D7.3 Prototype and overview code generation framework

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

Version 1.2

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 code generation of HW-accelerated kernels. Moreover, this document shares a snapshot of the developed prototype and describes two examples in more detail.

D7.3 Prototype and overview code generation framework			
WP7 – Hardware Accelerators			
Type of document	D	Version	1.2
Dissemination level	PU		
Lead partner	TUD		
Author(s) Reviewer(s)	Dirk Habich (TUD), Mark Dokter (KNOW) Matthias Boehm (TUB), Marcus Paradies (TUI)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Initial structure	Dirk Habich
V0.2	Write-up introduction	Dirk Habich
V0.3	Initial codegen text	Mark Dokter
V0.4	Initial text for SIMD/FPGA example	Dirk Habich
V0.5	Enhanced description for SIMD/FPGA example	Dirk Habich

V0.6	Incorporated feedback by Matthias Boehm (TUB) and Marcus Paradies (TUI)	Dirk Habich
V0.7	Minor corrections according to comments	Mark Dokter
V1.0	Final corrections and additions	Mark Dokter
V1.1	Incorporating more feedback	Mark Dokter
V1.2	Fixing document template and PDF rendering issues	Eva Paulusberger, Mark Dokter

Abbreviations

Abbreviation	Definition
DM	Data Management
DSL	Domain Specific Language
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HW	Hardware
IDA	Integrated Data Analysis
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 follow-up 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. This specific DAPHNE deliverable D7.3 builds on D7.2 and describes extended concepts for code generation of hardware-accelerated kernels and the integration into the entire DAPHNE infrastructure. Thus, this report summarizes the work and achieved results developed in Task T7.5.

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 snapshot of the DAPHNE development repository including scripts and binaries to run some of the examples described in this document. It is available under the following link:

- **Link:** https://tinyurl.com/daphne-D73

This snapshot is a copy of the DAPHNE open-source repository at https://github.com/daphne-eu/daphne (branch D7.3).

3 Demonstration scenario

In this deliverable, we present two different scenarios, each focusing on a different hardware accelerator type:

- [SIMD/FPGA-Example]: The first scenario focuses on accelerating relational data processing with the usage of Single-Instruction Multiple-Data (SIMD) extensions of general-purpose CPUs as well as FPGAs.
- [GPU Example]: The second scenario summarizes how our efforts regarding code generation for sparsity exploitation on GPU are improving end to end performance in an example algorithm from our collection of supported algorithms available in the DAPHNE source repository.

3.1 SIMD/FPGA-Example

In this part, we give an introduction to our work on extending our domain-specific framework TVL/TSL (Template SIMD Library) to support Intel FPGAs. Generally, the Single Instruction Multiple Data (SIMD) paradigm has become a core technique to improve data processing on modern general-purpose CPUs. SIMD is characterized by the fact the same operation is simultaneously applied on multiple data elements within a single instruction. Modern CPUs provide direct support of such SIMD capabilities using an increasing variety of SIMD instruction set extensions such as AVX2, AVX512 on Intel or NEON, SVE on ARM systems. To make the most out of SIMDified code, industry and research have invested considerable effort and time into designing and developing SIMD abstraction libraries to address the challenges of SIMD variety and portability of highly optimized SIMDified code. Without claim of completeness, XSIMD, Google Highway, or our TVL/TSL are representative examples. Those libraries are usually implemented in C/C++ for performance reasons allowing (i) to implement SIMD-oblivious query operators and (ii) to enable compile-time deduction as well as code generation for a specific, available SIMD extension with usually negligible runtime overhead. The resulting separation into SIMD-oblivious operators or kernels and a SIMD abstraction library greatly reduces programming and code management complexity with a clear separation of concern.

Besides SIMD, Field Programmable Gate Arrays (FPGAs) are becoming an increasingly viable option to implement efficient data processing due to increasing compute capabilities as well as high-bandwidth access between host and device memory. However, the complexity of the required low-level programming was a limiting factor resulting in a second code base for FPGA operators besides the ones for SIMD-oblivious operators. Meanwhile, FPGA manufacturers also offer tools that allow programming for FPGAs in higher-level languages such as C/C++ or OpenCL. Most recently, Intel has released a new and powerful unified programming model called oneAPI to facilitate the development among various hardware architectures, including FPGAs. Intel 's oneAPI's cross-architecture language Data Parallel C++ (DPC++) is based on the SYCL standard for heterogeneous programming in C++.

Generally, SIMD and FPGA have many architectural features in common. Therefore, our idea is to unify SIMD and FPGA by considering FPGAs as SIMD processing units. With this way of thinking, FPGAs can be included as another backend option in SIMD abstraction libraries such as TVL/TSL [H+23]. This allows to execute single-source SIMD-oblivious code on both CPU and FPGA without having to consider FPGAs in isolation.

3.1.1 Seamless Integration of Intel FPGAs

On the one hand, the existing SIMD abstraction libraries such as our TVL/TSL library generally focus on supporting common SIMD instruction set extensions from Intel and ARM as backends. On the other hand, Intel’s oneAPI is a software development kit providing a unified

programming model for diverse architectures such as CPUs, GPUs, and FPGAs. It includes a set of programming tools and libraries that allow developers to optimize performance, increase productivity, and reduce development time. In addition, the toolkit supports a variety of programming languages such as C++, Fortran, Python, and Data Parallel C++ (DPC++), an extension of C++ designed for heterogeneous computing.

Intel oneAPI for FPGAs uses a Board Support Package (BSP), which describes all hardware interfaces to the FPGA, like PCIe and DDR4 as well as provides a shell design for these for faster kernel integration and synthesis to the FPGA. Furthermore, Universal Shared Memory (USM) is available as BSP feature, which makes it possible to let the data transfers be managed by the FPGA itself. This functionality frees up the CPU from the data transfer management; it is only involved in creating input and output buffers on the CPU memory side and the FPGA firmware gets the data as needed by the kernels from the given input and output host pointers. The virtual-to-physical address translations for these data transfers are handled by the BSP part of the FPGA design. Thus, our developed solution is relying heavily on this USM BSP feature which is ideal for streaming applications.

Interesting key features of the DPC++ compiler for our solution idea include (i) its capability to synthesize arbitrary C++ code into circuits and (ii) the possibility to annotate an array as a register. Suppose a specific sequence of operations is executed on every element within such an array. In this case, the auto-vectorization feature of DPC++ (with the help of annotations) can detect data-parallel processing and will create circuits accordingly. Consequently, we can realize an FPGA SIMD backend in a simple and programmable way. Like any other SIMD extension, we require SIMD registers. Following the design decision of SIMD abstraction libraries with template metaprogramming, we define a templated C++ struct called *fpvec* as an FPGA SIMD register based on a regular array as shown in Figure 1. To support arbitrary SIMD register sizes, the base data type *BType* and the SIMD register size *RSize* are template parameters. Thus, the number of elements in an FPGA SIMD register depends on the size of *BType* and the SIMD register size *RSize*. As with ARM SVE, this approach allows the FPGA SIMD register size to be set dynamically at compile-time, providing a large variant space.

```
// BType = Data BaseType
// RSize = SIMD Register Size in Bytes
template<typename BType, int RSize>
struct fpvec {
    __attribute__((register)) std::array<BType, (RSize/sizeof(BType))> elements;
};
```

Figure 1: FPGA SIMD register template declaration.

In addition to SIMD registers, relevant SIMD primitives of the SIMD abstraction libraries must be provided by the FPGA backend. Since SIMD primitives operate only on SIMD registers, they can be implemented using a loop over the register type and execute the specific operation on

every register element. Figure 2 shows the implementation for two SIMD primitives, namely for the element-wise addition of two SIMD registers and the loading of data from main memory into a SIMD register. Both differ only in the concrete operation within the loop. To ensure that the DPC++ auto-vectorizer properly detects the SIMD opportunity, the loops are annotated with the preprocessor directive *pragma unroll*. This scheme can be applied to many SIMD primitives such as store, gather, scatter, and so on. Since we are building on the USM BSP feature, we can access the data regularly as shown in the implementation of the load SIMD-primitive.

```
template<typename BType, int RSize>
fpvec<BType,RSize> add(fpvec<BType, RSize>& a, fpvec<BType,RSize>& b) {
    auto reg = fpvec<BType, RSize>{}; // initialize result SIMD register
    #pragma unroll
    for (uint idx = 0; idx < RSize/sizeof(BType); idx++) {
        reg[idx] = a.elements[idx] + b.elements[idx];
    }
    return reg;     // return SIMD register with added data
}
```

(a) Element-wise addition (add) primitive.

```
template<typename BType, int RSize>
fpvec<BType,RegSize> loadu(BType* data) {
        auto reg = fpvec<BType,RSize>{};  // initialize result SIMD register
        #pragma unroll
        for (int idx=0; idx<(RSize/sizeof(BType)); idx++) {
                reg.elements[idx] = data[idx];
        }
        return reg;  // return register with loaded data
}
```

(b) Unaligned load primitive.

Figure 2: Exemplary implementation of two representative SIMD primitives.

However, this scheme is only sufficient for element-wise SIMD primitives, which are characterized by the fact that they do not introduce dependencies between the elements of the same SIMD register. That means, the corresponding operation is independently applied to every single element within a vector register. In contrast, horizontal SIMD primitives do not treat the elements of a register independently and thus, this scheme must be extended. An example is the horizontal reduction which sums up all elements within a vector register returning a single value. This horizontal reduction is typically realized using an adder tree where elements are added pairwise in a multi-stage process, and the results are again added pairwise until a single value is produced. The depth of such an adder tree equals $\log_2(N)$, where N is the number of elements. Consequently, if N is known at implementation time, the algorithm executes a fixed number of operations. However, since our FPGA SIMD backend has a variable, but compile-time constant register size, we had to come up with a size and type agnostic algorithm. Our algorithm (see Figure 3) consists of a nested loop, where the outer loop determines the number of executions, that will be carried out in the inner loop, starting with half of the element count and halving on each iteration. The inner loop adds adjacent values within the array and consecutively stores the result in the same array. For example, assuming a 16-element wide register, the outer loop would be executed four times. Eight adjacent pairs are added in the first iteration, and the results are stored in the lower half of the register. In the second iteration, the lower eight adjacent pairs are added and stored in the lower quarter of the register, and so on. Thus, the compiler will generate the corresponding adder tree as described previously.

```
template<typename BType, int RSize>
BType hadd(fpvec<BType, RSize>& a) {
    auto reg = fpvec<BType, RSize>{};
    #pragma unroll
    for(size_t idx = 0; idx < RSize/sizeof(BType); idx++) {
        reg[i] = a[i];
    }
    #pragma unroll cilog2(RSize/sizeof(BType))
    for(size_t current_width = ((RSize/sizeof(BType))>>1); current_width >= 1;
↪   current_width>>=1) {
        for(size_t i = 0; i < current_width; ++i) {
            reg[i] = result[(i<<1)] + result[(i<<1)+1];
        }
    }
    return reg[0];

}
```

Figure 3: Type and register size agnostic implementation of a horizontal reduction primitive (hadd).

Overall, it can be stated that Intel's oneAPI is a building block to realize a comprehensive FPGA SIMD backend in C++ without any knowledge of complex, FPGA-specific programming. We believe that even more complex SIMD primitives such as conflict detection - which was introduced by Intel with the SIMD instruction set extension AVX512 - can be implemented in a straightforward way. In our future work, we will examine this aspect even more closely.

3.1.2 Implementation and Execution

Accordingly, we extended our TVL/TSL library to support Intel FPGAs as described above. The complete TVL/TSL library with support for Intel FPAGs is available open source on GitHub: https://github.com/db-tu-dresden/TSL. For this deliverable, we prepared a comprehensive example to showcase that the same SIMDified kernel can achieve the peak bandwidth of the employed FPGA and switching between execution environments (CPU and FPGA) is simply done by providing a different template parameter. In more detail, we prepared a basic aggregation SIMD-kernel, e.g. scanning over the data and adding every element on the one hand. On the other hand, we implemented a filter-count SIMD-kernel, which counts the number of values in a given range. Figure 4 shows the SIMD-oblivious code for the filter-count SIMD kernel.

```
template<typename SIMDT, typename DataPtrT, typename DataT = Type>
uint32_t filter_kernel(DataPtrT data_ptr, size_t element_count, DataT lower, DataT upper) {
    using namespace tvl;
    // As the data is of type float we have to create a specific SIMD type which can carry out the simdified counting.
    using CoundSIMDT = tvl::simd<uint32_t, typename SIMDT::target_extension, SIMDT::vector_size_b()>;

    // Initialization of the result vector.
    auto result_vec = set1<CoundSIMDT>(0);
    // Creation of an increment vector which is used to increase the result vector whenever the corresponding value is in the given range.
    const auto increment_vec = set1<CoundSIMDT>(1);

    // Creation of the compare-vectors.
    const auto lower_vec = set1<SIMDT>(lower);
    const auto upper_vec = set1<SIMDT>(upper);
    for (size_t i = 0; i < element_count; i += SIMDT::vector_element_count()) {
        // Load (unaligned) data into a vector.
        const auto data_vec = loadu<SIMDT>(&data_ptr[i]);
        // Execution of the comparison.
        const auto result_mask = between_inclusive<SIMDT>(data_vec, lower_vec, upper_vec);

        // Creation of the increment vector of the current iteration (contains 0 for all elements which are not in the given range, 1 otherwise).
        const auto increment_result_vec = binary_and<CoundSIMDT>(reinterpret<SIMDT, CoundSIMDT>(to_vector<SIMDT>(result_mask)), increment_vec);
        // Increment of the result vector.
        result_vec = add<CoundSIMDT>(result_vec, increment_result_vec);
    }
    // Horizontally add all elements within the result vector to retrieve the final result.
    return hadd<CoundSIMDT>(result_vec);
}
```

Figure 4: Filter-Count SIMD-kernel

Both examples use various SIMD primitives of our TVL/TSL library such as the illustrated *loadu*, *add*, or *hadd*. Each of the required primitives can be effectively implemented for Intel FPGAs using our solution idea with a maximum of 13 LoC, without considering boilerplate code. Our approach allows to not only specify an implementation for a custom FPGA register (cf. Figure 1), but it also enables us to vary the underlying SIMD extension (e.g., SSE, AVX2, AVX512, "FPGA") or the employed FPGA register width on a per-invocation basis. Figure 5 showcases, how we can call the same SIMD-kernel with different register widths, and hence achive more concurrently processed elements, by varying only the third parameter of the *tvl::simd* struct.

```
using Type = float;
...
template <typename MyVec>
size_t agg_wrapper(queue& q, Type* in_host, long* out_host, size_t size) {...};
...
time[0] = agg_wrapper<tvl::simd<Type, tvl::fpga, 128>>(q, in, out + 0, elemCnt);
time[1] = agg_wrapper<tvl::simd<Type, tvl::fpga, 256>>(q, in, out + 1, elemCnt);
time[2] = agg_wrapper<tvl::simd<Type, tvl::fpga, 512>>(q, in, out + 2, elemCnt);
time[3] = agg_wrapper<tvl::simd<Type, tvl::fpga, 1024>>(q, in, out + 3, elemCnt);
time[4] = agg_wrapper<tvl::simd<Type, tvl::fpga, 2048>>(q, in, out + 4, elemCnt);
```

Figure 5: Dispatching the same kernel to the FPGA with different SIMD registers widths in bits.

Executing the examples: The sample code can be executed using Intel's DevCloud on real FPGA hardware, which is unfortunately necessary. The sample code is also available via the following GitHub project: https://github.com/db-tu-dresden/SiMoD23-SIMD-FPGA. In the following, we describe the entire execution workflow.

Execution using Intel's DevCloud: A free account on Intel's DevCloud for oneAPI can be created at the following website: https://devcloud.intel.com/oneapi/. Afterwards, the following steps must be carried out:

(1) Set up an SSH connection to Intel's DevCloud using

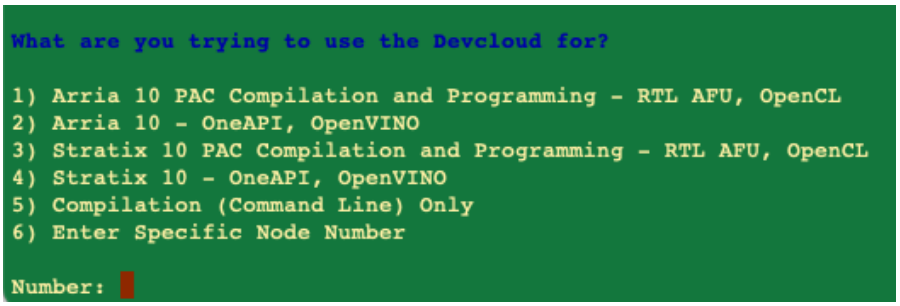
`1: ssh devcloud`

(2) Connect to an FPGA node

`2a: source /data/intel_fpga/devcloudLoginToolSetup.sh`

`2b: devcloud_login`

The following question appears:



`2c: Type in number 4 to use a Stratix 10 - OneAPI, OpenVINO node`

Then, an interactive session is set up on a corresponding node.

(3) Download the sample code

`3: git clone https://github.com/db-tu-dresden/SiMoD23-SIMD-FPGA`

(4) Compile the sample code

4a: Change in the corresponding folder

`cd SiMoD23-SIMD-FPGA`

4b: Setting up essential environment variables

`source /opt/intel/inteloneapi/setvars.sh – force`

4c: Initialize FPGA board

`aocl initialize acl0 pac_s10_usm`

4d: Compile executable code (takes several hours)

`make hw_agg` (compile aggregation example)

`make hw_filter` (compile filter-count example)

(5) Execute the sample code

`./build/eval_agg_kernel.fpga 1000000`

In this case, the aggregation is performed on 1 million random integer values.

`./build/eval_filter_kernel.fpga 1000000`

In this case, the filter-count kernel is performed on 1 million random integer values.

5.1.3 Evaluation

Based on the examples described above, we present inital evaluation results to demonstrate the applicability and efficiency of our developed solution.

Evaluation setup: For our evaluation, we used a dual-socket server with 3rd-generation Intel Xeon Scalable processors (code-named "Ice Lake"). The system is equipped with two Intel Xeon Platinum 8360Y processors, each having 36 cores with a base frequency of 2.2 GHz. The main memory consists of 16x DDR4 memory DIMMs with 32 GB each, which results in 512 GB of

memory per processor. As FPGA acceleration card a BittWare IA-840f card is used, which is equipped with an Intel Agilex 7 AGF027 FPGA and 4x 16 GB DDR4. The FPGA card is supporting PCIe gen4 with 16 lanes and can achieve a maximum bandwidth of approximately 16 GiBs in our setup. We ran all our experiments on 4 GiB of synthetically generated data and placed everything in the host memory being exposed as USM.

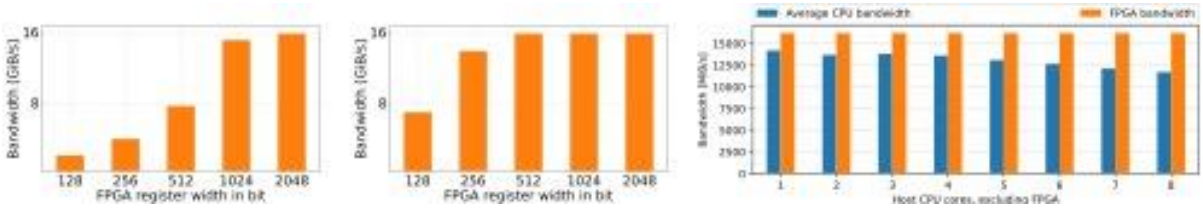


Figure 6: Evaluation Results.

FPGA Performance: Figures 6(a) and 6(b) show the achieved bandwidth for both kernels with register sizes matching those available to Intel SIMD registers of 512-bit and beyond. For the aggregation SIMD-kernel, we observe that the achieved bandwidth doubles for every doubling of the register width, until the global bandwidth threshold is reached. The filter-count SIMD-kernel achieves generally a higher bandwidth and reaches the limit earlier, at a register width of 512-bit, which matches the PCIe bus width. The aggregation SIMD-kernel faces a data dependency in the last stage of the adder tree, which is created from our current implementation of the *hadd* (cf. Figure 3) call. Here, the DPC++ compiler cannot properly pipeline the complete addition and stalls for at least a cycle. However, we can invest more resources of the FPGA and leverage a larger register, which is twice the size of the PCIe bus. This allows us to trade some FPGA resources for more overall bandwidth and finally to reach the available PCIe bus bandwidth.

Acceleration Evaluation: Our next experiment shows the achieved bandwidth for concurrent calculations on both the host CPU and the FPGA in Figure 6(c). We vary the number of concurrent threads on the host processor and additionally dispatch the same SIMD-kernel to the FPGA. The host code is executed using 512-bit sized AVX512 registers and the FPGA kernel is also parameterized to use 512-bit for its register width. For this experiment, we allocate 4 GiB of synthetically generated data per thread as well as FPGA and a dedicated compute thread is spawned for every core on the host as well as the FPGA. We synchronize the beginning of the computation of all threads, measure the wall clock time per thread and report the average bandwidth for all host threads and separately for the FPGA thread. As expected, more concurrently working threads on the host memory lead to a decreased per-thread bandwidth. However, also the simultaneously running FPGA can constantly achieve its peak performance. Thus, we are safe to conclude that it is possible to treat the FPGA card as an additionally available core running just the same host code.

Resource Utilization: Figure 7 shows the cumulated amount of required resources for all generated variants of the filter-count and aggregate kernels, i.e., the code for 128 through 2048 bit sized registers. Each of the best-performing kernel variants requires approximately 2% of the available Adaptive Logic Modules (ALM) and registers (REG). This implies that we could potentially integrate a larger number of different kernels into a single FPGA image, which can then be used to accelerate any required and available operator.

Component	ALM	REG	MLAB	RAM	DSP
Available	912800	3651200		13272	8528
Kernel System (Agg)	87897	211224	422	300	0
Kernel System (Filter)	89617	291260	431	300	144

Figure 7: Available resources on the Intel Agilex FPGA Device and aggregated utilization by all generated kernel variants.

3.1.3 SIMD-oblivious Kernels

While SIMD is a core technique for accelerating data processing, efficiently leveraging and integrating FPGAs is traditionally a complex task. However, the combination of SIMD abstraction libraries such as our TVL/TSL, Intel's oneAPI and the availability of USM allows us to seamlessly port standard templated C++ SIMD host code to the FPGA without the necessity of complex FPGA-specific programming. Based on our developed concept, we are able to implement arbitrary SIMD-oblivious kernels and enable compile-time deduction as well as code generation for a specific available SIMD extension as well as FPGA with usually negligible overhead for the runtime.

3.1.4 Integration into DAPHNE

To demonstrate the usability of our SIMD-oblivious concept, we implemented different relational query operators as kernels using our TVL/TSL library within the DAPHNE code base. The kernel sources can be found in "src/runtime/local/kernels/SIMDOperatorsDAPHNE". Moreover, the TVL/TSL library is included as thirdparty library into DAPHNE. In addition, we extended the compilation chain to map relational query operators to these SIMD-oblivious kernels. To show the entire integration in terms of mapping and execution, we prepared a simple example using query Q1.1 of the Star-Schema Bechmark (SSB). To execute the query with the utilization of the SIMD-oblivious kernels, the following steps are necessary.

1. Build daphne
   `./build.sh --scalar` (for compatibility reason, we restrict to a scalar execution which corresponds to a SIMD register width of one element)
2. Switch to data_generation folder
3. Generate the SSB data for scale factor 1
   `./data.gen.sh -sf 1`
4. Switch back to daphne folder and execute SSB query Q1.1

```
bin/daphne --columnar --vector_extension=SCALAR ./scripts/evaluation/ssb-Q1-
1-SF1.daph
```

The option "vector_extension" can be used to define the SIMD extension to be used. For this, DAPHNE must also be built with the corresponding option. In the current state, the mapping to FPGA is not possible, as the memory management still needs to be extended in this respect.

3.2 GPU Example

In this part of the deliverable, we give a compact overview of how our efforts regarding code generation for sparsity exploitation on GPU are improving end-to-end performance in an example algorithm from our collection of supported algorithms available in the DAPHNE source repository.

3.2.1 CUDA Code Generation

Generating and compiling source code is a technique commonly employed in modern data management, machine learning and high-performance computing systems and frameworks. While some of them focus on providing executables for new hardware platforms or leveraging features of commonly supported systems (e.g., special SIMD instructions), most of them aim to optimize execution in one way or another. This can go into the directions of operator fusion, query compilation, loop fusion, tiling and sparsity exploitation. Of course, a mix of several of these techniques are often combined [B+18, E+17], as is the case in our approach as well. While existing implementations use Java and run on CPUs, we will bring operator fusion and sparsity exploitation to CUDA based GPU platforms to improve the efficiency by leveraging larger memory bandwidth on GPUs and keeping data locally on the GPU.

3.2.2 Code Templates

Operator fusion brings many advantages such as avoiding allocation of intermediates, reduced memory bandwidth requirements, and specialization according to input operations provided by our DaphneDSL. Combining this with the support for sparse data formats allows us to exploit sparsity across chains of operations, which makes avoiding intermediates (often becoming dense and therefore huge in size) even more effective.
Our entry point into the vast space of fusion opportunities when taking all their permutations into account (e.g., if there are operators A,B and C, where A and B could be fused, B and C could be fused but not all three at once) is the type of data access that operations require. Out

of the four patterns from the CPU bound code generation described in [B+18], we have implemented two for now, which we will show in more detail in this demonstrator.

Cell-based template: The cell-based code template refers to a class implemented in CUDA-compatible C++ with a number of anchor points (textual placeholders that are replaced during code generation) and CUDA-kernel definitions that instantiate this class appropriately. During execution an "exec_dense()" or "exec_sparse()" method is called for every data item of the input (usually a matrix) depending on the format being plain dense or compressed sparse row (CSR). This computation for every cell of the input data can also be employed in an aggregation, combining, for example, some computation for every item with a sum over all items to be processed.

Row-wise template: This code template shares the same basic software architecture as the cell-based one but is geared towards a row-by-row computation. Furthermore, his template allows for operations that need to access the entire row (potentially combined with cellwise operations and final aggregation). By having this distinction between cell and row-based computation, threads can be launched in a more suitable way and more optimized primitives can be invoked if the by-row relationship of the input can be expected.

3.2.3 Implementation Details

For the demonstration in this deliverable, we have ported functionality from the code base where development for this method has initially been started. While the initial implementation in [BA+20] contains sophisticated DAG analysis, enumeration, and cost-based selection of fusion plans, we focused on a working example of the generation and execution parts of our solution on GPU in DAPHNE. To this end, the DAPHNE compiler has been extended to detect the patterns used in our example, replace them with an MLIR representation of our generated operators and rewire the inputs and outputs accordingly. The code templates used are hard-coded for now and compiled via CUDA's nvrtc – the run-time compiler. This will produce another intermediate representation, namely PTX, which is an IR used in Nvidia's compiler stack. When launching a generated operator, the final compilation step is performed by the NVIDIA driver.

3.2.4 Practical Example

We will use the Connected Components algorithm, which can be found in the DaphneDSL script collection of the project's source tree as well as in code listing 1 below, as an example to demonstrate how our code generation method spawns fused operators to optimize the execution in terms of memory requirements and run-time.
The algorithm takes an adjacency matrix as input, where each row and column represent nodes in a graph and the non-zero values of the matrix represent edges. The row numbers serve as node IDs. The output is a vector with number-of-nodes rows where each value represents the highest ID the node is connected to. For example, if the output was [1, 3, 3, 4] we see two nodes that have no connections (node 1 and 4) and one connection between node 2 and 3, where the highest ID 3 is shown for node 2 and 3 in the output vector.

```
01: // Arguments:
02: // - f ... filename of the adjacency matrix (provide as `--args f=\"foo.csv\"`)
03:
04: t0 = now();
05:
06: // Read adjacency matrix.
07: G = readMatrix($f);
08:
09: // Initialization.
10: // Maximum number of iterations (to stop if diff never reaches zero)
11: maxi = 1000;
12: c = seq(1.0, as.f64(nrow(G)), 1.0); // init w/ vertex IDs
13: diff = inf;
14: iter = 1;
15:
16: t1 = now();
17: // Iterative computation of connected components (decisive part).
18: while(as.si64(diff > 0.0) && iter <= maxi) {
19:     ti0 = now();
20:
21:     u = max(aggMax(G * t(c), 0), c);
22:     diff = sum(u != c);
23:     c = u;
24:
25:     ti1 = now();
26:     print("iteration ", 0, 1);
27:     print(iter, 0, 1);
28:     print(" took [ns]: ", 0, 1);
29:     print(ti1 - ti0);
30:
31:     iter = iter + 1;
32: }
33:
34: t2 = now();
35: // Print elapsed times in nano seconds.
36: print(t1 - t0); // initialization
```

```
37: print(t2 - t1); // core algorithm
38: // Note that, for distributed execution, (t2 - t1) includes reading the input
39: // files due to some reordering done by the compiler.
40:
41: // Result output.
42: print(c);
```

Code Listing 1: Connected Components in DaphneDSL

```
01: // … code omitted for brevity
02: %38 = "daphne.transpose"(%arg1) : (!daphne.Matrix<?x1xf64>) -> !daphne.Matrix<1x?xf64>
03:     %39     =     "daphne.ewMul"(%21,     %38)     :     (!daphne.Matrix<17500x17500xf64>,
!daphne.Matrix<1x?xf64>) -> !daphne.Matrix<?x?xf64>
04: %40 = "daphne.maxRow"(%39) : (!daphne.Matrix<?x?xf64>) -> !daphne.Matrix<?x1xf64>
05: %41 = "daphne.ewMax"(%40, %arg1) : (!daphne.Matrix<?x1xf64>, !daphne.Matrix<?x1xf64>) -
> !daphne.Matrix<?x1xf64>
06: %42 = "daphne.ewNeq"(%41, %arg1) : (!daphne.Matrix<?x1xf64>, !daphne.Matrix<?x1xf64>) -
> !daphne.Matrix<?x1xf64>
07: %43 = "daphne.sumAll"(%42) : (!daphne.Matrix<?x1xf64>) -> f64
08: // … code omitted for brevity
```

Code Listing 2: Relevant DaphneIR before fusing operations

```
01: // … code omitted for brevity
02: %40 = "daphne.codegenRW"(%21, %arg1, %39) {operand_segment_sizes = array<i32: 3, 0>} :
(!daphne.Matrix<1000x1000xf64>, !daphne.Matrix<?x1xf64>, ui64) -> !daphne.Matrix<?x1xf64>
03:    %41    =    "daphne.constant"()    {value    =    1    :    ui64}    :    ()    ->    ui64
04: %42 = "daphne.codegenCW"(%40, %arg1, %41) {operand_segment_sizes = array<i32: 3, 0>} :
(!daphne.Matrix<?x1xf64>, !daphne.Matrix<?x1xf64>, ui64) -> f64
05: // … code omitted for brevity
```

Code Listing 3: Relevant DaphneIR after fusing operations

In Code Listings 2 and 3, we see the DaphneIR as it went through the code generation compiler passes. The former shows the original instructions spawned by the DAPHNE compiler. The latter shows where instructions have been replaced and operands/results have been rewired.

The diagram in Figure 8 shows the relevant operations of the first fused operator and sketches of input and output data sizes (scalars, vectors and matrices). The blue boxes are data items existing in the algorithm while the red boxes show intermediate states that emerge between operations.

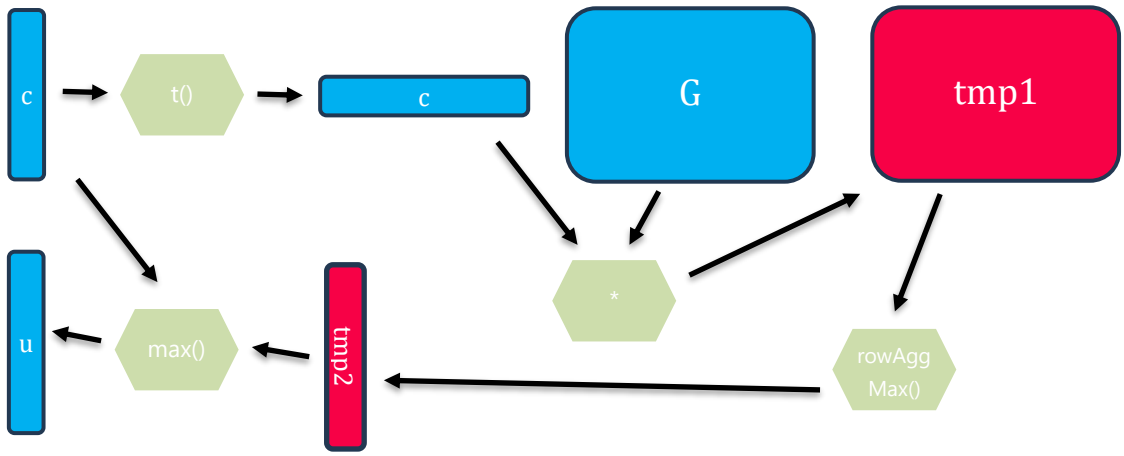


Figure 8: First fusion opportunity showing unnecessary intermediates (from line 21 in Code Listing 1)

The four green hexagons resemble the four operations that will be fused into one row-wise operator. The transpose t() is a no-op, which means it does not perform any computation or data movement. As the memory representation for row or column vectors would be the same, this operation merely swaps the row/column dimensions. The next operation is an element-wise multiply of the transposed input **c** and the potentially sparse matrix **G**. The result is stored in a temporary intermediate (**tmp1**) that has the same dimensions as **G**. Subsequently, the rows of **tmp1** are aggregated to find the maximum value of each row. These max values are compared against the input vector **c** in the last max() operation to finally store the output in u. The generated row-wise operator will process row-by-row to aggregate the max value of the **G*c** multiplication. The result of this multiplication is compared to the value of input vector **c** at the corresponding index. The larger value of this comparison (max()) is stored in the output **u**. All of this happens in one go without storing the intermediate values to main memory in between each step.

Figure 9 shows the schematic of the next two operations in line 22 of the input script. Here we can avoid saving the output of the **c != u** operation in main memory just to load it again right away to sum it all up to form the output "**diff**". This fused operation works from a cell-based aggregation code template to do the inequality (!=) and sums up the results up in one go. This behavior could also be achieved in row-wise processing. The reason why the cell-based solution has been taken in favor of the row-wise approach has to do with how threads are spawned to do the processing, which is more efficient when done cell-based.

This practical example can be tested with the provided downloadable artifact (see section 2). Please refer to the read-me file within the Zip archive for instructions how to setup and run the demonstrator.

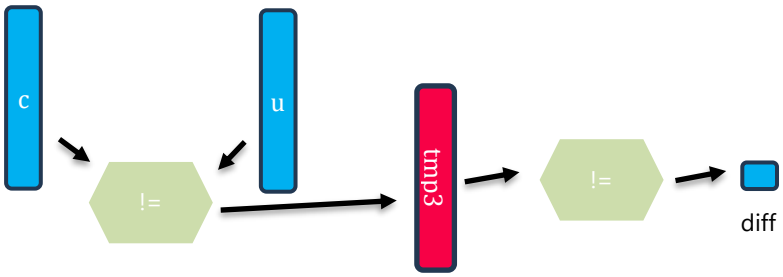


Figure 9: Second fusion opportunity (from line 22 in Code Listing 1)

3.2.5 MLIR Based Code Generation

Besides the code generation technique described above, we also pursue a different approach in DAPHNE. Since our compiler is based on MLIR, it quite naturally asks for trying to directly compile down from the DAPHNE dialect to LLVM without the use of precompiled kernels, which is our default solution. This effort is in its initial stages and focuses on CPU execution for now but might also lend itself to experimenting with other compilation targets supported by the MLIR/LLVM compiler stack. While this work is not featured in this deliverable, a first pull request of considerable size has been merged into the main branch of DAPHNE [DGHI+633] providing a first working preview.

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)
- **D7.3/** (scripts for running the GPU example)
- **lib/** (generated kernel libraries)
- **scripts/** (DaphneDSL scripts as examples)
  - **evaluation/** (Deliverable specific examples)
- **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)
    - **SIMDOperatorsDAPHNE/** (kernels using SIMD operators)
- **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 https://daphne-eu.eu/wp-content/uploads/2021/11/Deliverable-2.1-fin.pdf

[D2.2] DAPHNE: D2.2 Refined System Architecture https://daphne-eu.eu/wp-content/uploads/2022/08/D2.2-Refined-System-Architecture.pdf

[D3.1] DAPHNE: D3.1 Language Design Specification

[D5.1] DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks https://daphne-eu.eu/wp-content/uploads/2021/11/Deliverable-5.1-fin.pdf

[D7.1] DAPHNE: D7.1 Design of integration HW accelerators https://daphne-eu.eu/wp-content/uploads/2022/05/DAPHNE_Deliverable_7.1.pdf

[D7.2] DAPHNE: D7.2 Prototype and overview HW accelerator support and performance models https://daphne-eu.eu/wp-content/uploads/2022/12/D7.2-Prototype-and-Overview-HW-Accelerator-Support-and-Performance-Models.pdf

[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

[E+17] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen: "SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning", CIDR, 2017

[H+23] Dirk Habich, Alexander Krause, Johannes Pietrzyk, Christian Faerber, Wolfgang Lehner: Simplicity done right for SIMDified query processing on CPU and FPGA. SiMoD@SIGMOD 2023: 3:1-3:5