, SIMD is employed in analytical database engines whenever a columnar storage or decomposition storage model (DSM) is implemented. Two reasons are decisive for this: (i) only the columns that are relevant to the query need to be read, and (ii) the values per column are stored contiguously and can therefore be processed very well with a linear access pattern.

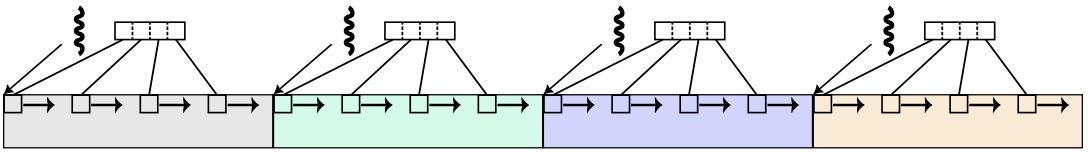


Figure 4: Unified data access pattern for a combined outer MIMD and inner SIMD aggregating sum.

In the previous section, we showed that our data-partitioned SIMD processing is on-par or even better than a linear SIMD processing with the limitation to a single-threaded and single-column environment. This section transfers this finding in the multithreaded environment and to the DAPHNE system.

Figure 4 and Listing 3 show our proposed unified parallelization concept for thread-level and data-level parallelism illustrated on our aggregating sum example. In the illustration of Figure 4, each of the four threads is assigned to a contiguous, coarse-grained and equally sized

```
#pragma omp parallel num_threads(T_CNT) reduction(+:sum)
{
  uint64_t sum = 0;
  const size_t offset = elements_per_thread / VEC_SZ;
  const __m512i idx = _mm512_setr_epi64(
    0, 1 * offset, 2 * offset, ..., 7 * offset);
  for (size_t i = 0; i < offset; ++i) {
    const auto vals = _mm512_i64gather_epi64(
      idx,
      data + (omp_get_thread_num() * elements_per_thread) + i,
      sizeof(uint64_t));
    sum += _mm512_reduce_add_epi64(vals);
  }
}
```

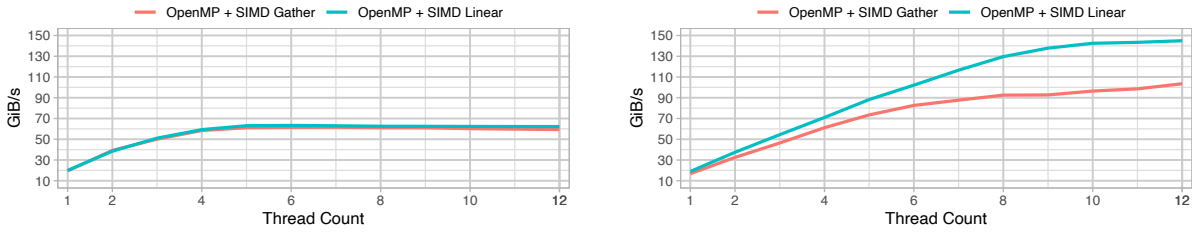
Listing 3: Hierarchical partitoned aggregating sum using OpenMP and strided access.

partition of the data column. Within each logical partition, we subdivide again into four smaller partitions, whereby each equally sized fine-grained partition is now processed by one of the register lanes in parallel. To apply the same partitioning scheme, the MIMD stride size equals one-fourth of the data size, whereas the SIMD stride size equals one-fourth of the coarse-grained partition size. Hence, the access pattern for both the global and local computation follows the same pattern. The corresponding code snippet is depicted in Listing 3.

To demonstrate the feasibility of our approach, we implemented the different concepts for the aggregating sum within the DAPHNE code base using the kernel extension concept. To execute this demonstrator, the following steps are necessary:

1. Build daphne
   ./build.sh
2. Switch to the corresponding extension folder
   cd scripts/examples/extensions/mimd_simd_interplay
3. Build all extensions within this folder
   make
4. Go back to the DAPHNE root directory
   cd ../../../..
5. Execute the aggregating sum using the state-of-the-art MIMD-SIMD interplay
   ./bin./daphne –kernel-ext scripts/examples/extensions/mimd_simd_interplay/myKernels.json scripts/examples/extensions/mimd_simd_interplay/AggSIMDLoad.daphne
6. Execute the aggregating sum using the state-of-the-art MIMD-SIMD interplay
   ./bin./daphne –kernel-ext scripts/examples/extensions/mimd_simd_interplay/myKernels.json scripts/examples/extensions/mimd_simd_interplay/AggSIMDGather.daphne

In both cases, we populate an array with randomly generated integer values and compute a sum over these values. The corresponding DAPHNE scripts are: AggSIMLoad.daphne and AggSIMDGather.daphne. In both cases, the sum is displayed as output, which must be identical, since the same data is generated. To execute both examples, a general-purpose Intel CPU with a least 8 cores and the SIMD extension AVX512 is required.



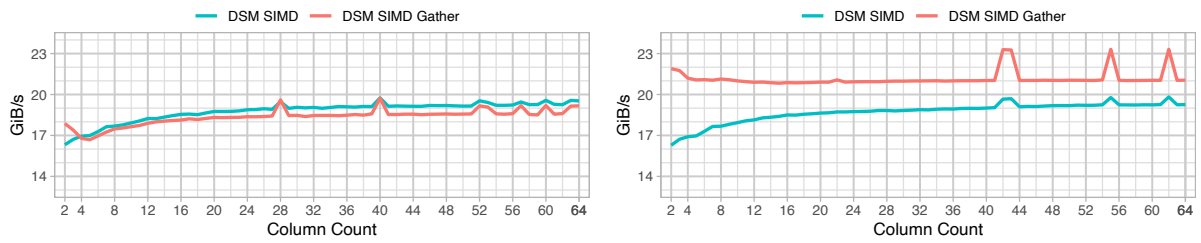
(a) Throughput on DRAM. **(b) Throughput on HBM.**

Figure 5: Througput for OpenMP with a linear and data-partitioned aggregating sum on SIMD with different memory types.

Now, Figure 5 compares the achieved performance results for the aggregating sum of the state-of-the-art OpenMP approach from Listing 2 to our proposed unified data-partitioned approach as outlined in Listing 3 on our tested hardware platform. In our evaluation, we varied the explicit *T_CNT* parameter from 1 to 12, since our hardware exhausts the memory bus with 12 concurrent threads in a memory-bound scenario. Further, we tested different data placements for both DRAM and HBM. When the data was placed in DRAM, cf. Figure 5(a), we can observe a plateau forming at around 63 GiB/s, for both SIMD-linear and SIMD-Gather. The right-hand side of this figure shows the results of the same experiment but with data placed in HBM. However, no such plateau can be observed for either variant, leaving a throughput gap between the two of about 40 GiB/s. We used the code from the mentioned listings for this experiment and fixed the coarse-grained partition size per thread to 256 MiB, filled with 64-bit unsigned integer values. Consequently, when increasing *T_CNT*, the total amount of data increases, but the coarse- and fine-grained partition size and, hence, both the outer and inner stride size stays constant. We used AVX512 for SIMD processing, which provides 8 SIMD lanes for 64-bit values. The general finding of this experiment emphasizes our claim that, at least for DRAM, our proposed data-partitioned SIMD processing can be applied to the inner loop part while maintaining a comparable throughput.

3.1.4 Furter Evaluation Results

To analyze the effect of the slightly lower throughput compared to the microbenchmark results from Section 3.1.2, we take a step back to the single-threaded execution but with multiple columns now. We expanded our aggregating sum into a filter-aggregating sum scenario to do this. This scenario works as follows: Assuming we have X columns (table with X attributes as frame), then a filter condition is checked on each of the X-1 columns. Only the last attribute is used for the aggregation if the tuple qualifies, and we observe that the selectivity of the filter has no visible impact since we have to read all the data in any case. In Figure 6(a), we have chosen a column size of 128 MiB and used this size for all columns. Since we consider AVX512 and 64-bit integer data, our fine-grained partitions have a size of 16 MiB in this case. This size is also our stride size for the necessary strided access. Moreover, each fine-grained partition



(a) Column Size fixed to 128 MiB. **(b) Column Size fixed to 127.98 MiB.**

Figure 6: Filter-aggregating sum comparison for linear load and gather with data in DRAM.

occupies 4096 pages in this case since our system uses a page size of 4 KiB. This results in a setting where the fine-grained partitions are page-aligned, and therefore, the strided access is also page-aligned. The strided access now loads the following data: The first gather instruction loads the elements at position 0 of 8 different pages, whereby the page distance between two elements is 4096 pages (stride size of 16 MiB). The subsequent gather instructions load the following elements with ascending positions. As illustrated in Figure 6(a), the achieved throughput of the data-partitioned SIMD processing (DSM SIMD Gather) is slightly slower compared to linear SIMD processing (DSM SIMD). This is also consistent with the results as presented in [H+22].

Based on this finding, however, the question arises as to why the microbenchmark results from Section 3.1.2 have shown a performance benefit. The reason for this can only be the page alignment. To investigate that, we reduced the total amount of data by a small fraction in a second experiment, i.e., to 127.98 MiB per column, which in turn means that not every fine-grained partition starts on a new page. As visible in Figure 6(b), the linear SIMD processing achieves the same throughput results as for the column size of 128 MiB (cf. Figure 6(a)). However, now our data-partitioned SIMD processing clearly outperforms the linear variant. The main takeaway from this experiment is, that we achieve higher throughput values compared to the state-of-the-art when the resulting fine-grained partitions are not perfectly page aligned. We suspect the hash function of the cache system to play a crucial role in this effect since it might hash accessed data elements to conflicting cache positions.

3.1.5 Interim Conclusion

Exploiting parallelism is crucial to achieve low latency for analytical queries. For example, modern general-purpose CPUs offer high-computational power through two data-oriented parallel paradigms: MIMD and SIMD. As both tackle different granularities, data has to be partitioned differently to satisfy their respective requirements, which in turn creates algorithmic overhead when combining them. To overcome that, we showed that we can seamlessly transfer the MIMD partitioning to SIMD due to the recent advances for SIMD on CPUs. Moreover, we clearly demonstrated that the resulting unified parallelism approach offers several advantages.

From our perspective, our proposed approach offers interesting points for future work. On the one hand, the transfer of MIMD techniques for SIMD should be further investigated. Our approach can only be seen as an initial attempt showing the potential. On the other hand, it would of course also be very interesting to investigate whether the simple OpenMP programming approach for MIMD can also be transferred to SIMD. This would (i) simplify the SIMD utilization and (ii) improve the possibilities for autovectorization (with an deep integration into the DAPHNE compiler).

3.2 Heterogeneous CPU/GPU

In contrast to CPUs, GPUs leverage a larger amount of much simpler compute cores. While a top end general-purpose CPU has 192 cores, a GPU can have about 15.000 cores. These are partitioned into streaming multi processors which contain between 32 and 128 cores. Considering the memory hierarchy, streaming multiprocessors have a small amount of fast on die memory and are connected to a large amount of VRAM by a shared memory bus. The programming model of GPUs is called Single Instruction Multiple Threads (SIMT). Every instruction is performed on a group of 32 threads. If an instruction cannot be executed on 32 threads, it needs to be executed multiple times, leading to performance losses. Programs running on GPUs are called Kernels. A Kernel is executed with many thousands threads which are partitioned into blocks. One block is assigned to one streaming multiprocessor.
There are different approaches to program GPUs. While vendor specific languages like CUDA only work on GPUs from the specific Vendor there are also more generic approaches. SYCL allows parallel code to run on GPUs, FPGAs (as shown in deliverable D7.3) and CPUs across different hardware vendors. As it is important to allow a data processing systems to run on a variety of hardware, SYCL has been chosen as GPU programming model. Out of the different SYCL implementations, the Intel oneAPI toolset has been used to evaluate heterogeneous CPU GPU co-processing for Daphne. The Intel DPC++ compiler is based on LLVM/Clang. To execute kernels with SYCL, a SYCL device needs to be chosen. This can be a CPU, an Intel based GPU or an Nvidia GPU. Both accelerator and integrated GPUs can be used with SYCL. One kernel code is mapped to a specific hardware by the compiler. Thus, a single hardware-oblivious kernel can be run on different hardware devices from different vendors with SYCL, unlike CUDA.

To explore the potential benefit of CPU-GPU coprocessing as addition the demonstrated benefit of CPU-FPGA co-processing in DAPHNE deliverable D7.3 [D7.3], a prototype has been integrated into DAPHNE. It takes two columns as input, performs an element-wise operations and writes back the result to a third column. It uses SYCL unified memory, which provides a pointer to memory that can be accessed both from kernel code on GPU and C++ Code on CPU. This simplifies development as memory transfers are handled at driver level if necessary. For the DAPHNE prototype, an integrated Intel GPU has been evaluated for coprocessing. The input data is partitioned into two parts. Each part runs in a different thread. One part runs in parallel on CPU cores using OpenMP. The other part is executed on a GPU using SYCL in parallel. As unified memory is used, no additional synchronization steps between devices are needed.

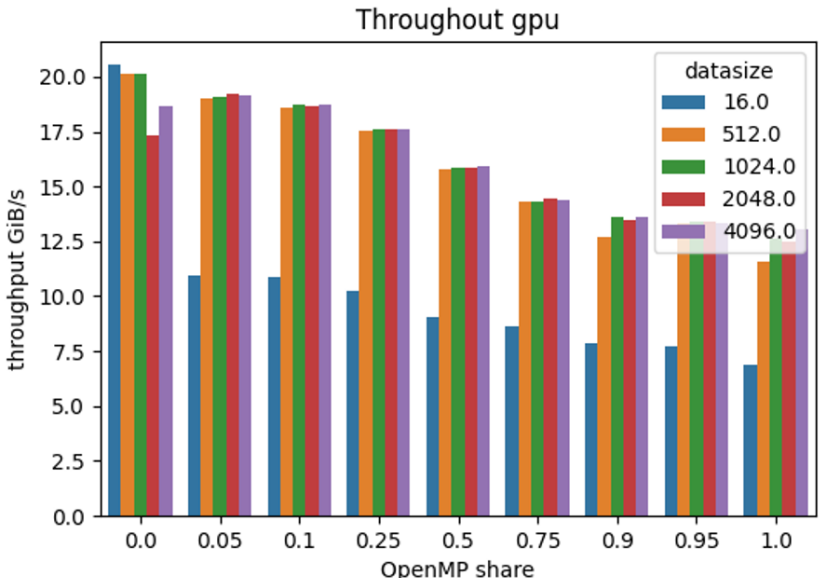


Figure 7: CPU-GPU co-processing evaluation results.

Our achieved results as depicted in Figure 7 shows that there is no performance benefit to run CPU-GPU co-processing instead of using only the GPU. For this reason, we have not pursued a deeper integration into DAPHNE beyond the CPU-GPU co-processing prototype. On the x-axis in the diagram in Figure 7, the distribution between GPU SYCL and CPU based OpenMP is shown. At OpenMP share 0, only the GPU processes data, which leads to the highest observed throughput.

For the corresponding prototype, the elementwise addition operation is overloaded using a kernel extension called *coproc*. This extension can be found in the folder *scripts/examples/extensions/cpu_gpu_coproc*. To run our example, it needs to be compiled as a DAPHNE extension. The hardware requirement is to have a SYCL capable GPU device. Which can be either an intel CPU with an integrated graphics or a Nvidia GPU. The software requirements are listed in the DAPHNE documentation. Additionally, toolset and drivers of the chosen GPU backend are needed. In case of an Intel GPU, the OneAPI tool set is required. If

you chose to use a Nvidia GPU, a working CUDA GPU driver and the matching toolset version including compiler and libraries is needed. To change the device executing the SYCL kernel in our implementation, a different SYCL device selector has to be chosen in the file myKernels.cpp: *sycl::gpu_selector_v* executes on a GPU, while *sycl::cpu_selector_v* runs on the CPU.

To execute our provided example, please execute the following steps:

1. Go to the directory
   cd scripts/examples/extensions/cpu_gpu_coproc/
2. Build the CPU-GPU coprocessing extension as shared library for daphne
   make
3. Go back to the DAPHNE root directory
   cd ../../../..
4. Execute the sample DAPHNE script
   ./bin/daphne --kernel-ext scripts/examples/extensions/cpu_gpu_coproc/myKernels.json scripts/examples/extensions/cpu_gpu_coproc/ewAddCoproc.daphne

The expected output will look like this:

```
Starting coproc() function
PRINT CONFIG
vector_size: 10000
CPU Share: 0.5
GPU Start index: 0
omp threads: 6
Processing mode: Co-processing
Running on device: NVIDIA GeForce GTX 1070 Ti
19992627
```

Figure 8: DAPHNE output of CPU/GPU micro benchmark

An elementwise addition will be performed as described in this section using CPU-GPU coprocessing. Afterwards DAPHNE runs an aggregation by computation a total sum of all output elements. This is shown in the last line of the screenshot. To summarize this section, a flexible co-processing approach that can be run on any SYCL capable GPU in parallel to CPU has been realized by the DAPHNE kernel extension.

3.3. Multi-Device GPUs

In this section we describe the use of multiple GPU devices at the same time when running DAPHNE scripts. While the previous methods make use of SYCL through Intel's OneAPI, the method described here is based on DAPHNE's older (as in more

mature) GPU support through the Nvidia CUDA API. Consequently, this solution restricts the user to a single vendor. As SYCL-support evolves, this shortcoming will be attenuated.
DAPHNE's multi-threading model is implemented using a task queueing paradigm that spawns worker threads that consume these tasks. This feature, that was already described in previous deliverables [D2.1, D5.1] and publications [D+22]. We now extend the vectorized engine with a work queue for GPU tasks and multiple worker threads – one thread per GPU device. A schematic explaining this setup is displayed in figure 9. While it is possible to mix CPU and GPU execution with the vectorized engine, we put the multi-device operation in focus in this example and therefore do not put tasks into the CPU queues.

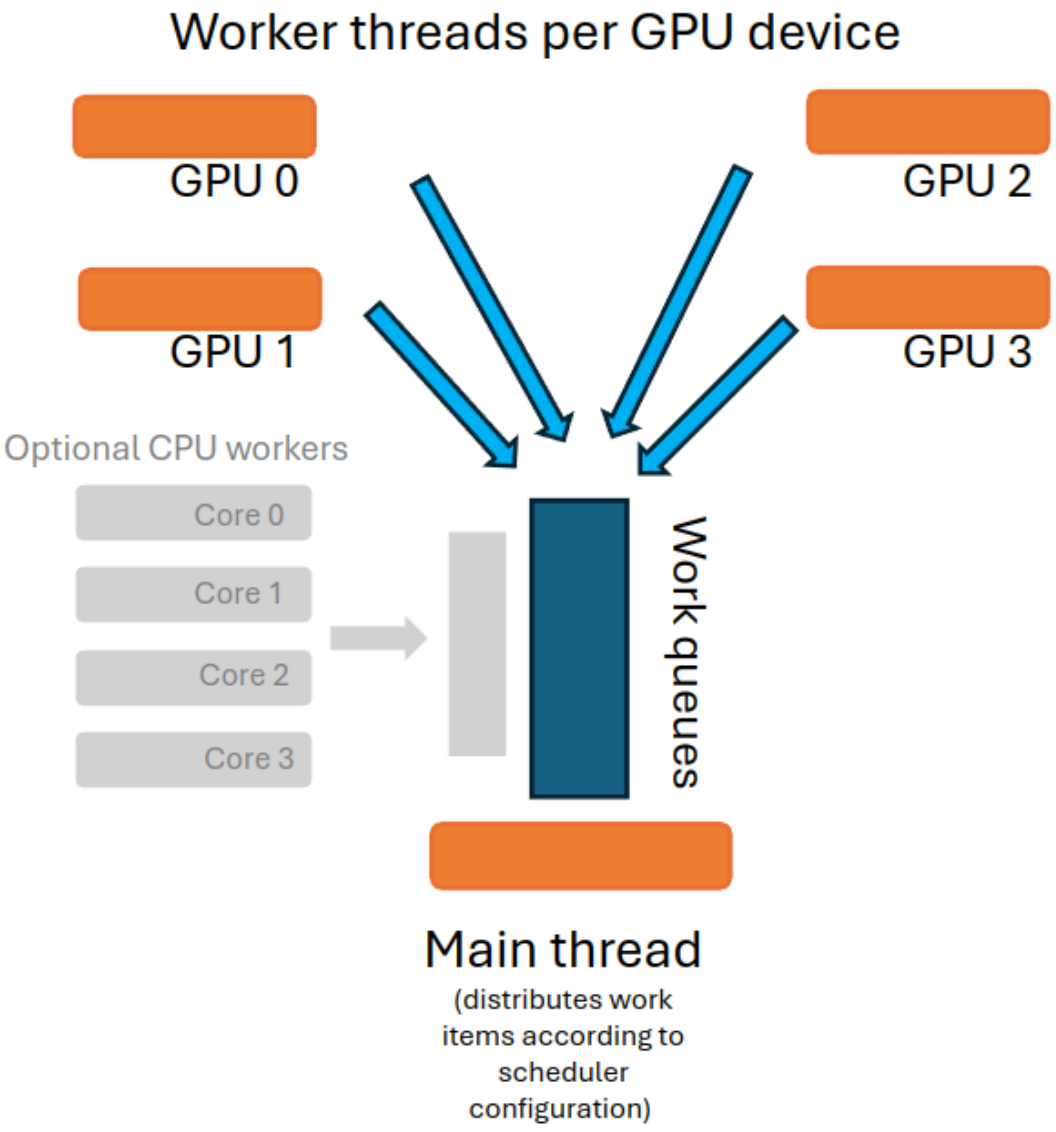


FIgure 9: Vectorized engine work queues

The setup for an upcoming paper experiment will be running as a job in a singularity container on a SLURM cluster. The system configuration is a single socket Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz with 32 cores and 512 GB of memory. Each node hosts four A100 GPU devices (see figure 10 for details). As a workload we chose a task that is one of GPU's prime disciplines. Coming from a computer graphics background, this is single precision matrix multiply. Nevertheless, it is worth mentioning that the A100 can do double precision calculations with *only* a 50% slow-down. In other words, the performance ratio of this device between single and double precision is 1 : 2, making it a suitable platform for high precision scientific simulations, where this overhead is often put up with for the sake of improved results. This double precision performance is one of the reasons (amongst extra high memory bandwidth and a build quality for 24/7 operation) that makes this type of device very expensive. To put this in a better context – consumer graphics cards (also at the high end of the performance spectrum) yield a floating point performance ratio of 1 : 32 or even 1 : 64 and rely on active cooling, while data center devices like the A100 require rack servers with a cooling design that accommodates the requirements for operating such a device. The source code of our micro benchmark is listed in DaphneDSL below in code listing 1.

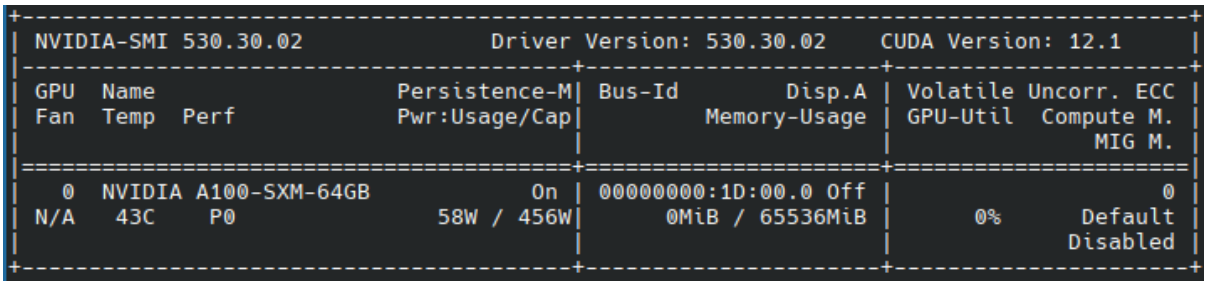


FIgure 10: GPU information for multi device micro benchmark

To execute the provided example, go to the directory where you have built DAPHNE and issue the command

```
bin/daphne --vec --cuda --timing --config UserConfig.json --explain=kernels -select-matrix-repr D7.4/d74_3mm.daph x=1000 y=1000
```

In the example above, an input matrix dimension of 1000x1000 is chosen. DAPHNE will print out the intermediate representation (IR) of the chosen kernel calls and will yield a script output something like this:

```
9.99805e+14
{"startup_seconds": 0.165294, "parsing_seconds": 0.0016899, "compilation_seconds": 0.334608, "execution_seconds": 0.154946, "total_seconds": 0.656538}
```

The command output above lists the sum of all elements in the matrix multiply result and some timing information. This sample output was generated on a local development server with a Tesla T4 GPU to preview the implementation while some details for the final paper experiments are sorted out.

```
# Copyright 2024 The DAPHNE Consortium
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

x=$x;
y=$y;

# Initialize test inputs
X = rand(x, y, 0.0f, 2000, 1, 12345);
Y = rand(y, x, 0.0f, 2000, 1, 67890);

# Calculate the matrix product
s = X @ Y;
z = sum(s);

# Print the sum.
print(z);
```

Code Listing 1: Matrix multiply micro benchmark in DaphneDSL

```
IR after kernel lowering:
module {
  func.func @main() {
    %0 = "daphne.constant"() {value = 12345 : si64} : () -> si64
    %1 = "daphne.constant"() {value = 0.000000e+00 : f32} : () -> f32
    %2 = "daphne.constant"() {value = 67890 : si64} : () -> si64
    %3 = "daphne.constant"() {value = false} : () -> i1
    %4 = "daphne.constant"() {value = true} : () -> i1
    %5 = "daphne.constant"() {value = 1.000000e+00 : f64} : () -> f64
    %6 = "daphne.constant"() {value = 2.000000e+03 : f32} : () -> f32
    %7 = "daphne.constant"() {value = 1000 : index} : () -> index
    %8 = "daphne.constant"() {value = 94026496311968 : ui64} : () -> ui64
    %9 = "daphne.constant"() {value = 94026496311840 : ui64} : () -> ui64
    %10 = "daphne.constant"() {value = 94026496311712 : ui64} : () -> ui64
    %11 = "daphne.constant"() {value = 140721229647768 : ui64} : () -> ui64
    %c0_i32 = arith.constant 0 : i32
    %12 = "daphne.call_kernel"(%11, %10, %9, %8, %c0_i32) {callee =
"_createDaphneContext__DaphneContext__uint64_t__uint64_t__uint64_t__uint64_t"} : (ui64, ui64, ui64, ui64, i32) ->
!daphne.DaphneContext
    %c1_i32 = arith.constant 1 : i32
    "daphne.call_kernel"(%c1_i32, %12) {callee = "CUDA_createCUDAContext"} : (i32, !daphne.DaphneContext) -> ()
    %c2_i32 = arith.constant 2 : i32
    %13 = "daphne.call_kernel"(%7, %7, %1, %6, %5, %0, %c2_i32, %12) {callee =
"_randMatrix__DenseMatrix_float__size_t__size_t__float__float__double__int64_t"} : (index, index, f32, f32, f64, si64, i32,
!daphne.DaphneContext) -> !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>
    %c3_i32 = arith.constant 3 : i32
    %14 = "daphne.call_kernel"(%7, %7, %1, %6, %5, %2, %c3_i32, %12) {callee =
"_randMatrix__DenseMatrix_float__size_t__size_t__float__float__double__int64_t"} : (index, index, f32, f32, f64, si64, i32,
!daphne.DaphneContext) -> !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>
    %15 = "daphne.vectorizedPipeline"(%13, %14, %3, %7, %7, %12) ({
    ^bb0(%arg0: !daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, %arg1:
!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, %arg2: i1):
      %c4_i32 = arith.constant 4 : i32
      %17 = "daphne.call_kernel"(%arg0, %arg1, %arg2, %arg2, %c4_i32, %12) {callee =
"_matMul__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float__bool__bool"} :
(!daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i1, i1, i32,
!daphne.DaphneContext) -> !daphne.Matrix<?x?xf32:sp[1.000000e+00]>
      %c5_i32 = arith.constant 5 : i32
      "daphne.call_kernel"(%arg1, %c5_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
      %c6_i32 = arith.constant 6 : i32
      "daphne.call_kernel"(%arg0, %c6_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
      "daphne.return"(%17) : (!daphne.Matrix<?x?xf32:sp[1.000000e+00]>) -> ()
    }, {
    ^bb0(%arg0: !daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, %arg1:
!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, %arg2: i1):
```

```
    %c7_i32 = arith.constant 7 : i32
    %17 = "daphne.call_kernel"(%arg0, %arg1, %arg2, %arg2, %c7_i32, %12) {callee =
"CUDA_matMul__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float__bool__bool"} :
(!daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i1, i1, i32,
!daphne.DaphneContext) -> !daphne.Matrix<?x?xf32:sp[1.000000e+00]>
    %c8_i32 = arith.constant 8 : i32
    "daphne.call_kernel"(%arg1, %c8_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
    %c9_i32 = arith.constant 9 : i32
    "daphne.call_kernel"(%arg0, %c9_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<?x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
    "daphne.return"(%17) : (!daphne.Matrix<?x?xf32:sp[1.000000e+00]>) -> ()
  }) {combines = [1], operand_segment_sizes = array<i32: 3, 1, 1, 1>, splits = [1, 0, 0]} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i1,
index, index, !daphne.DaphneContext) -> !daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>
  %c10_i32 = arith.constant 10 : i32
  "daphne.call_kernel"(%14, %c10_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
  %c11_i32 = arith.constant 11 : i32
  "daphne.call_kernel"(%13, %c11_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
  %c12_i32 = arith.constant 12 : i32
  %16 = "daphne.call_kernel"(%15, %c12_i32, %12) {callee = "CUDA_sumAll__float__DenseMatrix_float"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> f32
  %c13_i32 = arith.constant 13 : i32
  "daphne.call_kernel"(%15, %c13_i32, %12) {callee = "_decRef__Structure"} :
(!daphne.Matrix<1000x1000xf32:sp[1.000000e+00]>, i32, !daphne.DaphneContext) -> ()
  %c14_i32 = arith.constant 14 : i32
  "daphne.call_kernel"(%16, %4, %3, %c14_i32, %12) {callee = "_print__float__bool__bool"} : (f32, i1, i1, i32,
!daphne.DaphneContext) -> ()
  %c15_i32 = arith.constant 15 : i32
  "daphne.call_kernel"(%c15_i32, %12) {callee = "_destroyDaphneContext"} : (i32, !daphne.DaphneContext) -> ()
  "daphne.return"() : () -> ()
 }
}
```

4 Prototype structure

This project structure shows most of the important directories of the prototype.

- **bin/** (compiled system and parser; generated via build.sh)
- **containers/** (Docker container specific files)
- **data/** (data files for experimenting)
- **doc/** (basic setup and developer documentation)
- **D7.4/** (scripts for running the multi-GPU example)
- **lib/** (generated kernel libraries)
- **scripts/** (DaphneDSL scripts as examples)
    - **examples/**
        - **extensions.** (multi-CPU and CPU-GPU examples using SYCL; based on DAPHNE's kernel extension mechanism)

- **src/** (main source code repository)
  - **api/** (cli including daphne which orchestrates the remaining components)
  - **compiler/** (execution, explain, inference, lowering)
    - **codegen/** (code generation source)
    - **lowering/** (compiler passes)
  - **ir/** (DaphneIR including the DAPHNE MLIR dialect)
  - **parser/** (DaphneDSL, SQL)
    - **sql/** (SQL Parser)
  - **runtime/** (distributed, local including data, I/O, kernels, and vectorization)
    - **local/kernels/** (kernels)
      - **CUDA/** (CUDA device kernels)
      - **FPGA/** (FPGA device kernels)
  - **util/** (helper functions etc.)
- **test/** (test suite of component and integration tests, organized by components)
- **thirdparty/** (dependencies such as llvm, including their build directories)
- **build.sh** (build script to build the DAPHNE compiler)
- **test.sh** (Daphne test suite)

References

[B+18]	Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, Niketan Pansare: "On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML", PVLDB, 2018
BA+20	M. Boehm et al.: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. CIDR 2020.
[D2.1]	DAPHNE: D2.1 Initial System Architecture
[D2.2]	DAPHNE: D2.2 Refined System Architecture
[D3.1]	DAPHNE: D3.1 Language Design Specification, EU Project Deliverable
[D5.1]	DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks
[D7.1]	DAPHNE: D7.1 Design of integration HW accelerators
[D7.2]	DAPHNE: D7.2 Prototype and overview HW accelerator support and performance models.

[D7.3]	DAPHNE: D7.3 Prototype and overview code generation framework.
[D+22]	Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pınar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz Wrosz, Aleš Zamuda, Ce Zhang, and Xiao Xiang Zhu. "DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines", In 12th Annual Conference on Innovative Data Systems Research (CIDR 2022).
DGHI-#633	DAPHNE GitHub Issue #633 - https://github.com/daphne-eu/daphne/pull/633
[H+22]	Dirk Habich, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, Wolfgang Lehner: „To use or not to use the SIMD gather instruction?", In DaMoN@SIGMOD 2022, pages 9:1-9:5
S+25	Lennart Schmidt, Johannes Pietrzyk, Juliana Hildebrandt, Alexander Krause, Dirk Habich, Wolfgang Lehner: "Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines", In 15th Annual Conference on Innovative Data Systems Research (CIDR 2025).