A Case Study on DaCe Portability & Performance for Batched Discrete Fourier Transforms

Måns I. Andersson
Stefano Markidis
mansande@kth.se
markidis@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

With the emergence of new computer architectures, portability and performance-portability become significant concerns for developing HPC applications. This work reports our experience and lessons learned using DaCe to create and optimize batched Discrete Fourier Transform (DFT) calculations on different single node computer systems. The batched DFT calculation is an essential component in FFT algorithms and is widely used in computer science, numerical analysis, and signal processing. We implement the batched DFT with three complex-value array data layouts and compare them with the native complex type implementation. We use DaCe, which relies on Stateful DataFlow multiGraphs (SDFG) as an intermediate representation (IR) which can be optimized through transforms and then generates code for different architectures. We present several performance results showcasing the potential of DaCe for expressing HPC applications on different computer systems.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; **Shared memory algorithms**.

KEYWORDS

Data-Centric Parallel Programming, on-node optimization

ACM Reference Format:
Måns I. Andersson and Stefano Markidis. 2023. A Case Study on DaCe Portability & Performance for Batched Discrete Fourier Transforms. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023), February 27-March 2, 2023, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3578178.3578239

1 INTRODUCTION

In the last decade, we assisted in the golden age of computer architecture [14] with the development and adoption of new computer architectures, ranging from accelerators from different vendors to matrix engines [19] and processors designed for deep-learning workload [16]. Given the current trend with more companies investing in processor design and the emerging computational paradigms, such as quantum computing, it is likely that more and more new computer systems will be available for the HPC programmer who is tasked to design applications that can run efficiently on a large spectrum of computer systems. Most of these new computer hardware come with a set of programming interfaces and abstractions that are typically specific to computer systems. This diversity of the systems and programming interfaces puts a significant burden on the programmers who must target several systems and programming frameworks to develop the application.

Ideally, there exists a programming framework that allows to write the application code once and run on different architectures without code rewriting. Such a framework is portable. The key for portability is the establishment of common abstractions that can generally express data movement, operations and parallelism on diverse hardware [23].

Ideally also, the programming framework would be *performance-portable* [5, 21], guaranteeing a minor performance degradation when compared to the performance of a programming interface specifically designed for a given architecture. For instance, if a code has been written in CUDA to run on NVIDIA GPUs, the ideal portable framework should guarantee that performance degradation is not larger than a small fraction of the performance achieved with CUDA. Given the burning importance of writing portable and performance-portable code, scientists developed several approaches to provide a unified approach for programming efficient code on diverse computer systems. Among the most notable examples in the HPC ecosystems, there are Kokkos [8], RAJA [15], Mamba [7] and DaCe [3]. In particular, an application, called OMEN and fully developed with DaCe [27], won the Gordon Bell award in 2019, making DaCe one of most promising approaches for developing portable application on exascale supercomputers and heterogeneous systems. In this work, we report our experience in using DaCe for designing portable code for solving batched Discrete Fourier Transform (DFT) kernel. Such a computational kernel is not trivial and allow us to experiments different design choices, such as data layout for the complex number, data reduction and parallelization. In addition, to have a high performance implementation of batched DFT is critical for the development of fast high-radix multi-dimensional Fast Fourier Transforms (FFTs) that rely on optimized DFTs. Furthermore it is a growing trend in HPC to run batched computations of small kernels [6].

HPC Asia '23, June 03–05, 2023, Singapore

ACM ISBN 978-1-4503-9805-3/23/02…$15.00
https://doi.org/10.1145/3578178.3578239

In this work, we use the DaCe framework to design high-performance portable Batched DFTs. The main contributions of this work are the following:

(1) We present our experience and lesson learned in developing a non-trivial kernel, such as the DFT, using DaCe.
(2) We port our DaCe batched DFT to multi-core CPUs and Nvidia GPUs using the DaCe's Intermediate Representation (IR) and code generation.
(3) For the DFT calculation, we design and implement three data layouts for complex arrays: Array of Structures (AoS), Split Data Layout, and Structure of Arrays (SoA).
(4) During our usage of DaCe, we extend the framework with optimized reductions to include complex128 and complex64 types natively in DaCe.
(5) We apply DaCe optimization techniques, specific to different backend hardware.

The paper is structured as follows. First, we provide a brief overview of the DaCe programming model and of the Batched DFT kernel in Section 2. We describe our methodology in using DaCe for developing batched DFT in Section 3. The experimental setup and performance results are described in Sections 4 and 5, respectively. Previous work is briefly reported in Section 6. Finally, we conclude the paper summarizing and discussing the results with Section 7.

2 BACKGROUND

2.1 The DaCe Framework

DaCe is a framework for developing high-performance portable scientific code with restricted Python. The main strengths of the DaCe approach are its Intermediate Representation (IR) and the code generation: these two components allow for performance optimization and portability. There are multiple supported front-ends for creating IR including MATLAB/Octave and Python frontend. The Python frontend is the most used one and the one used in this paper.

The IR can be manipulated by transforms through the command-line or with a GUI. There are two graphical methods: Diode, an IDE run through the Internet browser or with a plugin for Visual Studio Code (VS Code). Although there are multiple front ends the IR can easiest be generated by adding the `@dace` decorator to a Python function written in restrictive Python and then calling the IR creating function `to_sdfg()`. The computation can then be run Just-in-Time (JiT) by simply calling the IR-object as a function.

DaCe's code generator produces C++ code based on the IR and then lets the C++ compiler (g++ as default) compile it. The user can also create the IR, apply performance and portability transforms and then compile the code ahead-of-time.

The IR in DaCe is based on a data-centric model which aims to compartmentalize computation and data-movement. This is enforced by the design of the IR which is called Stateful DataFlow multiGraph (SDFG). To enable the key concept of compartmentalization, SDFG is focused on dataflow, and the ability to move data from a storage container to another or to a computation without performing any computation.

The Data-Centric model is built up of a collection of programming primitives, summarized in Fig. 2 and described in detail in Ref. [3]. In Fig. 2, the graph's edges are called `Memlets` and correspond to the data movement between data containers (`Data`) or computation (`Tasklet`). The data-container is an N-dimensional array. The fine-grained computation in tasklets is immutable from data-centric transforms and can be written in any language supported by the target platform. To support Python there is an implementation of a Python-to-C++ converter. It is important when writing custom code to not allow any data access without memlets. The multigraphs of the data movement between containers and computation are confined within `States` which allow the representation of cyclic data dependencies and control flow, after the execution of a state, the state transition edges can specify conditions and assignments whilst connecting states forming a state machine. This representation allows for a description of data-flow and parallelisms with any granularity, and is therefore a viable IR for portable high-performance programs.

DaCe provides an easy way to express parallelism. For instance, `Map` is parallel scope in which nodes can reside. *Coarsening* allows a hierarchical view of parallelism.

The performance and the portability of the generated code (and therefore SDFGs and DaCe) are heavily dependent on the DaCe library, where backend specific code for primitives such as reductions and write-conflict handling is done. The backend library makes a large use of C++ templates to enable the use of generic types for multiple types of hardware.

2.2 Batched Discrete Fourier Transform

In this work, we focus on using DaCe for implementing batched DFT on different architectures. The DFT can be simply formulated as a complex-valued matrix-vector product. Instead of implementing the batched DFT as several matrix-vector products, we implement it as a matrix-matrix product, where each column is an independent vector. We do this to highlight the data-movement aspect of DaCe and the algorithm. A complex number is an expression on the form $a + ib$ where $i^2 = -1$. The common algebraic properties (commutativity, associativity, etc.) hold.

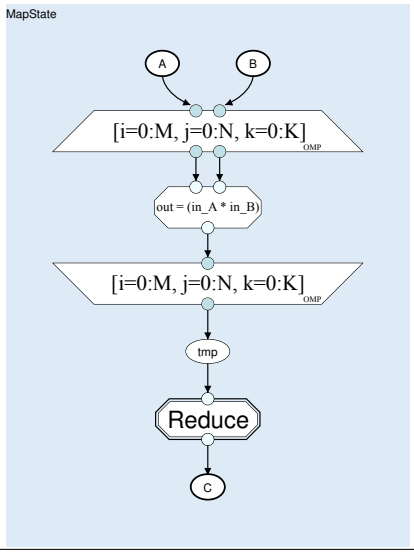
We see that the book-keeping is limited to a few number of operations. This allows for the idea of manually handling the data-layout of the complex number. The DFT matrix used for the matrix-vector multiplication is constructed as follows:

$$\mathrm{DFT}_{N_{m,n}} = (\omega_N)^{mn}, \quad \text{where} \quad \omega_N = e^{-2\pi i/N} \quad \text{for} \quad 0 \leq m, n < N.$$

N is the size of the input vector. The real and imaginary matrices are symmetric. However, we do not use of this property in the implementation.

Batched DFTs are widely used in developing FFT implementations. For instance, we can find the batched DFT in the *Pease* FFT algorithm [24], a modification of the classic Cooley-Tukey algorithm.

There are generally two types of implementations of complex matrix-matrix multiplication. The first type is by using an implemented complex type that supports "native" complex calculations. In this case, DaCe relies on the fact that the destination language/programming model has a native implementation of complex such as C++'s `std::complex` or CUDA's `cuda::std::complex`. This is in DaCe managed through the DaCe equivalent `dace::complex`.



Primitive	Description
State	state machine (light blue square)
Data	array container (ellipse)
Memlet	data-movement (arrows)
Stream	streaming data (dotted ellipse)
Tasklet	nodes with any computation (hexagon)
Map	abstraction for parallelism (split hexagon)
Consume	Computations on streams (dotted split hexagon)
WCR	Write-Conflict resolutions (dotted line)
Reduce	Reduction
Invoke	Call a SDFG in a nested fashion.

Figure 1: An SDFG representation of a GEMM and the primitives used in DaCe's SDFG

The second method is to transform the problem to use only real-valued variables, e.g., by treating the real and complex components individually - *induced* complex calculations, among these you can find several implementations such as 1M, 3M, 4M and so on presented in [25, 26], where the name represents the number of real-valued matrix multiplications used to induce a complex-valued matrix multiplication.

A native complex type is probably most appreciated by programmers and would score high on a productivity score, but needs hardware-specific support which might not be met, it is also theoretically a hindrance for performance as there are multiple ways of ordering the data-type, some potentially more efficient than others. To allow for more data-movement optimizations we will investigate the use of different hand-implemented structures.

3 METHODOLOGY

In this section, we start by presenting the high-level implementation of the different data-layouts for complex numbers matrices and arrays in DaCe. We then present the baseline comparisons we use to evaluate the performance.

After that, we describe the DaCe implementation of the different batched DFT algorithms and present the performance transforms that are applied to the SDFGs of the different implementations.

We then investigate the portability by evaluating the GPU-transformed SDFG, some performance transforms, and compiled code. Finally, we show some necessary changes to the DaCe library that have been made to obtain the expected behavior from the code generation on GPU.

We define the baseline algorithm implementations as a standard and batched complex GEMM (CEMM for complex numbers in double precision and ZEMM for complex numbers in single precision). We choose to use DaCe library expansions of CEMM as the baseline. DaCe has a *pure* expansion which can be used when there is no BLAS libarary available, included for completeness and will view it as a benchmark for the implementation. However, the main comparison is with MKL's BLAS implementation on both AMD and Intel and cuBLAS on Nvidia GPUs. Note that parts of the implementation of node-expansion of batched CEMM on multi-core CPU is handled by the DaCe library in comparison with suggested batched BLAS designs [6].

3.1 Implementation

The batched DFT calculation is in essence a matrix-matrix product between a square matrix with the twiddle factors and a rectangular matrix with the input vectors. We assume the DFT matrix to be a constant and the matrix-values are stored as a `dace.constant` [1] which in the generated code as a C++ `contsexpr`. There is no complex constant so we separate the real and imaginary part into individual constants. The DFT computation is made up of a `map` over three variables, a temporary array, that is later reduced. The naive algorithm is written with DaCe's Python-frontend Listing 1 which results in the SDFG seen in Fig. 2. For this study we have the same data layout for the input and output.

```
@dace.program
def DFT128(x : dace.complex128[N M]):
    tmp = np.empty((N, M, N), dtype=dace.complex128)
    for j, k, i  in dace.map([0:N,0:M,0:N]):
        tmp[i,j,k] = DFT[i,j]*x[j,k]
    dace.reduce(lambda a,b:a + b,tmp,x,axis=[2])
```

Listing 1: Naive skeleton code implementation of the DFT based on the GEMM example provided by the DaCe repository. N is the size of the input vectors and M is the size of the batch.

We implement three different approaches:

(1) A traditional *Interleaved Data Layout*, where the pairs of real and imaginary values are stored in an alternating fashion. We call this data layout *AoS*. This is similar to the native complex type we compare with.
(2) An easy-to-implement SoA data-layout, where the real part is stored first and the imaginary part is stored after.
(3) A *Split Data Layout* where the two are separated into two independent arrays.

We implement the same SDFG with the SDFG backend on the following form. Where we easily can manipulate the `tasklet` which

[1] since DaCe 0.14.0 deprecated and replaced by `dace.compiletime`

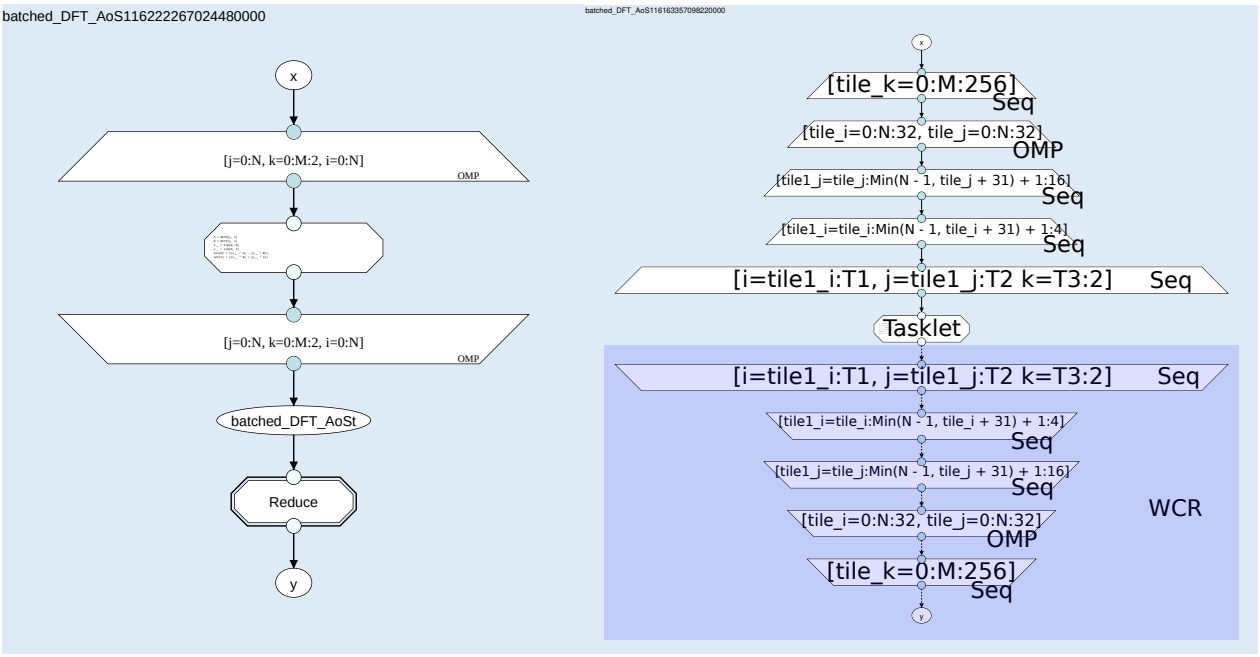


Figure 2: To the left: The inital SDFG of the AoS implementation of the batched DFT. To the right: transformed with a tiling strategy. We note that the main difference between the graphs is that the reduction node has been transformed and that Write Conflict Resolution (WCR) memlets have been introduced together with multiple nested sequential and parallel maps.

$$\begin{aligned} \text{AoS} &: [a_0, b_0, a_1, b_1, \cdots a_N, b_N] \\ \text{SoA} &: [a_0, a_1, \cdots b_0, b_1, \cdots] \\ \text{SP} &: [a_0, a_1, \cdots], [b_0, b_1 \cdots] \end{aligned}$$

Figure 3: How the data-structures are designed for a single complex-valued vector, at the top AoS, in the middle the SoA and at the bottom the split data layout where the color represents different arrays that the element belongs to.

can be seen as a micro-kernel. We do this to get better control working with unsupported features.

At the current state some combinations of the tasklet implementation languages and transforms are incompatible. The native complex tasklets written in Python work only for complex128 not for complex64. However, if we write the tasklet in C++, this can be worked around but with the risk of performance transforms failing to be correct.

For the performance portability, we evaluate the `auto_optimize`-function from `dace.transformation.auto` for both GPU and CPU. This function is described in Ref. [28] and employs a simple optimization heuristic depending on the intended hardware, which we test on similar SDFGs. We also apply the tiling optimization strategy described in Ref. [3], the transforms here are based on the GEMM optimizations in the repository. Tiling is a common optimization strategy for improving cache performance by better exploiting temporal locality. We also employ the `LocalStorage`-transform to leverage caching within the tiles.

For each implementation, we mimic the performance transforms describe above. There should be a very similar possible optimization for each implementation ending with a main state depicted in Fig 2. However, this is not always the case. For SoA the initial step of fusing the map and the reduction with `MapReduceFusion` does not work as expected.

The kernel is based on multiplication with a known constant. The CUDA performance deteriorates quickly with the size of the DFT because of `DaCe`'s handling of constants. Therefore, we choose to create an SDFG where the constants are changed to standard arrays. As for the CPU optimization, the basis for this paper is the GEMM and corresponding SDFG optimization (IR passes) presented in Ref. [3] and found in the repository.

3.2 DaCe Backend Update

We noticed that the performance for the native complex implementation underperformed compared with the corresponding real GEMM. This discrepancy comes from that the original DaCe library has a specialized method for reductions of float and double based on `omp atomic capture`. However, the fall-back method used for complex relies on `omp critical`, with a detrimental effect on multi-core CPU performance.

We implemented an extension of `_wcr_fixed` for complex64 (2 x single) and for complex128 (2 x double), by reusing the optimized methods for float and double, we do this by unrolling each complex pair and handling them separately as presented in Ref. [4]. However, we have to be careful when adding this to the backend. It is very beneficial for some reductions, but can be detrimental in others. The code generated from the `MapReduceFusion` transform the generic

optimization is much faster than this type-specific one and should not be overrided.

An example of the reduction with a sum can be seen in Listing 2.

```
template <>
struct _wcr_fixed<ReductionType::Sum, dace::complex64>
{
    static DACE_HDFI void reduce_atomic(dace::complex64 *
     ptr, const dace::complex64 &value)
    {
        single *real_ptr = reinterpret_cast<single *>(ptr
     );
        single *imag_ptr = real_ptr + 1;

        _wcr_fixed<ReductionType::Sum, single>::
     reduce_atomic(real_ptr, value.real());
        _wcr_fixed<ReductionType::Sum, single>::
     reduce_atomic(imag_ptr, value.imag());
    }

    DACE_HDFI dace::complex64 operator()(const dace::
     complex64 &a, const dace::complex64 &b) const
    {
        return _wcr_fixed<ReductionType::Sum, dace::
     complex64>()(a, b);
    }
};
```

Listing 2: In this extension of fast summing reduction for `std::complex<float>` we see how a typical "primitives" back-end look, utilizing the already implemented `_wcr_fixed<ReductionType::Sum, single>`.

4 EXPERIMENTAL SET-UP

The performance experiments in this study are carried out on the following systems:

Dardel is an HPE Cray EX supercomputer at KTH in Stockholm, Sweden. We use the `main` partition on Dardel where each node has two AMD EPYC 7742 3.2GHz CPUs with 64 cores each and with 256 GB RAM in total.

DEEP-System is a supercomputer at Jülich Supercomputing Centre (JSC) in Jülich, Germany. We use the `CN-nodes` which has two Intel Xeon 'Skylake' Gold 6146 3.2GHz with 12 cores (24 threads). 192 GB RAM.

GPU-system: This system has a consumer grade Nvidia RTX3060 mobile with a AMD Ryzen 9 5900hs CPU and 32GB DDR4 RAM.

We perform the timings through the DaCe profiling and timing tools, using the standard configuration for DaCe-0.13.3. We compile our codes using GCC 11.2.0 on both systems for the multi-core CPU code (since it is the suggested compiler in the library dependencies). We note that the code generated with the `aopt` automatic optimization for complex could not be compiled with GCC but with icpx/clang. CUDA 11.5.0 for the CUDA code on the `GPU` machine.

For each test-case we run random input vectors. The test-cases we run are the native, SoA, AoS, and BLAS with complex128.

The use-case we are mostly interested in are large radix-computations in FFT algorithms, as it is common to use radices from two to 32-64 it is therefore useful with a batched DFT kernel when developing FFT [10, 11]. The size of an input array to a FFT can vary from two to three orders in Molecular Dynamics (MD) [2] to several of million elements in Cosmology and Medical Imaging [1]. The input array for a one-dimensional FFT corresponds to the radix size (N) multiplied with the batch size (M). For a three-dimensional FFT a batch of one-dimensional FFTs must be performed, which essentially just increases the batch size M. We first test an input of $N = 64$ and $M = 128$, corresponding to a input vector length of 64 (or radix of 64 in an FFT) and a batch size of 128. This is equal to an one-dimensional FFT of 8192 elements or a batch of smaller FFT, which we consider an interesting size of FFT. For the GPU test we increase the range of inputs to capture the architecture strengths.

The first test case shows the strong scaling of the naive implementation on the two test systems on a input of 64×128, this is followed by the evaluation of the automatic optimization and then the user optimized version.

We do not utilize the integration of CuPy on device arrays to measure only the kernel timings without communication to GPU. However, this needs to be done in the code before generating the SDFG by annotating the expected type of the input variables to be a GPU array, therefore any performance measurement is including communication of data.

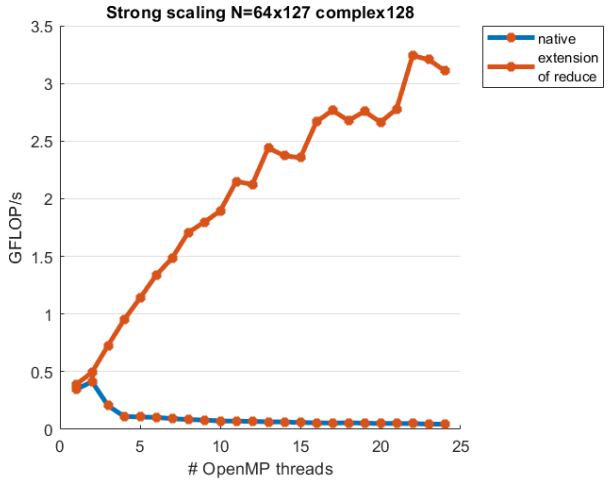
We assume all algorithms to scale as $8MN^2$ which is the basis for the FLOPS calculations.

5 RESULTS

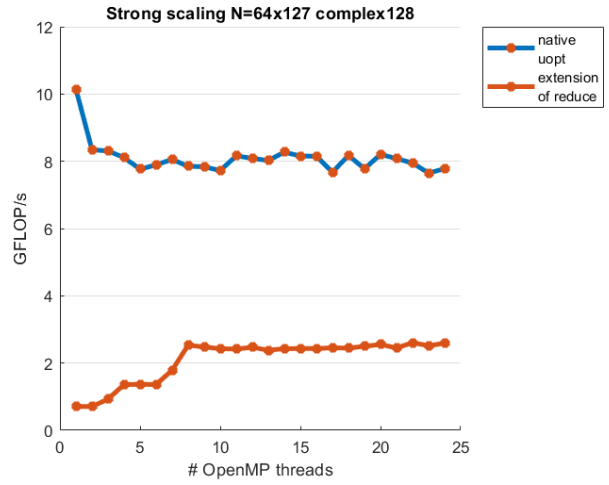
In Fig. 4 shows two examples of the scaling before and after applying the reduce extension describe in 3.2. For the problem size of this task, it is necessary to apply this extension when running the naive implementation Fig. 4a, where the standard DaCe library clearly dose not scale. It is however not clear to a DaCe-coder what code-paths will be taken, so one must be careful with the implementation, this is seen by the Fig. 4b where a transformed SDFG produces code with less performance.

Now, we present the performance results in GFLOPS for multi-threaded code, generated by DaCe performing the naive implementation without any performance transformations. Fig. 5a shows the strong scaling result for our DaCe batched DFT varying the number of threads on the AMD EPYC 7742. The different color represents different data layouts and implementation complex arrays in DaCe. We can observe that strong scaling increases linearly until eight threads and saturates approximately at 3 GFLOPS for the complex native implementation and at 2.8 GFLOPS for the AoS and SoA data layouts. After saturation, we note erratic behavior from the AoS and SoA implementations whereas the native dies out. In Fig. 5b, we present the strong scaling results for the batched DFT on the DEEP cluster for the naive implementation without any performance transformations. On the DEEP system, we also note a clear linear dependency for low core counts up to approximately eight and then a gradual decline from there on. We also see that all of the data structures follow a type of staircase pattern not seen on the Dardel system. We see that it also scales better and achieves a higher peak performance at around 4.4 GLOPS for SoA and around 3.7 GFLOPS for AoS.

For the auto-optimized code, gcc fails to compile the native complex code. In Fig. 6a we also see that the auto-optimization does not improve performance compared with the naive implementation, in fact, it is significantly slower than the naive implementations.

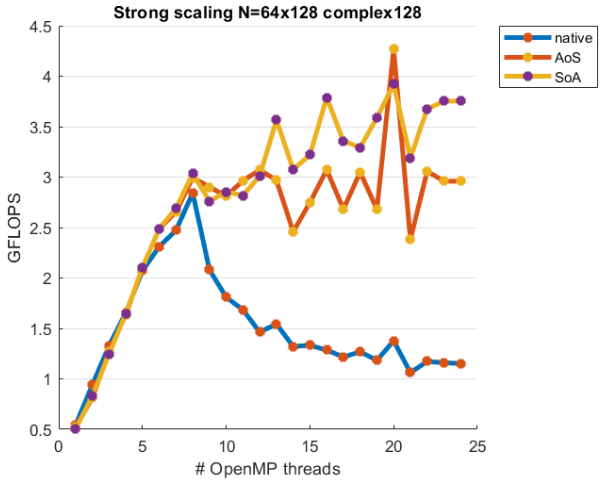


(a) DEEP-system

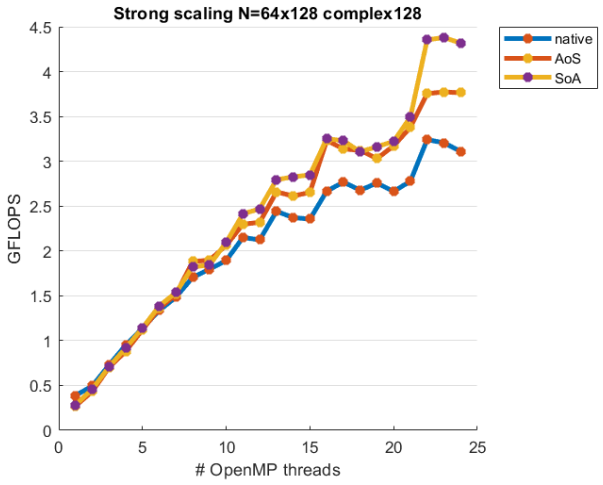


(b) DEEP-System

Figure 4: The scaling with and without the reduction presented in 3.2. for (a) the naive implementation and (b) after applying a `MapReduceFusion` in the user optimized case.



(a) Dardel



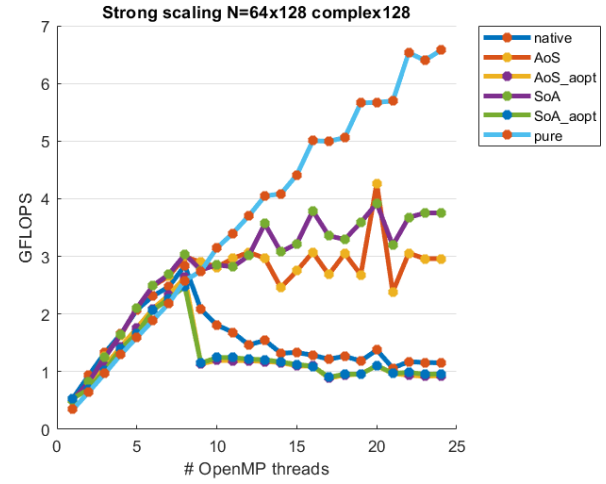
(b) DEEP-System

Figure 5: Strong scaling for input 64 x 128 for the baseline algorithms on (a) Dardel cluster with AMD Processors and (b) the DEEP-System with Intel Processors.

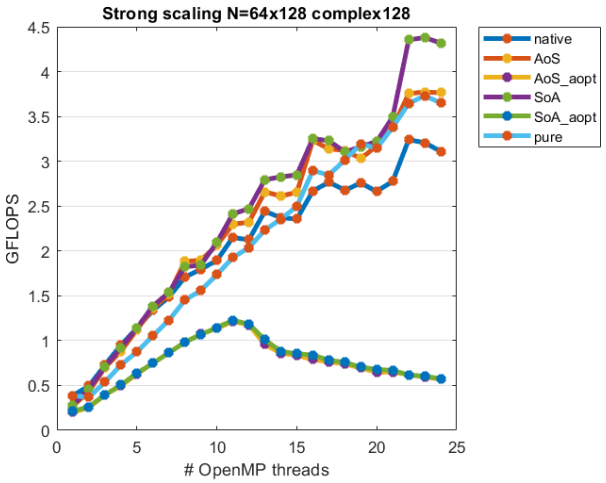
In Fig. 6b it is clear that the auto-optimization is slower than the naive implementation on the DEEP-system even before saturation. We can see no significant difference between the data layouts.

We also included the DaCe `pure` expansion, which can be seen as an optimization or a reference solution as it is always available through DaCe, we can see that it performs as well as the naive implementations in the DEEP-system and that it scales better than the naive implementations on Dardel.

We then perform optimizations as described in the Section 3. The scaling results using an optimized version of the batched DFTs on Dardel is shown in Fig. 7a. We note that the optimized versions of AoS and SoA reach to approximately 30 GFLOPS as peak performance at around 10 threads. When the optimized SoA plateaus and starts a slow decline, the AoS does a notable dip between 10 threads and 15 threads. The split data layout (SP) with user optimizations peaks at 70 GFLOPS at the nine threads and stays above 50 GFLOPS. Comparing with DEEP in Fig. 7b we find a notable difference, instead of finding the maxima at 10 threads the AoS plateaus but starts scaling again, achieving the same performance as on Dardel. On the other hand the SoA implementation does not reach the same level of performance on DEEP as on Dardel. The SP implementation has almost half the performance compared with on Dardel, but retains a similar shape. The performance of MKL is surprisingly low, especially on Dardel it is likely due to the small problem size and the high single threaded performance, but it is also possible that it is due to the AMD CPUs on Dardel. The native

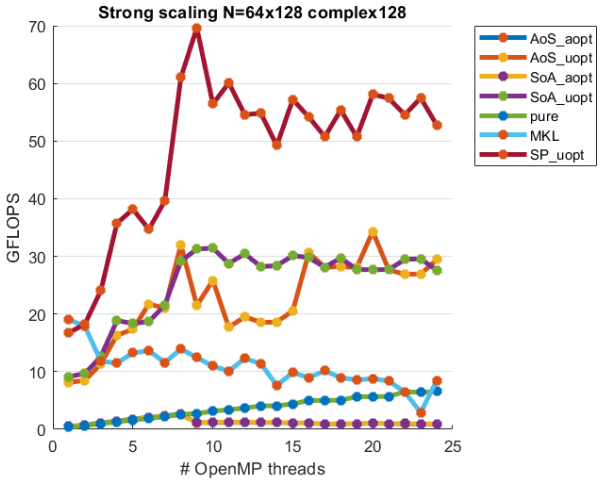


(a) Dardel

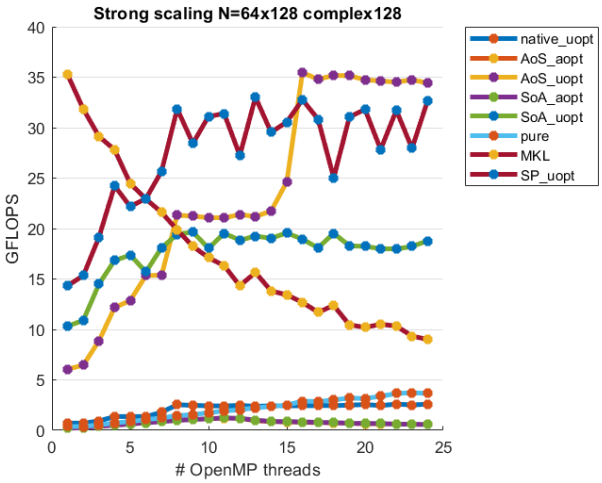


(b) DEEP-System

Figure 6: Strong scaling on Dardel (a) and DEEP-System (b) for 64 x 128 for the auto-optimized (`aopt`) algorithms compared with the naive implementations and the `pure` library extension



(a) Dardel



(b) DEEP-System

Figure 7: Strong scaling on Dardel (a) and DEEP-system (b) for input of size 64 x 128 for the optimized algorithms

complex performs poorly due to the effects shown in Fig. 4b; however unchanged backend only performs a fraction of the ones we get from AoS and SoA with its' seven GFLOPS.

We evaluate the GPU performance by looking at a range of input vector sizes and batch sizes. In Fig. 8 we have the naive implementations transformed to GPU SDFGs and compiled to CUDA. The performance is similar between different data-layouts, and is saturated at a small problem size although the native implementation seems to perform around 10% better for the largest problem sizes.

In Fig. 9 we compare the auto-optimized methods with the naive code, all the implemented data layouts performs very well with the automatic optimizer on GPU. The automatically optimized AoS implementation performs around six times better than the naive. The native complex type does not work with the automatic optimizer.

In Fig. 10, we present the weak scaling results of the user-optimized implementations on the GPU. In these tests, we vary the transform size and batch size. The implementation with the AoS and the SoA data layouts performs poorly because of the failure of applying `MapRuduceFusion`. The Split-Data layout performs approximately the same as the native CUDA implementation, both are significantly slower than the cuBLAS implementation. For a large range of input sizes, the auto-optimization is faster than the user-optimized one.

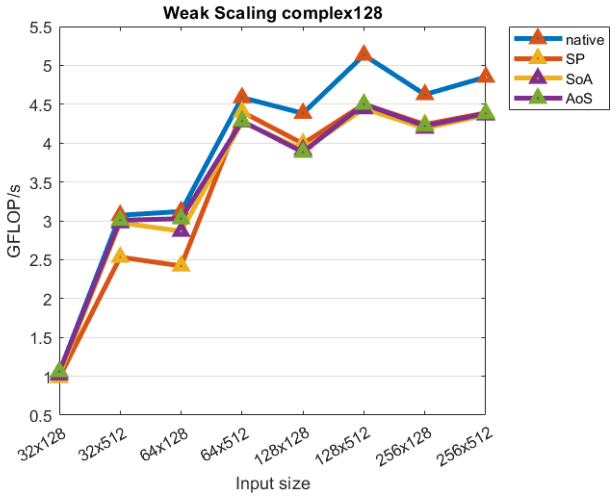


Figure 8: Weak scaling for the GPU transformed naive implementations.

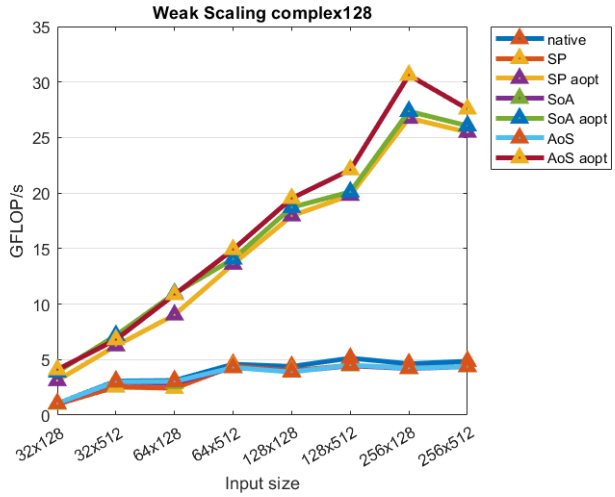


Figure 9: Weak scaling for the GPU transformed implementations with auto optimization (`aopt`) compared with the naive code.

6 RELATED WORK

Many authors investigated the performance of CEMM/ZEMM kernels using already optimized GEMM kernels, with the portability of the algorithm as the main motivation on several systems [25, 26].

The impact of different data layouts for CEMM and FFT optimized for vector instructions is evaluated in Ref. [22], showing promising results by allowing different data layouts at different stages of the algorithm, including the split data layout used in this paper.

Data layout optimization and transformations are important in compiler technology. A plethora of work has been produced over the years about this topic. One compelling approach is to employ

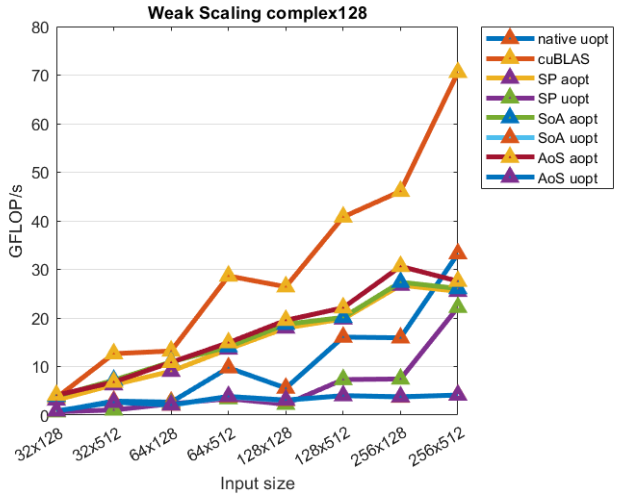


Figure 10: GPU with user optimization compared with cuBLAS and the automatic optimization.

transformable graph-based IR with language and platform independent optimization passes similar to the SDFGs in DaCe. The most notable example is the LLVM IR [17], which has gained significant traction within the research community. Multi-Level Intermediate Representation (MLIR) [18] combines domain- and backend-specific dialects such as the FFT-specific implementation [13] and the GPU implementation found in Ref. [12].

Automatic tuning has a history of use in the design of FFT libraries such as FFTW [11], SPIRAL [9] and UHFFT [20].

There are multiple approaches for high-level libraries for *performance portability* in scientific software, such as DaCe: examples are Kokkos [8], RAJA [15], and Mamba [7].

7 DISCUSSION AND CONCLUSION

In this paper, we described the process of implementing a batched DFT with the framework DaCe to make a portable kernel with the possibility to achieve high performance. We investigated the data-centric programming style of DaCe together with three different data-layouts and compared it with the native C++ implementation and the corresponding BLAS operation. We investigated the performance of the code generated by the naive implementations, the auto-optimizing tool, and the hand-optimized.

The different data-layouts perform somewhat differently on the different vendors' CPUs. On Dardel (AMD) there seems to be an edge for the native implementation, whereas on DEEP (Intel) the higher-level handwritten AoS seems to be more performant for each of the optimization levels.

We note that it might be challenging to use standard optimization techniques developed for use on SDFGs with tasklets with only one input and output per array. For example, we did not manage to get the `MapReduce` to work with the AoS and SoA implementations written with the DaCe front-end (the one AoS created directly with the SDFG API worked well). This led to particularly poor performance of the AoS and SoA GPU ports. On the other hand there is a lack of support for the native complex which requires updates

in the DaCe code generation to even get a running GPU code which. However, in the end turned out to be the most performant for our use-case. The split data-layout was most easy to port and run with reasonable performance on all platforms.

Auto optimization worked poorly for this problem on CPU regardless of the data layout, but on GPU it is promising performing well above the naive implementation. It was especially the AoS that benefited from the optimization, even compared with the native, which also is a AoS, which is encouraging for the data-enteric programming model, suggests that maybe complex should be implemented within DaCe limit the need for special case implementations in the backend.

The same template code that was necessary for the GPU code to run was beneficial as a reduction strategy for CPU too and was even necessary to make the native naive implementation comparable with the others. It was however detrimental to leave it in the user-optimized version, which by itself was slower than the real-valued implementations and clearly has a different backend.

The study can in the future be extended to a distirbuted setting with DaCe's support for MPI and to the other supported backends such as AMD GPUs and FPGAs, as we see great potential in the ability to write one code and port it to multiple backends.

To guarantee the success of DaCe an important task is to guarantee that the functions supporting the SDFGs are performant, such as copy, reduce, for different data-types float32, float64, complex128 and so on. We believe our contributions highlight the difficulty of achieving true performance-portability on diffrent hardware, especially for functionality that is dependent not only float64.

ACKNOWLEDGMENTS

Financial support was provided by the DEEP-SEA project. The DEEP- SEA project has received funding from the European Union's Horizon 2020/EuroHPC research and innovation program under grant agreement No 955606. National VR contribution from Sweden matches the EuroHPC funding. The computations of this work were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC, partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

REFERENCES

[1] Abhinav Agarwal, Haitham Hassanieh, Omid Abari, Ezz Hamed, Dina Katabi, et al. 2014. High-throughput implementation of a million-point sparse fourier transform. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–6.

[2] Måns I Andersson, N Arul Murugan, Artur Podobas, and Stefano Markidis. 2022. Breaking Down the Parallel Performance of GROMACS, a High-Performance Molecular Dynamics Software. *arXiv preprint arXiv:2208.13658* (2022).

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

[4] Gabriel Bengtsson. 2020. Development of Stockham Fast Fourier Transform using Data-Centric Parallel Programming.

[5] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance portability across diverse computer architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 1–13.

[6] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science* 108 (2017), 495–504.

[7] Tim Dykes, Clément Foyer, Harvey Richardson, Martin Svedin, Artur Podobas, Niclas Jansson, Stefano Markidis, Adrian Tate, and Simon McIntosh-Smith. 2021. Mamba: Portable Array-based Abstractions for Heterogeneous High-Performance Systems. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 10–21.

[8] H Carter Edwards and Christian R Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 18–24.

[9] Franz Franchetti and al. 2018. SPIRAL: Extreme Performance Portability. *From High Level Specification to High Performance Code* 106, 11 (2018).

[10] Matteo Frigo. 1999. A fast Fourier transform compiler. In *ACM SIGPLAN 1999 Conference on Programming language design and implementation*. 169–180.

[11] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[12] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2020. Domain-specific multi-level IR rewriting for GPU. *arXiv preprint arXiv:2005.13014* (2020).

[13] Yifei He, Artur Podobas, Måns I Andersson, and Stefano Markidis. 2022. FFTc: An MLIR Dialect for Developing HPC Fast Fourier Transform Libraries. *arXiv preprint arXiv:2207.06803* (2022).

[14] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[15] Richard D Hornung and Jeffrey A Keasler. 2014. The RAJA portability layer: overview and status. (2014).

[16] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[17] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *CGO 2004.* 75–86.

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[19] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[20] Dragan Mirković and S Lennart Johnsson. 2001. Automatic performance tuning in the UHFFT library. In *International Conference on Computational Science*. Springer, 71–80.

[21] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2016. A metric for performance portability. *arXiv preprint arXiv:1611.07409* (2016).

[22] Doru T. Popovici, Franz Franchetti, and Tze Meng Low. 2017. Mixed data layout kernels for vectorized complex arithmetic. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091024

[23] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. 2017. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020.

[24] Charles Van Loan. 1992. *Computational frameworks for the fast Fourier transform*. SIAM.

[25] Field G Van Zee. 2020. Implementing high-performance complex matrix multiplication via the 1m method. *SIAM Journal on Scientific Computing* 42, 5 (2020), C221–C244.

[26] Field G Van Zee and Tyler M Smith. 2017. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Transactions on Mathematical Software (TOMS)* 44, 1 (2017), 1–36.

[27] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[28] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, portability, performance: data-centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.