D7.2 Prototype and Overview HW Accelerator Support and Performance Models

DAPHNE

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

Version 1.5

PUBLIC

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

Document Description

This document describes a snapshot of our DAPHNE prototype regarding HW acceleration as a follow-up of deliverable D7.1. In particular, we introduce three example scenarios as the main drivers for the current prototype and describe how to execute these examples with enhanced hardware support. Additionally, we give an overview report on the devised performance models for a cost-based approach for kernel and data placement decisions in a heterogeneous hardware environment in this document.

D7.2 Prototype and Overview HW Accelerator Support and Performance Models			
WP7 – HW Accelerator Integration			
Type of document	R	Version	1.5
Dissemination level	PU	Project month	24
Lead partner	TUD		
Author(s)	Dirk Habich (TUD), Eric Mier (TUD), Mark Dokter (KNOW), Piotr Ratuszniak (INTP)		
Reviewer(s)	Tilmann Rabl (HPI), Patrick Damme (KNOW)		
Contributors	all		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial structure and write-up	Dirk Habich, Eric Mier
V1.1	Content to GPU and FPGA prototype	Mark Dokter, Piotr Ratuszniak
V1.2	Refinement	Dirk Habich
V1.3	Reviewer feedback incorporated	Dirk Habich
V1.4	GPU example revision, adaptations related to artifact content	Mark Dokter
V1.5	Final version	Dirk Habich

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 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 though. 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 it has detailed on accelerated operations and primitives.

As introduced in the DAPHNE report D7.1 [D7.1], the challenges for the integration of HW accelerators are (i) developing as well as generating operators - hereinafter also called computation kernels or kernels for short - which can be efficiently executed on accelerators such as CPUs, GPUs or FPGAs, (ii) integrating these accelerator-specific operators in the whole DAPHNE compilation and runtime infrastructure in a seamless way, and (iii) selecting the best-fitting accelerator for efficient execution depending on the specific IDA pipeline and hardware environment [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*. For the current implementation in the DAPHNE prototype, we have mainly focused on the challenges (i) and (ii) in order to be able to develop hardware-accelerated kernels and to anchor these hardware-accelerated kernels in the entire DAPHNE infrastructure. Thus, we describe the current state of our prototype in this document. Additionally, we give an overview report on the devised performance models for a cost-based approach for hardware-accelerated kernels and data placement decisions in a heterogeneous hardware environment.

The remainder of this deliverable is structured as follows:

- In Section 2, we detail the access to our prototype artifact for this written deliverable document.
- Then, we introduce our prototype by describing the underlying demonstration scenarios in Section 3.
- Afterwards, we explain the folder structure of our prototype in Section 4.
- Section 5 presents an overview of our devised performance models.

2 Artifact Access

The DAPHNE prototype for the integration of HW accelerators that is described in this deliverable is publicly accessible as a snapshot of the DAPHNE development repository (created in November 29 2022) under the following link:

Link: https://tinyurl.com/daphne-D72 (88.3 MB)

Note that the DAPHNE development repository is publicly available at https://github.com/daphne-eu/daphne under Apache License v2.0.

3 Demonstration Scenario

In this deliverable, we present three different scenarios, each focusing on a different hardware accelerator type.

- [SIMD-Example]: The first scenario accelerates an analytical relational SQL query of an example IDA pipeline with the usage of Single-Instruction Multiple-Data (SIMD) extensions of general-purpose CPUs.
- [GPU-Example]: The second scenario showcases the support of GPU for a linear regression algorithm which is part of the example IDA pipeline from the SIMD-Example. The example was run on an Nvidia Tesla T4 with 16GB RAM but should run on any modern Nvidia GPU with more than 4GB of RAM.

- [FPGA-Example]: The third scenario presents how to accelerate example linear algebra kernels, like SGEMM and SGEMV on an FPGA based accelerator. At the current project stage the Intel Programmable Acceleration Card D5005 has been used[1].

3.1 General Initial Setup

Step 0.1 Install Dependencies: Setup a Linux environment (tested with Ubuntu 20.04), and install the dependency versions specified in docs/GettingStarted.md, which includes clang, cmake, git, lld, ninja, pkg-config, python3, numpy, openjdk, gfortran, uuid-dev, libboost-dev, and wget.

Step 0.2 Download and Extract: Download the artifact from the link in Section 2, and extract it as follows into a directory called daphne-D7.2. The provided binaries were compiled on an Ubuntu 18.04 system (for compatibility with the FPGA code paths) with a toolchain compatible with Ubuntu 20.04 (our default DAPHNE platform using gcc 9.4 and llvm 10.0), using an x86-64 machine (Intel Xeon).

```
$ tar xf daphne-D7.2.tgz

$ cd daphne-D7.2

$ tar xf daphne--cuda-D7.2-bin.tgz
$ tar xf daphne--fpgaopencl-D7.2-bin.tgz
$ tar xf daphne--morphstore-D7.2-bin.tgz
$ tar xf daphne-D7.2-source.tgz
```

Step 0.3 Set some paths: To make it easier to find the correct paths throughout the examples, we store some paths into variables.

```
$ artifact=$(pwd)

$ source=$artifact/daphne-D7.2-source
```

3.2 SIMD-Example

The following example uses a part of the presented pipeline P1 in [D+22], more precisely the querying part. Fundamentally, pipeline P1 consists of a TPC-H inspired relational query followed by linear regression model training on the query result. This relational SQL query computes the total price of all purchases each customer did, filtered by a range of market segments after a certain date. The DaphneDSL-script to process this query looks like this (Figure 1):

[1] https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac/d5005.html

```
/// read input
c = readFrame($inCustomer);
o = readFrame($inOrders);

/// register frames to sql scope
registerView("customer", c);
registerView("orders", o);

/// execute query
res = sql(
  "Select customer.C_CUSTKEY, sum(orders.O_TOTALPRICE)
   From customer
   Inner Join orders
   On customer.C_CUSTKEY = orders.O_CUSTKEY

   Where customer.C_MKTSEGMENT <= 2
   And orders.O_ORDERDATE >= 19960802
   Group By customer.C_CUSTKEY;"
);

// Range of Market_Segment: 0 - 4
// Range of Orderdate: 19920101 - 19980802


/// print the results
print("================================");
print(res);
print("================================");
```

Figure 1: Analytical SQL Query using TPC-H data in DAPHNE

Description: The input is read from two CSV-files that are provided via runtime parameters. The data is stored in two frames, which have to be registered with a label to make them visible to the SQL parser. Next, the query is executed and the result is stored in a new frame, which is printed to the console in the last step.

Step 1.1 Build DAPHNE (optional): Then build the prototype, its dependencies, and the parser as follows from within the daphne directory (if it fails run "./build.sh --clean" for a clean start). On the first run this step might take up to 60min.

```
$ cd $source

$ ./build.sh
```

If you wish to skip building yourself, the artifact contains prebuilt binaries for each demonstration scenario. Just skip the commands above and navigate into directory of the first scenario:

```
$ cd $artifact/daphne--morphstore-D7.2-bin
```

Step 1.2 Generate TPC-H data: For this scenario, we require the tables 'customer' and 'orders' of the TPC-H data. For this the dependency 'realpath' is needed.

```
$ ./benchmarks/tpc-h/generate.sh
```

Step 1.3 Alias to DAPHNE compiler (optional): To shorten things, one could create an alias to the executable of the DAPHNE compiler.

```
$ alias daphne=./bin/daphne
```

If this step is skipped, the daphne calls in following steps have to be replaced by './bin/daphne'. If you wish to use the prebuilt binary,

Step 1.4 Set some paths: The following paths are needed multiple times, so store them in variables. (We assume we are still in the DAPHNE root directory)

```
$ script=$(pwd)/scripts/deliverables/del_7_2_example.daphne

$ customer=inCustomer=\"$(pwd)/benchmarks/tpc-h/data/customer.csv\"

$ orders=inOrders=\"$(pwd)/benchmarks/tpc-h/data/orders.csv\"
```

Step 1.5 Build DAPHNE with the SIMD framework MorphStore: This framework enables the use of kernels that exploit vector operations through the Template SIMD Library (TSL) [DU+20, U+20]. Fundamentally, MorphStore is an in-memory columnar analytical query engine with a novel holistic compression-enable and highly-vectorized processing model [DU+20]. If you choose to use the prebuilt binaries, skip this step.

```
$ ./build.sh --morphstore
```

This should not take as long as the first build, since only some additional components have to be compiled.

Step 1.6 Execute the Scenario without SIMD extension: The example script works also without any API extensions.

```
$ daphne $script $customer $orders
```

Step 1.7 Execute with SIMD extension: Now run the script again using MorphStore operations [DU+20].

```
$ daphne --api MorphStore $script $customer $orders
```

The api flag sets the API-name used by our developed *KernelSelectionPass* to map element-wise compare operations (==,!=,>,>=,<,<=) to MorphStore kernels. This *KernelSelectionPass* is our central building block in the DAPHNE system for mapping kernels to hardware-specific kernels and is implemented within the DAPHNE compiler. The mapping is currently determined by specific flags, which will later be automated using a cost-based selection technique. The results of both execution variants should be the same.

Step 1.8 Print the DaphneIR: The key difference is shown in the IR after the kernel calls are generated. For this, DAPHNE offers an explain feature for different levels of the compiler lowering. We take a look at the 'sql' and 'kernels' level:

SQL:

Default:

```
$ daphne --explain sql $script $customer $orders
```

Output:

```
%18 = "daphne.ewLe"(%16, %17) :
(!daphne.Matrix<?x?x!daphne.Unknown>, si64) ->
!daphne.Matrix<?x?x!daphne.Unknown>

%23 = "daphne.ewGe"(%21, %22) :
(!daphne.Matrix<?x?x!daphne.Unknown>, si64) ->
!daphne.Matrix<?x?x!daphne.Unknown>
```

SIMD:

```
$ daphne --api MorphStore --explain sql $script $customer $orders
```

Output:

```
%18 = "daphne.ewLe"(%16, %17) :
(!daphne.Matrix<?x?x!daphne.Unknown>, si64) ->
!daphne.Matrix<?x?x!daphne.Unknown>

%23 = "daphne.ewGe"(%21, %22) :
(!daphne.Matrix<?x?x!daphne.Unknown>, si64) ->
!daphne.Matrix<?x?x!daphne.Unknown>
```

The Output shows relevant parts of the DaphneIR after executing with the explain feature. The DaphneIR after parsing the SQL query is exactly the same. But after lowering to kernel level we see that the *'ewLe'* and *'ewGe'* operations are mapped to different kernel calls.

Kernels:

Default:

```
$ daphne --explain kernels $script $customer $orders
```

Output:

```
%27 = "daphne.call_kernel"(%26, %10, %19) {callee =
"_ewLe__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t"} :
(!daphne.Matrix<?x1xsi64>, si64, !daphne.DaphneContext) ->
!daphne.Matrix<?x1xsi64>

%30 = "daphne.call_kernel"(%29, %12, %19) {callee =
"_ewGe__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t"} :
(!daphne.Matrix<?x1xsi64>, si64, !daphne.DaphneContext) ->
!daphne.Matrix<?x1xsi64>
```

SIMD:

```
$ daphne --api MorphStore --explain kernels $script $customer
$orders
```

Output:

```
%27 = "daphne.call_kernel"(%26, %10, %19) {callee =
"MorphStore_ewLe__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t"
} : (!daphne.Matrix<?x1xsi64>, si64, !daphne.DaphneContext) ->
!daphne.Matrix<?x1xsi64>

%30 = "daphne.call_kernel"(%29, %12, %19) {callee =
"MorphStore_ewGe__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t"
} : (!daphne.Matrix<?x1xsi64>, si64, !daphne.DaphneContext) ->
!daphne.Matrix<?x1xsi64>
```

The default version generates the kernel call '`_ewLe__DenseMatrix...`', which is associated with a DAPHNE native kernel for scalar CPU instructions, but for SIMD the call '`MorphStore_ewLe__DenseMatrix_`' is associated with another kernel leveraging MorphStore operations.

Step 1.9 Print the DaphneIR colorized (optional): To better visualize the differences, the previous commands can be extended like this:

```
$ daphne --explain sql $script $customer $orders 2>&1 | grep -E --
color "ewGe|ewLe|MorphStore|"

$ daphne --api MorphStore --explain sql $script $customer $orders
2>&1 | grep -E --color "ewGe|ewLe|MorphStore|"


$ daphne --explain kernels $script $customer $orders 2>&1 | grep -E
--color "_ewGe_|_ewLe_|MorphStore|"

$ daphne --api MorphStore --explain kernels $script $customer
$orders 2>&1 | grep -E --color "_ewGe_|_ewLe_|MorphStore|"
```

The std::error is piped to std::out because the explain functionality prints to std::error and otherwise is not grepped. To only print the lines containing the search terms, remove the last pipe character in the search string.

3.3 GPU-Example

The second part of the demonstrator D7.2 showcases the support of GPU operations in DAPHNE (second example). We continue the example of pipeline P1 from our CIDR 2022 paper [D+22] with the modification that we omit the SQL query part and use synthetically generated input data. We run the subsequent linear regression algorithm (LM) using the direct solve method. The data loading stays on the host side (CPU execution) and reads pre-generated input from a file (a matrix stored in DAPHNE binary format), while the LM part runs on the device. Currently, the only supported API for GPU operations is CUDA and therefore we restrict ourselves to Nvidia hardware. DAPHNE has been tested on the Pascal, Turing, and Ampere architecture so far. Support for Intel GPUs (via the OneAPI) is in an early proof of concept stage with code in a pending pull request on our Github repository while AMD support remains an open issue and will be dealt with in future work. This should cover all relevant GPU platforms currently available.

In Figure 2.1 we see the core of the linear regression algorithm using direct solve (lmDS). The full script can be found in `scripts/deliverables/lm.daph` in the deliverable artifact.

```
# … load, extract, normalize
# LM pipeline
t0_lm=now();
A = syrk(X);
lambda = fill(as.f32(0.001), ncol(X), 1);
A = A + diagMatrix(lambda);
b=gemv(X,y);
beta = solve(A, b);
t1_lm=now();
#...print timing
```

```
# Print the sum of the result vector for checking
print("Result sum(beta): ", 0); print(sum(beta));
```

Figure 2.1: Reduced code snipped of the linear regression algorithm

Step 2.0 Building(optional): To activate CUDA support in DAPHNE, a simple flag to the build-script is enough (besides having set up the CUDA SDK and the common third party prerequisites). This step can be run from the bundled source code snapshot (daphne-D7.2-source directory). If this optional step is taken, please also run all subsequent steps from the source directory where you compiled DAPHNE.

```
$ cd $source
$ build.sh --cuda
```

Step 2.1 Environment Setup: To enable the operating system's shared library loading mechanism to find the local library files used by DAPHNE, we need to set an environment variable. We start from the deliverable extraction directory `daphne-D7.2`:

```
$ cd $artifact/daphne--cuda-D7.2-bin
$ export LD_LIBRARY_PATH=$PWD/lib:$LD_LIBRARY_PATH
```

Step 2.2 Preparation: The input data is pre-generated by a DaphneDSL script and saved to a DBDF (DAPHNE binary format) file on disk. The invocations are wrapped in a shell script. By default, the generated data size is 1907 MB (500000 rows and 1000 columns of fp32 values (4 bytes)).

```
$ deliverables/7.2/setup-gpu.sh
```

Step 2.3 Execution: The runtime activation of the CUDA API uses the same flag like the build script "--cuda" (this flag enables the user to run without GPU ops even if support for that is compiled into the binary). The following convenience script takes care of launching the linear regression example in three different configurations. First running with CPU operations only, then vectorized CPU ops and lastly the run with CUDA ops activated. The output of the script sum(beta) indicates that the three invocations produce the same result while the printed timing documents the benefit of running our vectorized engine or with CUDA ops. The combination of CUDA operations with the vectorized engine is work in progress.

```
$ deliverables/7.2/run-gpu.sh

LM on CPU

Execution time: 11.2386 seconds

Result sum(beta): 0.473498

LM vectorized CPU
```

```
Execution time: 7.13675 seconds

Result sum(beta): 0.473498

LM on CUDA

Execution time: 5.91564 seconds

Result sum(beta): 0.473495
```

For reference, this example was run on a dual Xeon Gold 6238R (112 vcores) machine with 768GB RAM and an Nvidia Tesla T4 (16GB VRAM).
In Figure 2 we show the operations that were compiled down to their respective CUDA version by the DAPHNE compiler.

```
"daphne.call_kernel"(%15) {callee = "CUDA_createCUDAContext"} : (!daphne.DaphneContext) -> ()
%28 = "daphne.call_kernel"(%21, %26, %15) {callee =
         "CUDA_ewSub__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<500000x?xf32>, !daphne.Matrix<1x?xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x?xf32>
%29 = "daphne.call_kernel"(%28, %27, %15) {callee =
         "CUDA_ewDiv__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<?x?xf32>, !daphne.Matrix<1x?xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x?xf32>
%32 = "daphne.call_kernel"(%29, %31, %15) {callee =
         "CUDA_colBind__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<?x?xf32>, !daphne.Matrix<?x1xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x?xf32>
%35 = "daphne.call_kernel"(%32, %15) {callee =
        "CUDA_syrk__DenseMatrix_float__DenseMatrix_float"} : (!daphne.Matrix<?x?xf32>,
         !daphne.DaphneContext) -> !daphne.Matrix<?x?xf32>
%39 = "daphne.call_kernel"(%35, %38, %15) {callee =
         "CUDA_ewAdd__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<?x?xf32>, !daphne.Matrix<?x?xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x?xf32>
%40 = "daphne.call_kernel"(%32, %23, %15) {callee =
         "CUDA_gemv__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<?x?xf32>, !daphne.Matrix<500000x?xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x1xf32>
%41 = "daphne.call_kernel"(%39, %40, %15) {callee =
         "CUDA_solve__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float"} :
         (!daphne.Matrix<?x?xf32>, !daphne.Matrix<?x1xf32>, !daphne.DaphneContext) ->
         !daphne.Matrix<?x1xf32>
```

Figure 2: Operations converted to CUDA ops in the linear regression algorithm

3.4 FPGA-Example

The third scenario for the demonstrator D7.2 presents how to accelerate example linear algebra kernels, like SGEMM (matrix-matrix operation) and SGEMV (matrix-vector operation) on an FPGA based accelerator.

Prerequisites:

Integrated example linear algebra kernels require an installed FPGA device with configured OpenCL support. The support can be provided by Intel® FPGA OpenCL SDK[2] or by Intel® oneAPI with FPGA Add-On toolkit[3]. Additional configuration details are described in Daphne prototype documentation[4]. Example integrated FPGA based kernels have been developed using T2SP [S+19] tool from Intel Labs.

Step 3.1 Build DAPHNE with FPGA openCL kernels support: To build the prototype with additional support for example FPGA linear algebra kernels the build script requires an additional flag "–fpgaopencl". The following example presents the complete build command:

```
$ cd $source

$ ./build.sh --fpgaopencl
```

For a proper compilation process additional system variables must be defined. Required variables can be defined using the following example system commands:

```
$ export QUARTUSDIR=/opt/intel/intelFPGA_pro/21.4

$ source $QUARTUSDIR/hld/init_opencl.sh
```

Step 3.2 FPGA programming:

Additional commands must be executed to program an FPGA device using a compiled FPGA image and the programmed image must be described by the BITSTREAM variable. For example, the following command can used for FPGA SGEMM:

```
$ aocl program acl0 bitstreams/sgemm.aocx

$ export BITSTREAM=bitstreams/sgemm.aocx
```

Used, precompiled FPGA images can be downloaded using the following address: https://github.com/daphne-eu/supplemental-binaries/tree/main/fpga_bitstreams.

Step 3.3 Daphne DSL scripts for FPGA kernels demonstration:

For the FPGA-based SGEMM kernel execution, the following script has been used:

[2] https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.htm
[3] https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html#gs.itep4
[4] https://github.com/daphne-eu/daphne/blob/main/doc/FPGAconfiguration.md

```
01: # Creating input matrices
02: m  = rand(448,1024, as.f32(1.0), as.f32(2.0), 1.0, -1);
03: m2 = rand(1024,512, as.f32(1.0), as.f32(1.0), 1.0, -1);
04:
05: # Example input matrices values prints
06: print(m[0:5,0:5]);
07: print(m2[0:5,0:5]);
08:
09: # Matrix multiplication operation
10: Z = m @ m2;
11:
12: # Example output values print
13: print(Z[0:5,0:5]);
14:
15: print("Bye!");
```

Figure 3: SGEMM - matrix-matrix multiplication execution

Inside the script (Figure 3), two input matrices have been created, where the first matrix contains random values from range <1.0,2.0> and the second matrix contains the same value 1.0 for all elements. On line 10, the matrix multiplication kernel is executed and on line 13 output results from matrix Z are printed on the screen. To run the example DaphneDSL script the following command has been used:

```
$ /bin/daphne --fpgaopencl scripts/examples/fpga-sgemm.daph
```

An example result for the script execution is presented below:

```
DenseMatrix(5x5, float)
1.45519 1.17568 1.38401 1.46265 1.37441
1.22417 1.10624 1.39034 1.53821 1.94647
1.60796 1.32805 1.45487 1.16758 1.2184
1.41087 1.16551 1.52658 1.52007 1.47633
1.21382 1.39956 1.97165 1.27372 1.24413
DenseMatrix(5x5, float)
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
DenseMatrix(5x5, float)
1544.36 1544.36 1544.36 1544.36 1544.36
1532.02 1532.02 1532.02 1532.02 1532.02
1547.05 1547.05 1547.05 1547.05 1547.05
1542.2 1542.2 1542.2 1542.2 1542.2
```

Based on input matrix sizes, input data values the presented output presents correct values from range <1024,2048>.

The following second script is used for the FPGA SGEMV kernel.

```
# Creating input matrices
m  = rand(2048,1024, as.f32(1.0), as.f32(2.0), 1.0, -1);
m2 = rand(1024,1, as.f32(1.0), as.f32(1.0), 1.0, -1);

# Example input matrices values prints
print(m[0:5,0:5]);
print(m2[0:5,0:5]);

# Matrix vector multiplication operation
Z = m @ m2;

# Example output values print
print(Z[0:5,0:5]);

print("Bye!");
```

Figure 4: SGEMV - matrix-vector multiplication example

The presented DaphneDSL script (Figure 4) contains similar first input matrix creation but second is a vector. The FPGA GEMV kernel is embedded in the matrix multiplication kernel and called when the column number for the second matrix equals 1. At the current implementation stage, the FPGA device must be manually reprogrammed before another FPGA kernel usage. An example output results for second script run are presented below:

```
DenseMatrix(5x5, float)
1.07859 1.34949 1.45986 1.14722 1.73465
1.33295 1.07277 1.32148 1.8037 1.34213
1.20028 1.61552 1.10916 1.75976 1.7518
1.4027 1.59546 1.33853 1.74325 1.26625
1.84846 1.10433 1.26412 1.70519 1.40658
DenseMatrix(5x1, float)
1
1
1
1
1
DenseMatrix(5x1, float)
1541.13
1507.43
1547.2
1527.81
1532.3
```

```
        Bye!
```

Step 3.4 Print the DaphneIR for FPGA SGEMM example:

To print the intermediate representation, the following command has been used:

```
$./bin/daphne --fpgaopencl --explain kernels
       scripts/examples/fpga-sgemm.daph
```

The printed DaphneIR representation is presented below presenting functions calls for Daphne FPGA context creation and FPGA OpenCL matMul for dense matrix multiplication.

```
%0 = "daphne.constant"() {value = 140728553945216 : ui64} : () -> ui64
%1 = "daphne.call_kernel"(%0) {callee = "_createDaphneContext__DaphneContext
__uint64_t"} : (ui64) -> !daphne.DaphneContext
    "daphne.call_kernel"(%1) {callee = "FPGAOPENCL_createFPGAContext"} :
(!daphne.DaphneContext) -> ()
…
%12 = "daphne.call_kernel"(%3, %7, %4, %2, %5, %6, %1) {callee = "_randMatrix
__DenseMatrix_
…
%13 = "daphne.call_kernel"(%7, %8, %4, %4, %5, %6, %1) {callee = "_randMatrix
__DenseMatrix_
…
%14 = "daphne.call_kernel"(%12, %13, %11, %11, %1) {callee ="FPGAOPENCL_matMul
__DenseMatrix_float__DenseMatrix_float__DenseMatrix_float__bool__bool"} :
…
"daphne.call_kernel"(%1) {callee = "_destroyDaphneContext"} :
(!daphne.DaphneContext) -> ()
"daphne.return"() : () -> ()
```

Step 3.5 Example script execution in debug mode for FPGA SGEMM kernel :

To run FPGA SGEMM kernel in debug mode the following command can be used:

```
     $ /bin/daphne --fpgaopencl --debug scripts/examples/fpga-sgemm.daph
```

Additional chosen output results, obtained in debug mode, are presented in the listing attached below.

```
creating FPGA context...
===== Host-CPU setting up the OpenCL platform and device ======
Number of platforms = 2
Allocated space for Platform
…
```

```
Device Name: pac_s10_dc : Intel PAC Platform (pac_f300000)
Device Vendor: Intel Corp
…
===== Host-CPU setting up the OpenCL command queues ======
A rows 448
A cols 1024
X rows 1024
X cols 512
running GEMM kernel
===== Host-CPU transferring W and X to the FPGA device global memory (DDR4)
via PCIe ======
All kernels created
===== Host-CPU enqueuing the OpenCL kernels to the FPGA device ======
…
 *** FPGA execution started!
…
 *** FPGA execution finished!
Time taken: 0.000499 sec
===== Reporting measured throughput ======
Kernel execution time on FPGA: kernel_A_loader, exec time = 0.00040 s,
start=877461.94515 s, end=877461.94555 s
Kernel execution time on FPGA: kernel_B_loader,
exec time = 0.00032 s, start=877461.94523 s, end=877461.94556 s
Kernel execution time on FPGA: kernel_unloader_WAIT_FINISH,
exec time = 0.00050 s, start=877461.94527 s, end=877461.94577 s
Kernel execution time on FPGA: kernel_A_feeder,
exec time = 0.00049 s, start=877461.94530 s, end=877461.94578 s
Kernel execution time on FPGA: kernel_B_feeder,
exec time = 0.00045 s, start=877461.94533 s, end=877461.94577 s
Kernel execution time on FPGA: kernel_Out,
exec time = 0.00049 s, start=877461.94536 s, end=877461.94585 s

  Loader kernels start time          = 877461.94515 s
  Unloader kernels end time          = 877461.94585 s
  FPGA GEMM exec time          = 0.00069 s
  # operations = 469762048
  Throughput: 676.92083 GFLOPS
===== Host-CPU transferring result matrix C from the FPGA device global memory
(DDR4) via PCIe ======
Bye!
Destroying FPGA context...
```

The output contains additional data about particular stages for kernel execution, stages runtime measurement results, and total kernel performance data.

4 Prototype structure

This project structure shows most of the important directories of the prototype (daphne-D7.2-source folder).

- **benchmarks/tpc-h/** (scripts to generate example data)

- **bin/** (compiled system and parser; generated via build.sh)
- **doc/** (basic setup and developer documentation)
- **lib/** (generated kernel libraries)
- **scripts/** (DaphneDSL scripts as examples)
    - **deliverables/** (Deliverable specific examples)
- **src/** (main source code repository)
    - **api/** (cli including daphne which orchestrates the remaining components)
    - **compiler/** (execution, explain, inference, lowering)
        - **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)
            - **MorphStore/** (kernels using MorphStore API)
    - **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)

5 Performance Model for Kernel Selection Process

As introduced in the DAPHNE report D7.1 [D7.1], the challenges for the integration of HW accelerators are (i) developing as well as generating hardware-accelerated kernels, 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. In the previous sections, we described our prototype, which essentially focuses on challenges (i) and (ii). To tackle challenge (iii), we need appropriate performance models for the hardware-accelerated kernels to make well-reasoned selections. In this section, we show our current status with regard to these performance models, looking not only at performance but also at energy-efficiency.

Fundamentally, modern hardware and operating systems offer several control knobs to adjust hardware settings and accordingly the performance and the energy consumption. However, a mapping between the hardware configuration, accelerator device, performance, and energy-efficiency is not always trivial. For example, two CPU cores processing some data in scalar mode might perform as well as a single core processing the same data using SIMD extension (acceleration on CPU), but their energy consumption differs. Further, the performance equality in this example might not exist for all applications, e.g., if it is bandwidth bound, such that enabling a second core hardly produces a performance gain. Nevertheless, the determination of a hardware configuration or to select a specific hardware accelerator offering the best energy-efficiency for a desired performance is an important goal of DAPHNE. To achieve that, we propose to solve this challenge using so-called *Work-Energy-Profiles (WEPs)*. Our work is based on a preliminary work on WEPs [U+16, U+17] and is now being adapted as well as extended to the DAPHNE accordingly. Our current consideration is limited to general-purpose CPUs and the different offered SIMD-extensions for acceleration. We will extend our concept to GPUs as well as FPGAs in the future to design a unified solution.

Basically, a *WEP* is a mapping between performance and energy-efficiency for all possible hardware configurations [U+16]. The *WEPs* have to be determined for a specific kernel and on a concrete heterogeneous hardware system. Based on these *WEPs*, we are able to select an energy-efficient hardware configuration for a requested kernel performance range. In the remainder of this section, we introduce our WEPs as a general solution idea. Then, we demonstrate our approach and introduce a benchmark concept to create *WEPs.* Finally, we discuss selected example profiles and discuss our ongoing efforts in that direction.

5.1 A Model for Performance-Energy Mapping

*Work-Energy-Profiles (*WEPs*)* show the mapping between performance and energy- efficiency for different hardware configurations. In our current context, a configuration covers all system knobs, which are adjustable independent of the implementation. Depending on the hardware and operating system, this can include different properties. For example, general-purpose CPUs usually offer different SIMD extensions making the available instruction set extensions part of the configuration space. However, there are more properties influencing the performance as well as energy-consumption. The following list shows the most common of these properties:

- CPU core frequency
- Number and ID of active cores, where the ID is important if there are multiple sockets or heterogeneous cores
- Number of active threads and use of multi-threading

- SIMD Instruction set and (SIMD) register size

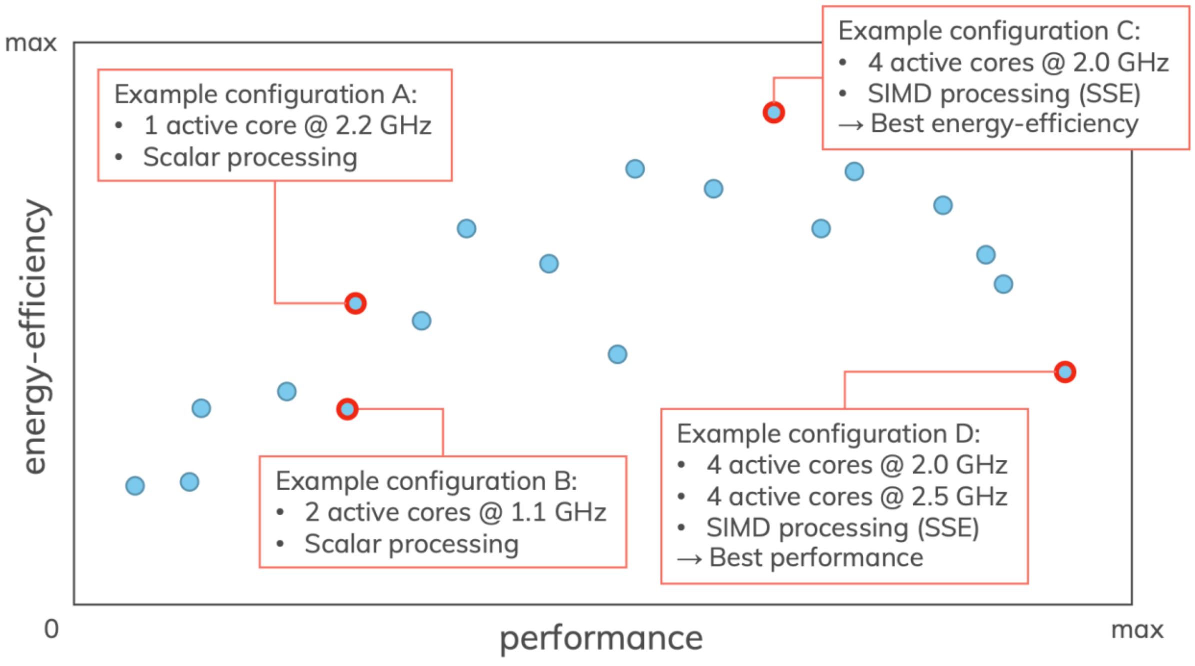

Figure 5: A fictional minimalist Work-Energy-Profile with highlighted example configurations. It shows some effects as observed in real profiles.

Figure 5 illustrates the idea of *WEPs*. Performance and energy-efficiency span a two-dimensional space, and each configuration is represented by a position in this space. Even if this example is a fictional one to keep it simple, two effects are shown, which we also observed in real *WEPs*:

1) The best performing configuration is not always the most energy-efficient one (configuration C vs. D), and
2) There are multiple configurations, which offer a similar performance, but a different energy-efficiency, and vice versa (configuration A vs. B).

Generally, a *WEP* can represent a *system view* or a *thread view*. A *system view* covers the configurations from a system perspective, i.e., the number and ID of the active cores and the frequencies of the active and inactive cores. This kind of profile is illustrated in Figure 5. A *thread view* does not include this system knowledge, but only the properties connected to a single thread, e.g., the frequency of the CPU core the thread is running on, or the instruction set it is using. A *Work-Energy-Profile* of a *thread view* can look different for the same application. This depends on the load of the remaining system, e.g., if other threads are using shared resources like main memory or shared cache. A *Work-Energy-Profile* of a *system view* is always the same for the same application, but becomes more populated the more complex the system is, i.e., the more configurations exist. This complexity can be caused by a larger number of cores or heterogeneous cores.

Figure 6: Configuration options of the ARM big.LITTLE ODROID-XU3.

5.2 Example Heterogeneous Hardware System

Our example system is a single-board computer equipped with an ARM® big.LITTLE CPU, the ODROID-XU3 (specification in Figure 6). We chose this system because it offers some kind of heterogeneity with the following properties. First, all components are hard-wired onto one board including the main memory. Second, the CPU features ARM® cores. Third, the cores are heterogeneous. There are two clusters, one with four Cortex A7 cores, and one with four Cortex

	LITTLE Core Cluster	big Core Cluster
Core Description	ARM-Cortex A7	ARM-Cortex A15
Number of Cores	4	4
Hyperthreads per core	1	1
Frequency Range	0.2 GHz - 1.4 GHz	0.2 GHz - 2.0 GHz
Frequency Steps	100 MHz	
Number of Freq. Steps	13	19
SIMD Extensions	NEON	NEON
L1, L2, L3 Cache	32 kB, 512 kB (shared), -	32 kB, 2048 kB (shared), -

A15 cores. Both clusters can be used independently, i.e., the system is not restricted to cluster switching and frequencies can be set per cluster. All cores offer the same instruction set, but the A15 cores are more powerful and their cluster is equipped with more L2 cache. Hence, this cluster is referred to as the *big* cluster, while the A7 cluster is referred to as the *Little* cluster. The L2 cache is shared between all cores of the same cluster. There is no L3 cache.

The heterogeneity on this system produces a wide spectrum of configuration choices. There are 13 frequency steps on the A7 cluster and 19 on the A15 cluster, as well as 24 different combinations of active and inactive cores. The 24 combinations are composed of 5 variants per cluster (0 - 4 active cores), where one variant is subtracted, which only includes inactive cores. Hence, there are 5 * 5 – 1 = 24 combinations. In combination with the CPU frequencies, this sums up to 13 * 19 * 24 = 5928 different configurations per instruction set. Since the SIMD instruction set NEON is offered additionally to the scalar instruction set, there is a maximum of 5928 * 2 = 11856 available configurations. This is not further reduced by the CPU frequency of a potentially inactive cluster, because that frequency influences the performance of the active cluster if resources, e.g., memory buses, are shared, as explained before.

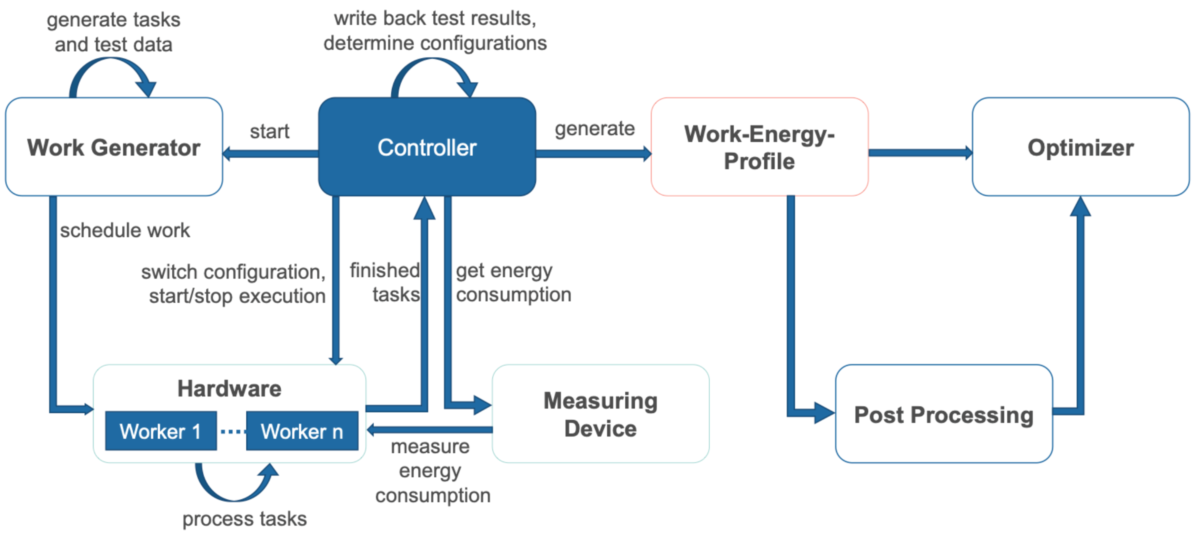

Figure 7: An overview of the benchmark concept

5.3 Creation of Work-Energy Profiles - Benchmarking

One of our main challenges is the creation of *WEPs* covering a large number of possible hardware configurations. To tackle this challenge, we developed a benchmark concept to examine the behavior of performance and energy-efficiency for different hardware configurations in a uniform way [U+16]. Fundamentally, the same kernel – also called task – has to be repeated and recorded for all possible configurations on the target systems. Moreover, not only the kernel but also the test data has to be the same in order to produce comparable results. Therefore, we separated the generation of test-cases and data from the control-flow of our benchmark concept. Generally, an overview of our benchmark concept is depicted in Figure 7. The *Controller* is the centerpiece of our benchmark. It starts the *Work Generator,* which produces tasks and test data. This *Work Generator* has to be adjusted for each investigated kernel. After the *Work Generator* has finished, the *Controller* chooses the first hardware configuration and starts the first test run. Within a test, the corresponding tasks are processed by every worker, whereas a worker is a thread running on a (virtual) core. The workers count their finished tasks. After a fixed time span, the *Controller* shuts down the threads and collects the number of finished tasks, which are later used for calculating the performance. During an active test, the values necessary for the energy computation are recorded by a *Measuring Device*. Depending on the abilities of the *Measuring Device*, the energy computation is either done by the device itself or by the *Controller*. In both cases the final values are collected by the *Controller*. For eliminating odd side effects, a test can be run multiple times. This is repeated for all configurations, with the same tasks on the same test data. After all configurations have been processed, the Controller generates a *Work-Energy-Profile* for the selected kernel and data as depicted in Figure 8. This profile can then be used for in-depth analysis and optimization purposes, e.g., for selecting an energy-efficient configuration satisfying the requested performance constraints or for optimizing the applied algorithm.

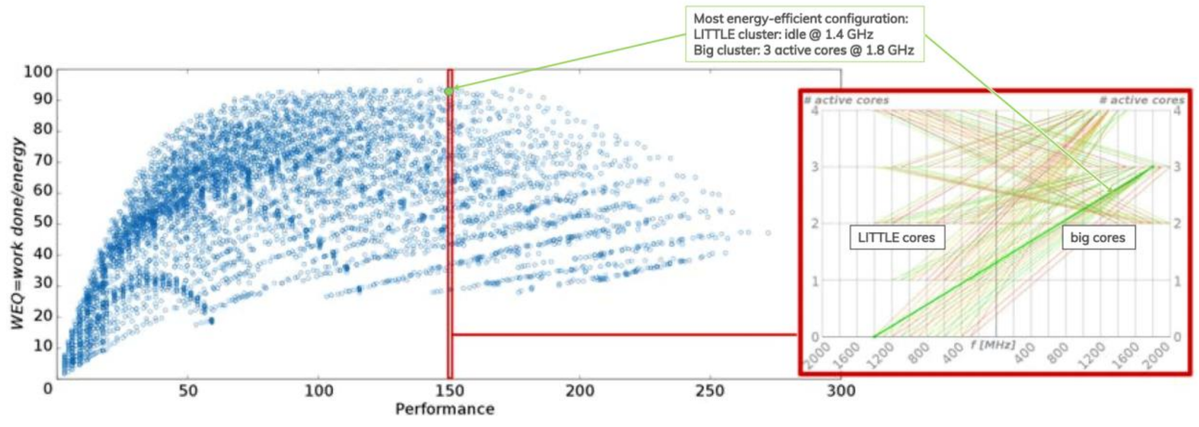

Figure 8: WEP of a system view on our example hardware system and a close-up of the highlighted performance range.

5.4. Example of Work-Energy-Profiles

To get a deeper understanding of our *WEP* approach, Figure 8 illustrates an example for a specific kernel running on our test hardware. The kernel is a synthetic scenario for scalar processing and consists of an equal ratio of memory accesses and numeric computations. The left chart in this figure shows the corresponding *WEP*. The performance is plotted on the x-axis, the y-axis shows the energy-efficiency in terms of a Work-Energy-Quotient (WEQ) [U+16]. To achieve the objective of reducing the overall energy consumed by the system for a specific amount of work, the natural decision for quantifying the energy-efficiency is to calculate it as the quotient of *work done* and *consumed energy*. Accordingly, this relation is usually called *Work-Energy-Quotient (WEQ)*. Each dot in this chart represents a specific hardware configuration. As we can see, different hardware configurations offer a similar performance with a high variance in the *WEQ*. From this *WEP*, we are able to derive various insights.

Generally, we are able to utilize such *WEPs* to directly identify the most energy-efficient configuration (high *WEQ*) for a desired performance range and kernel. In Figure 8, we highlighted a specific performance range using a vertical slice. This performance range can be realized with various hardware configurations as depicted in the right chart of this figure. In this chart, the most energy-efficient configuration is highlighted by a thick green line. The x-axis indicates the frequency of the clusters, the y-axis shows the number of active cores. The left side shows the A7 cluster, the right side the A15 cluster. A line connects the configuration of both clusters and forms the complete configuration. The least energy-efficient ones are marked with a thin red line. This close-up shows the variety inside the configurations, which produce the same performance but a different *WEQ* and the most energy-efficient configuration is not necessarily the most obvious one. For the highlighted performance range, the most energy-efficient configuration consists of 3 active A15 cores running at 1.8 GHz, while all A7 cores are idle at 1.4 GHz. The graph also shows a number of configurations with high CPU core frequencies on both, the idle cluster and the active cluster, which show a comparatively low *WEQ*. As already mentioned, the frequency of an idle cluster can still influence the overall performance if resources are shared. In this specific case, the cache and core clock are only shared within the same cluster, but the memory and the memory bus clock are shared between both clusters. We assume that the shared bus clock is the reason for the unexpected behavior.

5.5 Ongoing Work

From our point of view, these *Work-Energy-Profiles* are well-designed performance models to select the best-fitting hardware configuration with regard to performance as well as energy-efficiency perspectives. In particular, due to the generic nature of WEPs, we can distinguish between memory accesses and computations, covering all possible situations. So far, we have limited ourselves to general-purpose CPUs with SIMD functionalities for acceleration. In the near future, we want to transfer and generalize this approach to other accelerators such as GPU and FPGA. One drawback of our unified WEP approach is that the benchmarking of the profiles is very time-consuming. To overcome that shortcoming, we will investigate approaches for approximating such WEP profiles as shortly introduced in [U+17].

Another focus of our work in this area will be to fully integrate these WEPs into the DAPHNE system. From our point of view, our WEPs represent accurate performance (based on benchmarking per kernel per device) and thus, they are the key to the integration of HW accelerators into advanced compilation techniques as well as automatic kernel and data placement decisions. In detail, we aim to model data access along the data path with our WEPs under different access characteristics and compute capabilities for various HW accelerators. Then, this allows the comparison across different HW accelerators and thus, the selection of the best-fitting device.

6 Conclusion

This document reported a snapshot of our DAPHNE prototype regarding HW acceleration as a follow-up of deliverable D7.1. In particular, we introduce three example scenarios as the main drivers for the current prototype and describe how to execute these examples with enhanced hardware support. Additionally, we gave an overview report on the devised performance models in terms of Work-Energy-Profiles (WEPs) for a cost-based approach for kernel and data placement decisions in a heterogeneous hardware environment in this deliverable.

References

[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

[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).

[DU+20] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner: "MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model" (PVLDB 2020)

[S+19] Srivastava, Nitish and Rong, Hongbo and Barua, Prithayan and Feng, Guanyu and Cao, Huanqi and Zhang, Zhiru and Albonesi, David and Sarkar, Vivek and Chen, Wenguang and Petersen, Paul and Lowney, Geoff and Herr, Adam and Hughes, Christopher and Mattson, Timothy and Dubey, Pradeep."T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations "," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019

U+16	Annett Ungethüm, Thomas Kissinger, Dirk Habich, Wolfgang Lehner: Work-Energy-Profiles: General Approach and In-Memory Database Application. TPCTC 2016, pages 142-158, 2016
U+17	Annett Ungethüm, Patrick Damme, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner: Balancing Performance and Energy for Lightweight Data Compression Algorithms. ADBS (Short Papers and Workshops) 2017, pages 37-44, 2017
U+20	Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, Erich Focht: Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. CIDR 2020