Morpheus unleashed: Fast cross-platform SpMV on emerging architectures

Christodoulos Stylianou
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
c.stylianou@ed.ac.uk

Mark Klaisoongnoen
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
mark.klaisoongnoen@ed.ac.uk

Ricardo Jesus
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
rjj@ed.ac.uk

Nick Brown
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
n.brown@epcc.ed.ac.uk

Michèle Weiland
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
m.weiland@epcc.ed.ac.uk

***Abstract*—Sparse matrices and linear algebra are at the heart of scientific simulations. Over the years, more than 70 sparse matrix storage formats have been developed, targeting a wide range of hardware architectures and matrix types, each of which exploit the particular strengths of an architecture, or the specific sparsity patterns of the matrices.**

In this work, we explore the suitability of storage formats such as COO, CSR and DIA for emerging architectures such as AArch64 CPUs and Field Programmable Gate Arrays (FPGAs). In addition, we detail hardware-specific optimisations to these targets and evaluate the potential of each contribution to be integrated into *Morpheus*, a modern library that provides an abstraction of sparse matrices (currently) across x86 CPUs and NVIDIA/AMD GPUs. Finally, we validate our work by comparing the performance of the Morpheus-enabled HPCG benchmark against vendor-optimised implementations.

***Index Terms*—sparse matrix storage formats, AArch64, FPGA, performance portability, productivity**

I. Introduction

Since their inception, sparse matrices have become the centrepiece of many applications in science and engineering. Their ability to efficiently store the non-zero values of a matrix reduces the memory footprint of the matrix and eliminates redundant computations, allowing for larger problems to be tackled. More than 70 sparse matrix storage formats (i.e data structures) have been introduced over the years [1], each leveraging different properties of the matrix or target hardware architecture to achieve better performance. Many iterative methods for solving large-scale linear systems and eigenvalue problems, which often arise in a variety of scientific and engineering applications, consist of many Sparse Matrix-Vector Multiplications (SpMVs) operations that often dominate the applications' runtime. Literature shows that no single format can perform optimally across all kinds of matrices or hardware [1]–[5], with interest on optimising SpMV being renewed every time new platforms emerge.

Morpheus [6] is a C++ library that provides an abstraction of sparse matrices, allowing for efficient and transparent switching of sparse matrix storage formats at runtime across traditional backends such as x86 CPUs and NVIDIA/AMD GPUs. By dynamically adapting the underlying sparse matrix data-structure to optimally suit an operation, target architecture, or sparsity pattern of the matrix, *Morpheus* enables new optimisation opportunities and thus increased performance [6]. At the time of writing, *Morpheus* supports three core formats: Coordinate (COO), Compressed Sparse Row (CSR) and Diagonal (DIA).

In this work, we explore the performance of SpMV on different formats across two non-traditional High Performance Computing (HPC) architectures, AArch64 CPUs and FPGAs, both currently gaining traction in the area of HPC. In addition, we exploit specific hardware features of each target, such as the Scalable Vector Extension (SVE) on AArch64, to optimise the kernels, and investigate the challenges in integrating the optimisations in *Morpheus*. In summary, our contributions are:

- We discuss how we incorporated the Arm Performance Libraries (ArmPL) into Morpheus for AArch64 targets, thus making the performance of ArmPL available to Morpheus users "out-of-the-box" for COO and CSR.
- To work around ArmPL's lack of DIA support and to enable a better assessment of ArmPL's performance, we develop SVE-optimised SpMV routines for COO, CSR and DIA matrices.
- For over 2100 matrices available in the SuiteSparse [7] collection, we demonstrate the performance of the SpMV kernels on an A64FX-based (AArch64) HPE Apollo 80 cluster.
- We evaluate our efforts (i.e. incorporating ArmPL and our SVE-optimised routines into Morpheus) on HPCG [8], comparing our Morpheus-based HPCG implementation [9] against the Arm-optimised version [10] on the HPE Apollo 80 system.
- We provide implementations of the SpMV kernel on FPGAs for each of the three formats. Performance is evaluated on the SuiteSparse set and the challenges in porting the implementations in *Morpheus* are described.

II. Motivation

New storage formats are proposed every time new architectures emerge, aiming to exploit the new characteristics and features of the new hardware. Even-though more than 70 formats are available, no single format performs best across different hardware, operations, and sparsity patterns. As a result, being able to switch formats dynamically offers new opportunities for optimisation and increased performance. The adoption of new formats can be a tedious process as this requires significant changes in the source code. Libraries such as *PETSc* [11], *GINKGO* [12] and *Morpheus* [6] offer multiple formats through various abstractions, enabling users can switch to different formats at runtime. In other words, users can take advantage of new formats added in the library with minimal source code changes, thereby easing their development effort.

Nevertheless, when it comes to the adoption of new hardware, libraries might require major changes in their interface, especially when the hardware utilises a new programming model that the library does not yet integrate. For example, when GPUs emerged in HPC, libraries had to evolve to manage the existence of two mostly independent memory spaces between the CPU and GPU and to support the accelerator model, where CPUs offload work to the GPUs. For software to remain performant throughout the life-cycle of an hardware architecture (and beyond), we need to ensure that it can adapt to the requirements that are imposed by the current and future iterations of hardware.

In this work, we consider AArch64 CPUs and FPGAs, two very different architectures that are currently gaining traction in HPC. We investigate their performance in sparse linear operations, such as SpMV, as well as the challenges involved in integrating them in *Morpheus*. The integration of code in *Morpheus* for the aforementioned targets presents different challenges and requires different integration approaches, making the candidates representative for the integration of other architectures.

III. Background

A. Emerging Architectures

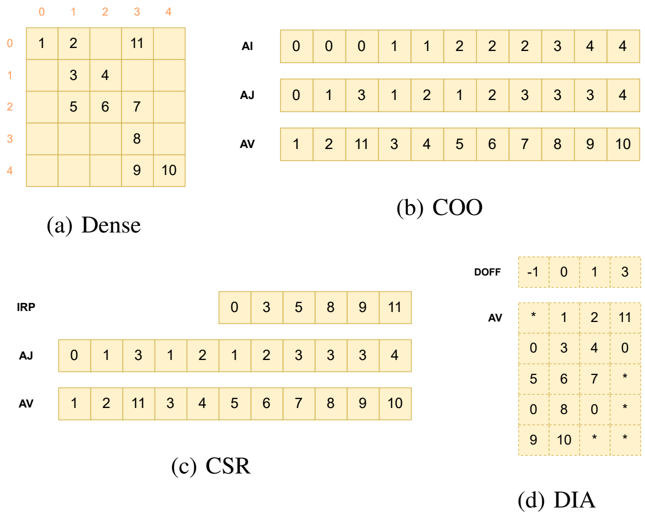
The two emerging architectures that we propose to integrate with the *Morpheus* libraries are AArch64 CPUs and FPGAs. Through the process of porting the SpMV routines for the three core *Morpheus* storage formats (COO, CSR and DIA) to AArch64 CPUs and to FPGAs, we explore the challenges around performance optimisation and performance portability for integration into the Morpheus library.

1) AArch64 CPUs: Despite being somewhat newcomers on the HPC scene, Arm CPUs have already proved to be extremely competitive against the more traditional x86 processors [13]–[16]. One of the key enablers for the high-performance of modern-day AArch64 CPUs is SVE [17], [18], an advanced architecture extension for Single Instruction Multiple Data (SIMD) processing that features long (scalable) vectors, gather-load and scatter-store instructions, and per-lane predication. These features make SVE an excellent architecture to implement fast SpMV computations.

To accelerate the adoption of Arm-based hardware in HPC, Arm has developed the ArmPL [19], a set of core routines for high-performance computing applications optimised for AArch64 processors. ArmPL consists of BLAS, LAPACK, FFT, Sparse, libamath (subset of libm) and libastring (subset of libc for strings) routines for both single-threaded and OpenMP multi-threaded processing. The sparse linear algebra routines provided by ArmPL support high-performance SpMV on dense, CSR, CSC, COO and BSR matrices. However, as of version 23.04 (the most recent at the time of writing), ArmPL does not support the DIA format.

2) FPGAs: Field Programmable Gate Arrays (FPGAs) provide a very large number of configurable logic components sitting within a sea of configurable interconnect. Modern FPGAs also contain hardened components, such as Block-RAM (BRAM) which provides fast on-chip memory similar to a CPU's level 1 cache and DSP slices for undertaking floating point arithmetic. Modern FPGAs are also commonly coupled with external High Bandwidth Memory (HBM2), DDR, and high performance networking capabilities. A major challenge with FPGAs has been the historically significant time investment required in programming the technology and need for detailed hardware-level knowledge on behalf of developers. Nevertheless, in recent years FPGA hardware and software development ecosystems have become far more capable, and with toolchains such as Intel's Quartus Prime and Xilinx's Vitis software developers can now program FPGAs and accelerate HPC workloads by writing code in C or C++ using High-Level Synthesis (HLS). Programming FPGAs is now becoming more a question of software development than hardware design, and consequently lowering the entry barriers for programming these devices has enabled numerous communities to investigate and explore FPGAs in their respective domains [20]–[22].

Being able to tailor hardware to the code at the electronics level provides the potential to implement custom optimisation techniques around memory access and data transfer. However, to obtain best performance the programmer must rework their algorithm into a dataflow style [23] and this also often delivers much higher energy-efficiency than traditional architectures too [24]. AMD Xilinx's most recent generation, known as the Versal Adaptive Compute Acceleration Platforms (ACAP) [25], combines the programmable logic (PL) resources on the FPGA chip with more than 400 AI Engines (AIE). These AIE are hardened on the chip and each represent a Very Long Instruction Word (VLIW) processor capable of executing seven instructions per cycle and interconnected between AIEs and to PL on the FPGA with fast Network-on-chip (NoC). This upgrade in compute capabilities with 8-way vectorisation per AIE is especially interesting to kernels such as SpMV which exhibit large potentials in this regard.



	0	1	2	3	4
0	1	2		11	
1		3	4		
2		5	6	7	
3				8	
4				9	10

(a) Dense

AI	0	0	0	1	1	2	2	2	3	4	4
AJ	0	1	3	1	2	1	2	3	3	3	4
AV	1	2	11	3	4	5	6	7	8	9	10

(b) COO

IRP	0	3	5	8	9	11					
AJ	0	1	3	1	2	1	2	3	3	3	4
AV	1	2	11	3	4	5	6	7	8	9	10

(c) CSR

DOFF	-1	0	1	3
AV	*	1	2	11
	0	3	4	0
	5	6	7	*
	0	8	0	*
	9	10	*	*

(d) DIA

Fig. 1: A 5×5 dense matrix with 11 non-zero values and its equivalent representations in 3 sparse matrix storage formats.

B. Sparse Matrix Storage Formats

Sparse matrices exploit the property that the majority of coefficients in the matrix are zeros by not explicitly storing those values. In other words, sparse matrix storage formats only store the non-zero coefficients and the necessary information that is required to rebuild the original position of each coefficient in the dense matrix. Each storage format rebuilds the original index of each coefficient in a different way and as a consequence each format can have very different memory layout as shown in Figure 1. With the underlying data structure across formats varying significantly, accessing and manipulating entries in each format can result in different memory access patterns, costs and interfaces amongst formats.

Figure 1 shows the representation of a 5×5 dense matrix in the three sparse storage formats used throughout this work: COO, CSR and DIA. The most basic and well-known formats are COO and CSR. Both are considered *general purpose* formats, suitable to a broad range of matrices of arbitrary sparsity patterns and target architectures. Below we provide a brief description of these formats. For a more comprehensive description please refer to Saad, Y. [26].

1) COO (Figure 1b): uses three arrays, whereby each non-zero element (`AV`) is explicitly stored together with its column (`AJ`) and row indices (`AI`) with no guarantees imposed in the ordering of the elements. The SpMV algorithm for COO is shown in Algorithm 1.

Algorithm 1 COO SpMV

```
for i=0:NNZ do
    y(ai(i)) += av(i)*x(aj(i));
end for
```

2) CSR (Figure 1c): was implemented as an optimisation to COO, where the `AI` array was compressed to generate an array of row pointers. As a result, CSR explicitly stores the column indices and non-zero values, and also uses an array of pointers (`IRP`) to mark the boundaries of each row, thereby reducing the memory footprint of the format by essentially compressing the row indices. As the row pointers are used to represent the position of the first non-zero element in each row, and the last entry shows the total number of non-zeros in the matrix, CSR naturally imposes an ordering across rows, though not within each row. The SpMV algorithm for CSR is shown in Algorithm 2.

Algorithm 2 CSR SpMV

```
for i=0:nrows do
    $sum = 0$;
    for j=irp(i):irp(i+1) do
        sum += av(j)*x(aj(j));
    end for
    y(i) = sum;
end for
```

3) DIA (Figure 1d): is a specific purpose format originally designed to perform optimally on vector architectures. It is suitable for regular sparsity patterns. DIA uses a two-dimensional array, where each column holds the coefficients of a diagonal of the matrix (`AV`), and an integer offset array (`DOFF`) keeps track of where each diagonal starts. Therefore, the DIA format is suitable for matrices with structures that dominate along the diagonals, such as banded matrices that result from discretisation methods like the Finite Differences Method (FDM). The SpMV algorithm for DIA is shown in Algorithm 3.

Algorithm 3 DIA SpMV

```
for i=0:nrows do
    $sum = 0$;
    for j=0:ndiags do
        k = i + doff(j)
        if k$\geq$0 and k<N then
            sum += av(i,j)*x(k);
        end if
    end for
    y(i) = sum;
end for
```

IV. SpMV implementations on AArch64 CPUs

Optimising code for general-purpose processors such as most AArch64 CPUs typically involves one of two things: at a higher level, application programmers can choose to utilise target-specific libraries that implement core algorithms and routines efficiently for the specific targets; meanwhile, at a lower level, application programmers can write efficient code targeting specific CPU (micro-)architectures themselves, usually either through intrinsics (or built-ins), which are functions treated specially by compilers to make features of the target architecture directly available to programmers, or via writing explicit assembly for their target. In this work we have explored integrating these two forms of optimisations for AArch64 CPUs into *Morpheus*, which we describe in this section.

The Arm Performance Libraries (ArmPL) [19] are a set of core routines developed by Arm for HPC applications for AArch64 targets (especially Neoverse-based). It contains BLAS, LAPACK, FFT, Sparse, libamath (a subset of libm) and libastring (a subset of libc for strings) routines for both single- and multi-threaded processing provided via both C and Fortran interfaces. ArmPL's sparse routines support dense, CSR, CSC, COO and BSR matrices. These routines are provided via an API similar to FFTW, where the description of the problem is independent from its execution. In this sense, to set up an SpMV operation with ArmPL we start by creating a handle to a sparse matrix. This is achieved with the `armpl_spmat_create_*` family of routines. Usually, this matrix is provided in a common format such as CSR. In our case, however, the handle is created for the specific format we are using at the time. Then, hints are provided to the handle in an attempt to speedup future SpMV calls with `armpl_spmat_hint` calls. The handle is then used in an optimisation stage similar to that found in other libraries such as the aforementioned FFTW, where the library tries to determine the best algorithms and implementations for the specific matrix and target. This step is issued with `armpl_spmv_optimize`. Once these optimisations have been run, the handle can be used repeatedly to execute SpMV and other sparse algebra computations via the `armpl_*_exec_*` family of routines. Once the handle is not needed anymore, it can be destroyed by calling `armpl_spmat_destroy`. Given ArmPL's high-performance for AArch64 targets and ease-of-use, we have chosen it in this paper to explore how and how well target-specific libraries (of which ArmPL is an example) can be integrated into *Morpheus*, thus offering its high-performance to *Morpheus* users transparently.

An alternative way of developing highly-optimised code for AArch64 CPUs is by leveraging the Arm C Language Extensions (ACLE) [27]. The ACLE are a set of compiler intrinsics that expose advanced features of the Arm achitecture and aim to enable the development of applications and libraries portable across compilers and across Arm micro-architectures. One of the most disruptive extensions of the Arm architecture (in particular AArch64) is the Scalable Vector Extension (SVE) [17]. Unlike other contemporary single instruction multiple data (SIMD) extensions such as Neon from Arm and the AVX extensions from x86, SVE is a "vector-length-agnostic" (VLA) vector extension. This means that the programmer does not program to vector registers of specific width; instead, they program to a slightly different programming model where the width of the vector registers is not known at compile-time. In practice, this fact makes SVE highly portable, as the same code (and, in fact, binary) can run transparently on hardware with vector registers of different widths. Besides offering this extra flexibility and portability, SVE is also an extremely complete instruction set, supporting instructions highly suitable for High Performance Computing (HPC) and Machine Learning (ML) applications. Some examples of the high-performance features of SVE are per-lane predication (i.e. control on a per vector element basis), gather-loads and scatter-stores, speculative vectorisation, and horizontal and tree-based reductions. The gather-loads/scatter-stores of SVE are especially useful for SpMV computations given the latter's irregular and indirect access patterns (as exemplified in Section III-B). Due to these reasons, in this work we used ACLE to develop SVE-enabled implementations of SpMV kernels for COO, CSR and DIA matrices. Our implementations result mostly from a transliteration of the default C++versions of the SpMV kernels present in *Morpheus* to ACLE. Below we highlight two of the main implementation details of our SVE-enabled SpMV kernels. We utilise the algorithms presented in Section III-B throughout to facilitate the discussion.

The indirection in the output vector `y` through the row index vector `ai` in the COO kernel complicates the vectorisation of the kernel's loop. This happens because independent elements of `ai` might point to the same element of `y`. In these cases, the writes to `y` have either to be serialised or accumulated before being issued, so that a single write is effected and no updates to `y` are lost. In our SVE-enabled COO implementation we have chosen the second approach, whereby in each iteration in `i` we only work with the elements `(ai(i), ai(i+1), ai(i+2), ...)` that match (i.e. are equal to) `ai(i)`. We leverage SVE's predication features to create a mask of such elements and only operate on them. This allows us to accumulate the products `(av(i)*x(aj(i)), av(i+1)*x(aj(i+1)), ...)` that correspond to the same `ai(i)` before writing them to `y`, thereby effecting a single accumulation in `y`. In C and ACLE pseudocode, this correspond to:

```
vbool_t pg;
for(i = 0; i < NNZ; i += vcntp(pg, pg)) {
  // Generate mask for the values of i < NNZ
  pg = vwhilelt(i, NNZ);

  // Load values ai(i, i+1, ...)
  vidx_t vai = vld1su(pg, ai+i);

  // Generate mask for the elements
  // (ai(i), ai(i+1), ...) == ai(i)
  pg = svcmpeq(pg, vai, ai[i]);

  // Load values of aj, av, and x
  vidx_t vaj = vld1su(pg, aj+i);
  vtype_t vav = svld1(pg, av+i);
  vtype_t vx = svld1_gather_index(pg, x, vaj);

  // Compute products av(i)*x(aj(i))
  vtype_t vr = svmul_x(pg, vav, vx);

  // Accumulate products
  yval[ai[i]] += svaddv(pg, vr);
}
```

Though this approach might be inefficient if the mask `pg` becomes too "hollow" (i.e. with too few active elements), in practice in our tests this does not happen frequently. Thus, as shown in Figure 4-a, this strategy allows us to achieve significant speedups over default (i.e. compiler generated) and

ArmPL implementations.

Another important subtlety in the way we implement our SVE-enabled SpMV kernels lies in the way we vectorise the DIA format. Instead of vectorising the inner loop (in `j`), as is more common, we have vectorised the outer loop (in `i`). We have done this for two reasons, namely (i) the memory accesses in `av` are contiguous (i.e. with stride 1) in `j`, thus we get better cache utilisation from loading several `(av(i, j), av(i+1, j), ...)` at a time and then looping through `(j, j+1, ...)` sequentially, and (ii) this avoids doing a horizontal reduction to accumulate the values of `sum` before writing it to `y`. Additionally, we once again resort to SVE's predication features to mask the valid `k` indices for each inner iteration. In pseudocode, we have:

```
vidx_t vidx = vindex(0, ndiags);
for(i = 0; i < rows; i += vcnt()) {
  // Initialise sum
  vtype_t vsum = vdup(0);

  // Create mask for the values of i < nrows
  vbool_t pg = vwhilelt(i, nrows);

  for(index_type j = 0; j < ndiags; j++) {
    index_type k = i + doff[j];

    // Generate mask for the valid k's
    // p1 = k < 0
    // p2 = k < N
    // pm = p2 && !p1
    vbool_t p1 = vwhilelt(k, 0);
    vbool_t p2 = vwhilelt(k, N);
    vbool_t pm = svbic_z(pg, p2, p1);

    // Load av and x
    vtype_t vav =
      svld1_gather_index(pm, av+i*ndiags+j, vidx);
    vtype_t vx = svld1(pm, x+k);

    // Compute the products av(i, j)*x(k) and
    // accumulate them
    vsum = svmla_m(pm, vsum, vav, vx);
  }

  // Store the results in (y(i), y(i+1), ...)
  svst1(pg, y+i, vsum);
}
```

Once again, it might happen that the predicate `pm` might have too few elements active for vectorisation to pay off, though this does not happen often. Furthermore, in our tests (Figure 4-c) the choice of performing outer-loop vectorisation over inner-loop vectorisation leads to significant speedups. However, we note that the compilers we tested, namely GCC 11.2.0 and LLVM 15.0.7, are not able to perform this outer-loop vectorisation automatically due to the complex control flow it entails.

V. SpMV implementations on FPGAs

FPGAs provide programmable logic that can be configured at the electronics level to represent the hardware tailored to a specific algorithm. The way that FPGAs operate fundamentally differs from how *Von-Neumann* architectures such as traditional CPUs operate, therefore during porting of CPU-based algorithms these need to be re-engineered to a dataflow style of computing suitable for such devices [23]. *Dataflow*, a fundamental concept for the quest of performance on FPGAs, is typically built around concurrently running stages, known as dataflow stages, that stream data between themselves and each stage comprises individual pipeline(s) which will start to process a new iteration each cycle. On AMD-Xilinx FPGAs, which are the focus of this work, this regularly includes the definition of *dataflow regions* which are connected through HLS streams providing usually lower number of cycles from reads and writes than accesses to global memory.

Porting the three *Morpheus* storage formats COO, CSR and DIA to the FPGA, we implement each of these as individual kernels and deliver three different bitstreams, which configure the FPGA, each containing a kernel of the SpMV of the respective storage format algorithm as presented in Algorithm 1 for COO, Algorithm 2 for CSR and Algorithm 3 for DIA. Working with AMD-Xilinx's HLS toolchain Vitis in C/C++, this means we had to set up the host code on CPU and the HLS kernels on the AMD-Xilinx Alveo U280 FPGA which is used in this work. AMD-Xilinx's host-device model is built upon OpenCL, and as such we (i) initialise the device in the host code, (ii) create OpenCL buffers for input and output data, (iii) transfer the required input matrix and vector data to the device, (iv) execute the kernel on device once data has been transferred and is available in the global device memory and lastly (v) transfer the result vector back to the host. Depending on the matrix dimensions, the required input and output data size varies and we transfer data and run the device kernels in single precision floating-point. While the number of elements and data sizes of input data for COO and CSR are relatively similar, the DIA format requires substantially more matrix values depending on the number of padded rows and the number of diagonals to be stored: for COO all inputs (`AI`, `AJ`, `AV`) require `n=NNZ` elements with data sizes of 32-bit for integers and 32-bit for floats, for CSR `IRP` requires `n=nrows+1` elements of 32-bit for integers, `AJ` and `AV` both require `n=NNZ` elements of 32-bit for floats, and for DIA `DOFF` requires `n=ndiags` for 32-bit integers and `AV` requires `n=padded_rows*ndiags` elements of 32-bit for floats. Compared to the input data requirements of COO and CSR, the DIA format requires significantly more input elements and therefore increases the amount of data to be transferred from host to device.

Under the assumption that the majority of SpMV computations occur as one out of many routines within a larger kernel such as in HPCG, it is reasonable to assume the availability of input data on the device and therefore we only report kernel execution time on the device (excluding device initialisation and setup time to download the bitstream and configure the hardware and excluding data copy on and data copy off time). Our implementation provides the data copy to device and data copy back to host each in a single OpenCL buffer and with the Vitis toolchain this means that we are bound to the tooling's single buffer size limit of 4 GB[1], inhibiting us from

[1] https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Buffer-Creation-and-Data-Transfer

running matrices that require individual input buffers larger than the buffer size limit. For the Alveo U280 FPGA, there is 8 GB of High-Bandwidth Memory (HBM) and 32 GB of DDR DRAM available on-card. We utilise the faster HBM for data transfers to device, which is why we do not run larger matrix benchmarks than those requiring less than 8 GB of accumulated input data for the respective SpMV algorithms.

Figure 2 illustrates the structure of our dataflow version of COO on the FPGA using AMD Xilinx Vitis HLS. Each green box is a separate dataflow region running concurrently, with arrows between these illustrating streams of data that flow from one cycle to the next. The purple boxes in Figure 2 depict connection to external, high bandwidth, memory where all reads and writes are packed in chunks of 512 bits to best utilise the memory controllers. Dataflow regions run in parallel to load the data from HBM2 and then pass individual data elements to the next stages. The dashed line in Figure 2 represents a ping-pong buffer, which is a common double buffering technique used in FPGA programming where the dataflow stage will concurrently write to one buffer whereas the subsequent stage is served with data from a previous copy of the buffer and these then switch at a predefined point.

One area of concern is to ensure that pipelines that comprise the dataflow regions have initiation intervals (II) of one, where the II refers to the number of clock cycles before the next iteration in a loop can be started. An initiation interval of one, commonly written as `II=1`, means that the dataflow design can essentially yield a result every cycle. Considering the much slower clock frequency of FPGAs, typically around 300MHz, compared to CPUs and GPUs, for performance it is critically important that every cycle counts when it comes to computation.

The following pseudocode describes our `reduce` stage in the dataflow region of Figure 2 which illustrates our approach in moving to an optimal II of 1:

```
1  void reduce(const unsigned int A_nrows,
2    const unsigned int A_nnnz,
3    unsigned int A_rind[MAX_NROWS],
4    hls::stream<dtype> &sum_stream,
5    dtype y_val[MAX_NROWS]
6  ) {
7  nrows_loop:
8  for (unsigned int row_index=0; row_index<A_nrows;
9    row_index++) {
10   dtype acc_part[LATENCY]={0,0,0,0,0,0,0,0};
11 nnnz_loop:
12   for (unsigned int i=0; i<A_nnnz;
13     i+=LATENCY) {
14     #pragma HLS pipeline
15 acc_partial_loop:
16     for (unsigned int j=0; j<LATENCY; j++) {
17       #pragma HLS unroll
18       dtype sum = sum_stream.read();
19       unsigned int tmp_A_rind =
20         A_rind[i*LATENCY+j];
21       if(tmp_A_rind == row_index) {
22         acc_part[j] += sum;
23       }
24     }
25   }
26 acc_final_loop:
27   for (unsigned int j=0; j<LATENCY; j++) {
28     #pragma HLS unroll
29       y_val[row_index] = acc_part[j];
30   }
31 }}
```

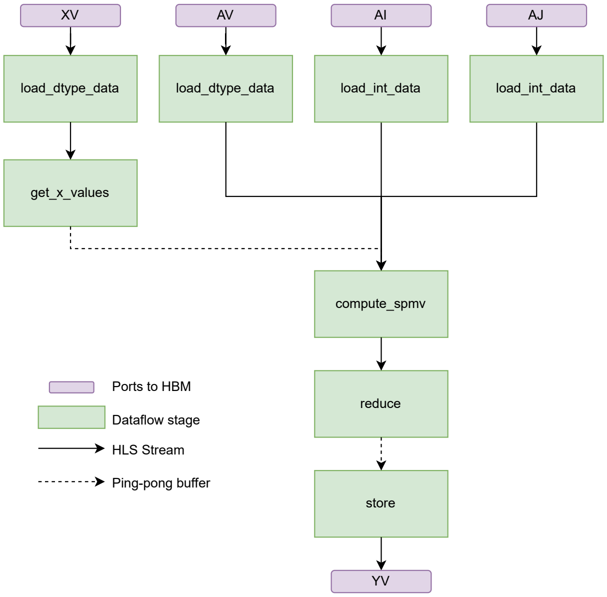


Fig. 2: Dataflow region of the COO algorithm implementation on FPGA with dataflow stages operating in parallel and connected through HLS streams and ping-pong buffers implemented in UltraRAM.

Implementing the naive reduction to `YV`, as part of the COO Algorithm 1, based on the *row index* of each non-zero element on FPGA as `fadd` required eight clock cycles. The HLS tooling detected a spatial dependency and therefore avoided pipelining our *reduce* function at lower II than eight. Splitting the reduction into `LATENCY=8` partial accumulations in the inner loop `acc_partial_loop` on Line 15 in the pseudocode above, which the tooling can fully unroll, the outer `nnnz_loop` will still be pipelined with II=8 by the HLS tooling but through the full inner unroll essentially yields a result every cycle. As in the COO algorithm we only reduce across the same *row index*, on Line 21 we introduce a conditional to only accumulate the partial sum for the same *row index*. On Line 26, we then perform the final accumulation across the previously computed partial accumulations and write the reduced value to `YV` (here `y_val`) based on the corresponding *row index*. The presented reduction builds on a `LATENCY` of eight for the `fadd` which required eight cycles, and to ensure that none of the HLS streams contain leftover data at the end of the SpMV kernel, on the host before creating OpenCL buffers and transferring data to the FPGA's global memory, we apply padding to all input data structures to be multiples of `LATENCY` (here eight). With our approach, the optimised SpMV kernel is not bound to any specific matrix dimensions, for instance does also run on matrices with row and column numbers which are not multiples of eight but is still limited by individual buffer size and accumulated input buffer sizes.

VI. Integration with Morpheus

Morpheus is a C++header-only library that heavily relies on templates and meta-programming to enable certain polymorphic capabilities. It follows a functional design that separates the data structures (containers) from the functions (algorithms), with algorithms acting on containers. In order to provide support for the various hardware platforms and memory hierarchies, *Morpheus* adopts two notions of abstraction: 1) the *Execution Space*, which specifies where the code will be executed; and 2) the *Memory Space*, specifying where the data will *reside in memory*. *Morpheus* currently supports four executions spaces: 1) Serial (Sequential), 2) OpenMP (Multi-threaded), 3) CUDA (NVIDIA GPUs) and 4) HIP (AMD GPUs), with each execution space also acting as a separate backend. As a result, it is possible to use a single interface for each supported algorithm. By specifying the backend we want to run in, we also target a different execution space. In addition, *Morpheus* offers data management routines to effectively managing data transfers between the available memory spaces. To provide support for heterogeneous hardware, *Morpheus* adopts the *Host-Device* model, enabling data management functionality between different memory spaces.

The integration of specific optimisations, such as the ones proposed in Section IV in a pre-existing backend presents different challenges that needed to be overcome from the integration of a new backend as proposed in Section V. Below we provide a description of the challenges for each of the two approaches.

A. Optimisations

Morpheus performs compile-time introspection on the algorithm and by examining the backend, provided by the user as a parameter, identifies which algorithm implementation to dispatch every time. This means it is only possible to dispatch one and only implementation of the algorithm for every backend. Multiple implementations in a single backend can be chosen only by re-compiling the code.

The optimisations proposed at Section IV effectively constitute alternative algorithm implementations of the SpMV routine in the Serial backend. With the current state of *Morpheus*, the integration is only possible by specifying which version should be enabled using a compile-time flag. Of course future developments could move this decision at run-time, such that it will be possible to enable all versions, although this will introduce runtime overheads every time the algorithm is executed. However, this would only be possible if the performance improvements of selecting the most optimal version (given the sparsity pattern of the matrix) justify this trade-off.

The integration of the SpMV implementations for COO and CSR offered by ArmPL inside each of the equivalent *Morpheus* routines requires the data of the *CooMatrix* and *CsrMatrix* containers in *Morpheus* to be converted in *armpl_spmat_t* so that they will be passed to the ArmPL specific routines. Since the data layout of the *CooMatrix* and *CsrMatrix* follows the same requirements as the ones internally in ArmPL, the *armpl_spmat_t* handle is created in `ARMPL_SPARSE_CREATE_NOCOPY` mode, meaning that only the pointers to the data are passed instead of any actual copies. To avoid creating a new handle for the same matrix every time the SpMV multiplication is performed one workspace is created for each format acting as a Singleton [28]. In other words, every time the SpMV multiplication is executed with a new matrix the workspace is responsible to register the newly created handle and in future calls use that one instead of creating a new one. Another challenge that had to be resolved is the adaptation of the polymorphic behaviour of *Morpheus* containers to explicit ArmPL function calls that have the format and value type information embedded in the function name. An example of the adaptor call that had to be implemented that maps the polymorphic containers available in *Morpheus* to the explicit function call for creating the ArmPL handler of either a COO or CSR matrix using ArmPL is shown in Table I.

TABLE I: Adaptation of the polymorphic *Morpheus* behavior to explicit calls required by ArmPL.

Morpheus Format	Adaptor call	ArmPL call

`CooMatrix<double>`
`CooMatrix<float>`
	`create_coo<T>`	`armpl_spmat_create_coo_d`
		`armpl_spmat_create_coo_s`

`CsrMatrix<double>`
`CsrMatrix<float>`
	`create_csr<T>`	`armpl_spmat_create_csr_d`
		`armpl_spmat_create_csr_s`

After a successful porting of the ArmPL implementations in *Morpheus*, the port of the individual SVE implementations for each format as described in Section IV followed the same principles as described for ArmPL. In other words, since no handles were used in the SVE implementations, no workspaces were required. However, the SVE intrinsics used in the implementations had to be adapted in a similar way described for ArmPL.

B. New Backends

The integration of a new backend in any code is a very challenging task as each new backend poses unique challenges that might require significant development efforts. In this work, we seek to understand the challenges involved in enabling support for FPGAs in *Morpheus* for future releases. The motivation behind this choice is that FPGAs share many similarities with GPUs because, from the developers perspective, both are regarded as accelerators with distinct memory spaces from the host (CPU) in a *host-device model*. Hence, since *Morpheus* already provides supports for GPUs, the high-level interface for managing heterogeneous hardware could potentially remain unchanged.

The proposed process of support for FPGA in *Morpheus* is as follows:

1) *Develop the execution space:* In order to maintain portability of *Morpheus*, the *ExecutionSpace* concept available in *Kokkos* [29] can be followed in order to implement a common set of functionality that remains in line with the interface of the existing execution spaces.

Upon completion, *Morpheus* will be able to discover, acquire and synchronise with an FPGA that is available in the runtime and dispatch any algorithms implemented for the FPGA backend. Note that housekeeping routines from OpenCL might be required for querying the underlying FPGA runtime.

2) *Develop the memory space:* In a similar manner to the creation of the execution space, the memory space for the global memories that are available on FPGAs has to be developed. Following the *MemorySpace* concept available in *Kokkos*, two memory spaces can be created representing the DDR and High Bandwidth Memory (HBM2), memories that are commonly available in many FPGAs. Upon creation, using the two memory spaces users will be able to allocate/deallocate memory on the target FPGAs effectively creating any of the available containers on the FPGA.
3) *Data Management:* Following the development of memory spaces that can obtain memory on the FPGA, the data management routines for each of the new memory space must be implemented. These routines include *copy* and *mirroring* operations responsible for transferring data between the supported memory spaces and creating new containers with the same characteristics and memory allocation size in a specified memory space. In addition, this will allow for data to be offloaded from the Host (CPU) to the Device (FPGA) and vice-versa.
4) *Low-level Implementation:* The last step will be to implement the algorithms available in *Morpheus* (including SpMV multiplication) for the FPGA. The implementations at this level will be the ones launched when the FPGA execution space is selected. Note that both the housekeeping done with OpenCL as well as the kernel launch are implemented at this level.

The first three steps described above share many similarities with the existing backends in *Morpheus*. However, the most challenging stage to implement is the low-level implementation as this is where the unique properties of FPGAs are manifested. One major challenge is the vast difference in the build process of the device code. Generally, the code written for FPGAs is compiled to generate a bitstream containing the hardware configuration, which even for small kernels such as SpMV can take several hours, especially with increasing solution spaces for optimisations such as the partitioning of large arrays. Compared to other architectures, this reconfigurability comes at the expense of relatively long build processes including routing and placing on the die for FPGAs leading to longer time-to-solution. While building the host code on CPU for managing the interaction with the device is relatively fast compared to the overall bitstream generation, this means that on-the-fly compilation and linking is not possible. In addition, *Morpheus* is a header-only library that is effectively compiled at the application level. Consequently, all the algorithms are written in the form of templates that the compiler is responsible for generating at the application level i.e. in user's code. As a result, the type of the inputs is generic until last minute meaning we wouldn't been able to generate the FPGA bitsream until that point. To circumvent this issue, a set of potential types could be used to explicitly instantiate these implementations during the build and installation of *Morpheus*, such that it will be possible to generate the bitstreams *apriori*.

Another important challenge is portability across different FPGA devices both of the same or different vendors. Klaisoongnoen et al. [30] explored the porting of HLS kernels between AMD-Xilinx and Intel FPGAs and described the challenges encountered when moving from one FPGA architecture to another and suitability of optimisation techniques between vendor tool chains. For the SpMV kernels in HLS presented in this work, moving from one FPGA architecture to another requires that connectivity configurations, for instance mapping kernel arguments to memory spaces, are adapted. An important consideration is whether the target FPGA card provides HBM2 memory or whether alternatively on-card DDR has to be targeted and how this is integrated in the specific build process of the available HLS tooling. While the HLS code tends to be portable between FPGA architectures, vendor specific `pragmas`, host code implementations and support for features such as *direct host-kernel streaming*[2] vary. One possible solution to this issue is the creation of different backends for each vendor as well as implementing the algorithms and optimisations with backward compatibility in mind. Note that by explicitly instantiating the FPGA implementations as well as having information about the range of the supported FPGA devices can allow us to generate a set of bitstreams in advance and distribute them as part of the *Morpheus* release and each time load the appropriate pre-built implementation depending on the respective runtime.

Focusing on automatic code generation for multiple backends, a key challenge with FPGAs is the fundamental difference in how such devices operate compared to traditional *Von-Neumann* based CPU architectures. Whilst HLS toolchains such as AMD-Xilinx's Vitis HLS and Intel's Quartus Prime Pro typically generate working bitstreams for FPGAs from CPU-based codes, these bitstreams tend to be significantly slower in performance compared to manually tuned HLS code which has been adapted to suit a dataflow style of computing optimal for FPGAs. For instance, differences in runtime performance between naive CPU-based HLS kernels and *dataflow-optimised* implementations have been shown to range significantly (over 1000 times is common) [31]. Whilst available HLS tools provide portability within limitations, performance portability between CPU and FPGA architectures remains an open challenge. Our approach, as described above, is not based on automatic code generation but instead allows developers to write custom code for FPGAs with explicit optimisations. In addition, users can exploit additional optimisations through the dynamic format switching capabilities of *Morpheus* that would also be available for the FPGA backend.

[2] Intel FPGA Programming Guide, Direct Communication with Kernels via Host Pipes: https://www.intel.com/content/www/us/en/docs/programmable/683846/22-1/direct-communication-with-kernels-via.html

VII. Results and Evaluation

A. Setup

All experiments targeting AArch64 CPUs were carried out on the Bristol-based HPE Apollo 80 partition of Isambard [32]. Each of the 72 nodes on the cabinet has a Fujitsu A64FX Processor with 48 ARMv8.2 cores and 512-bit SVE, running at the clock frequency of 1.8GHz, and 32GB HBM2 memory arranged in 4 core memory groups.

Each experiment was compiled with *GNU 10.2.0* using `-O3 -ffast-math -ftree-vectorize -funroll-loops -mcpu=native` compiler flags. For the distributed experiments *OpenMPI 4.1.0* was also used.

For the FPGA runs reported in this paper we use a Xilinx Alveo U280, running at the default clock frequency of 300MHz, which contains an FPGA chip with 1.08 million LUTs, 4.5MB of on-chip BRAM, 30MB of on-chip UltraRAM, and 9024 DSP slices. This PCIe card also contains 8GB of HBM2 and 32GB of DDR DRAM on the board.

The FPGA card is hosted in the ExCALIBUR H&ES FPGA testbed[3] system with a 32-core AMD EPYC 7502 CPU with 256GB DRAM . All bitstreams are built for the U280 using Xilinx's Vitis framework version 2021.2.All reported results are averaged over ten runs and FPGA run-time includes on-device execution time exclusive device setup time and excluding data transfer times.

B. Evaluation of SpMV on AArch64 CPUs

In order to evaluate the performance of the newly added SpMV implementations for AArch64 CPUs in *Morpheus*, for each implementation we perform 100 iterations of the SpMV multiplication over 2106 sparse matrices available in SuiteSparse [7] collection. Each run is executed in Serial on a Fujitsu A64FX Processor, as described in Section VII-A. The implementations are divided in three versions as shown in Table II, along with a short description and the supported formats for each.

TABLE II: Versions of each CPU-based SpMV implementation available in Morpheus along with the formats each version supports.

Version	Description	COO	CSR	DIA
Plain	Original implementations without any Arm Optimisations	✓	✓	✓
ARMPL	Implementations using ArmPL	✓	✓	×
SVE	Implementations using SVE Extensions	✓	✓	✓

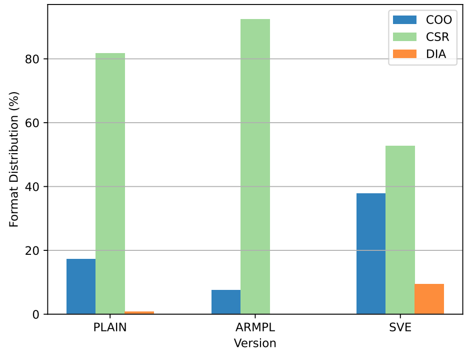
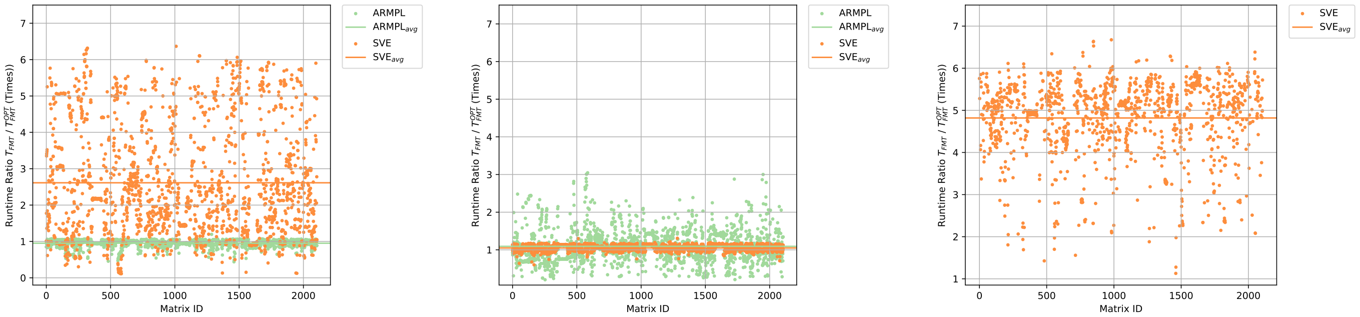


Fig. 3: Distribution of the optimal format for the SpMV multiplication operation in serial for over 2100 sparse matrices from SuiteSparse collection on A64FX. Distributions are shown for each version of the algorithm.

The optimal format distribution per version differs significantly, as shown in Figure 3. For most of the matrices in the SuiteSparse collection the optimal format is CSR, validating its role as the most commonly used storage format. However, almost 20% and 40% of the matrices are better with COO in the *Plain* and *SVE* versions respectively. Interestingly, although DIA format is almost of no use for *Plain* version, the vectorization performed by *SVE* version makes DIA format optimal for 10% of the matrices. This indicates that the vectorization performed by the compiler for DIA in *Plain* version might not be as effective as the use of custom SVE extensions in the *SVE* version. The main takeaway here is that for the same hardware, operation and set of matrices in the majority of times the optimal performance is given by CSR, although the distribution of the optimal format can vary significantly given a different implementation or by applying different optimisations.

Figure 4 shows the single-core performance of the SpMV multiplication for over 2100 sparse matrices from SuiteSparse collection on the A64FX processor. For each format, the runtime of the *Plain* version SpMV is compared against the runtime of each optimized SpMV version (*ARMPL* and *SVE*) using the same format. For COO (Figure 4a), *ARMPL* SpMV implementation performs at par with the *Plain* COO SpMV implementation whilst *SVE* implementation consistently outperforms it, obtaining average speedups of $1\times$ and $3.6\times$ respectively. The increase in performance achieved by the *SVE* version can be attributed in assumptions made during the implementation of the SpMV algorithm that allowed us to take advantage of different intrinsic commands. For example, by assuming that the matrix is sorted (which *Morpheus* ensures prior to applying any SpMV operation) a tree-based reduction was used instead of the traditional left-to-right reduction in order to accumulate the results in the output vector `y`. It is worth highlighting that even-though the *SVE* version significantly outperforms the *Plain* version for most of the matrices in COO, there is still a noticeable number of matrices for which it significantly under-performs. For very sparse and unstructured matrices, *SVE* version seems to introduce more overheads from the vectorization process effectively hindering the performance of SpMV. For CSR (Figure 4b), the average runtime performance for both *ARMPL* and *SVE* versions is at par with *Plain*. Interestingly, for a large number of matrices

[3] ExCALIBUR H&ES FPGA testbed, Field Programmable Gate Arrays (FPGAs) for accelerating scientific and data-science codes: https://fpga.epcc.ed.ac.uk/

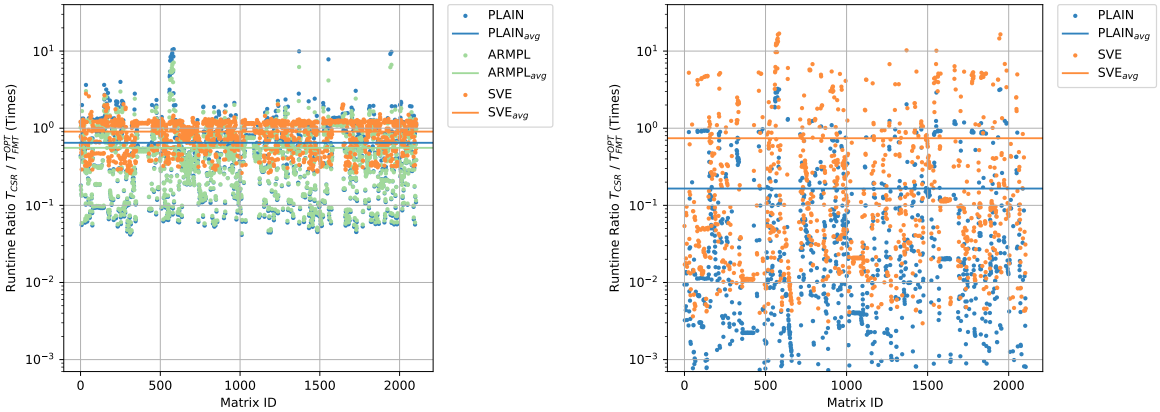


(a) COO

(b) CSR

(c) DIA

Fig. 4: Serial performance of the SpMV multiplication over 2100 sparse matrices from SuiteSparse collection on A64FX. For each format, the original performance (*Plain*) of the *Morpheus* SpMV is measured against the optimized (ARM) SpMV. Optimized versions include the *ArmPL* and *SVE* implementations and the formats considered are COO, CSR, DIA. A ratio above 1 indicates a speedup over the performance achieved when using the original implementation with the same format. The straight lines represent the average speedup over all matrices for each version.



(a) COO

(b) DIA

Fig. 5: Serial performance of the SpMV multiplication over 2100 sparse matrices from SuiteSparse collection on A64FX. For each format, the original performance (*Plain*) of the *Morpheus* CSR SpMV is measured against the optimized (ARM) SpMV. Optimized versions include the *ArmPL* and *SVE* implementations and the formats considered are COO, DIA. A ratio above 1 indicates a speedup over the performance achieved when using the original CSR implementation. The straight lines represent the average speedup over all matrices for each version.

the *ARMPL* version achieves speedups above 1× compared to *Plain*, with max speedup up to 3×. At the same time, for a large number of matrices it seems to significantly underperform, whilst *SVE* version offers a more stable performance profile. The largest benefit from exploiting the SVE intrinsics is reaped by DIA (Figure 4c) where the *SVE* version obtains an average speedup of $\approx 5\times$ compared to the *Plain* implementation. The fact that the *SVE* implementation beats the *Plain* implementation for all matrices in the set suggests that the compiler has a tough time performing effecting vectorization in *Plain* version.

Since for most of the matrices the optimal format for performing the SpMV operation is CSR, in Figure 5 we compare the runtime performance of COO and DIA for all three versions (*Plain*, *ARMPL* and *SVE*) against the runtime of the CSR for the *Plain* version, with the same configuration as before. For COO (Figure 5a), on average both *Plain* and *ARMPL* versions perform approximately the same and do worse compared to the *Plain* CSR implementation. On the other hand, the optimisations performed in the *SVE* version achieve an average performance at par compared to the *Plain* CSR implementation. Note that for most of the matrices, COO will result in significant slowdowns compared to CSR irrespective of the version used. However, for a small number of matrices, all three versions offer noticeable speedups reaching up to max speedups of 10×. Similar observations can be made about DIA (Figure 5b). It is obvious that DIA format finds use in a small number of matrices with specific characteristics. However, for those matrices the performance optimisations from the adoption of SVE intrinsics can offer a significant boost in performance with max speedups of $\approx 20\times$ compared to CSR.

It is worth pointing out, that even-though it was expected for the *ARMPL* versions to perform optimally for COO and CSR, on average they were at par with their *Plain* equivalents. However, for a large number of matrices, the CSR *ARMPL* implementation was optimal. Furthermore, the adoption of SVE intrinsics had a noticeable impact on DIA, increasing the number of matrices it was optimal for by an order of magnitude and offering consistently noticeable speedups compared to it's equivalent *Plain* implementation. This experiment makes it clear that in the same way no single format can perform best across the different sparsity patterns, no single implementation can do the same either. As a result, these findings motivate the extension of *Morpheus* to also support an efficient mechanism for selecting the optimal implementation at runtime.

C. Evaluation of SpMV on FPGAs

When evaluating our FPGA implementations of SpMV, we performed 10 iterations of each of the three SpMV kernels over the SuiteSparse collection, with Section VII-A providing more details on the experimental setup. Figure 7 shows the distribution of optimal formats for the SpMV operation in Serial for over 2100 sparse matrices. The CSR format dominates in terms of performance or shortest runtime, since for more than 80% of the matrices performs optimally when compared against the COO and DIA format. The second most optimal format across the SuiteSparse matrices is COO with more than 10% and DIA is optimal still for more than 5% of the matrices. It is worth highlighting that on the A64FX processor, similar distribution was obtained for the *Plain* SpMV Version as shown in Figure 3.

The serial performance of our FPGA prototypes for COO and DIA against CSR is presented in Figures 6a and 6b, confirming the conclusions from the optimal format comparison where the SpMV algorithm for CSR outperforms both the COO and DIA implementations on FPGA for the majority of matrices. Note that even-though the average speedup for both COO and DIA is well below 1 when compared against CSR, for a very few matrices we do observe some increase in performance (for instance, in one case with COO we observe a 2× speedup). Optimising the kernels further such that they are exploiting optimally the characteristics of the hardware can have the potential of a more diverse distribution of optimal formats. However, at this point with COO and DIA both yielding no speedup over CSR, we can report that the compressed sparse row advantage on traditional architectures also holds true for our baseline versions on FPGA.

In Figure 6c the optimisations of the COO SpMV kernel described in Section V are compared against the *naive* version. We observed that the use of HBM2 provided a small boost in performance for most of the matrices, but combining HBM with on-chip BRAM had no further noticeable effect. The *reduce* operation, shown in Figure 2, is sub-optimal for FPGAs because the floating point accumulation between iterations, which requires more than one cycle, adds a spatial dependency between loop iterations and thus pushes the initiation interval higher than one, meaning that cycles are wasted. It is common practice on FPGAs to optimise this by reducing in chunks of independent iterations, and performance numbers for this are reported in Figure 6c by *REDUCE*. However it can be seen that, in this case, this is not beneficial and that is because the conditional on *row index* means that the cycles which do not match the conditional are *doing nothing* anyway, and-so this offsets the added complexity of the reduction optimisation. However, it is worth highlighting that it achieved the highest maximum speedups out of all three optimisations for one of the matricies, where we observe that the reduce optimisation works better for small matrices as the overhead increases with larger `nrows` and `NNZ`. This result demonstrates once more, this time on FPGAs, the importance of having multiple implementations of the same SpMV operation and the ability to dynamically switch to the optimal format given the sparsity pattern of the input.

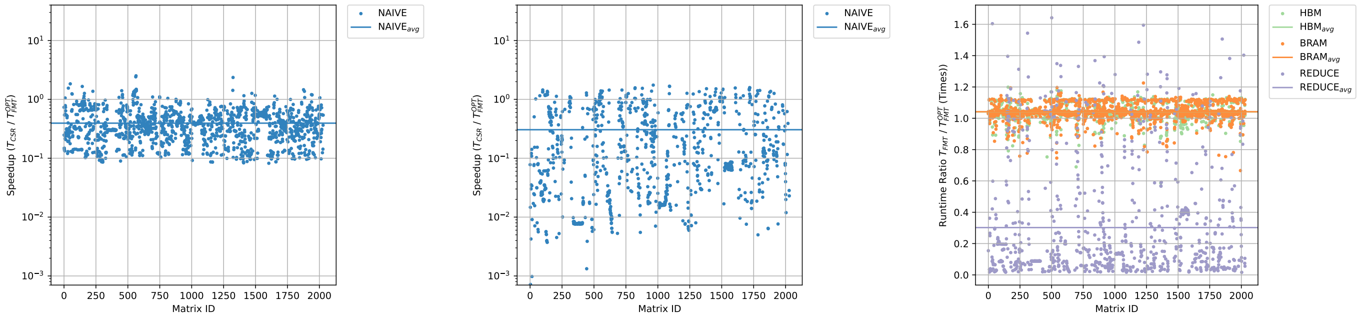
D. HPCG experiments

The HPCG benchmark solves the Poisson differential equation on a regular 3D grid, discretized with a 27-point stencil. It uses the Preconditioned Conjugate Gradient (PCG) algorithm with a symmetric Gauss-Seidel [26] as a preconditioner, and includes the following computations: sparse matrix-vector multiplications (SpMVs); vector updates; global dot products; a local symmetric Gauss-Seidel smoother (including a sparse triangular solve); and multi-grid (MG) preconditioned solvers. The performance bottleneck in HPCG is due to to the sparse operations that are carried out at every step of the iterative solver, i.e the SpMV and the Gauss-Seidel smoother.

HPCG is a widely accepted and well-understood benchmark used to measure the performance of HPC systems. For this reason, multiple vendor specific implementations exist ([10], [33], [34]). In previous work, we have also implemented a *Morpheus*-enabled HPCG [9]. The benchmark progresses in the following phases:

1) *Problem setup*: Constructs the synthetic problem by creating the geometry and linear system.
2) *Reference timing*: Measures the time taken to run the SpMV and MG reference implementations and the time to solution for the reference Conjugate Gradient (CG) solver.
3) *Problem Optimisation setup*: Configures the user defined data structures to be used in the optimised problem.
4) *Validation and Verification*: Checks that the optimised problem has returned the expected results.
5) *Optimised problem timing*: Measures the time to solution for the optimised CG solver.

In the following experiment, we benchmark the performance of: 1) the *Morpheus*-enabled HPCG with the newly added *ARMPL* and *SVE* versions of SpMV and 2) The vendor (Arm) implementation of HPCG against the original HPCG. We are focusing on Phase 5, although for the purposes of this work, since we are interested in the SpMV multiplication, we are disabling the use of the preconditioner from all implementations.



(a) Plain COO SpMV performance against CSR implementation.

(b) Plain DIA SpMV performance against CSR implementation

(c) Optimised versions of COO SpMV against plain COO version.

Fig. 6: Serial performance of the SpMV multiplication over 2100 sparse matrices from SuiteSparse collection on Alveo U280. A ratio above 1 indicates a speedup over the performance achieved against a reference implementation. The straight lines represent the average speedup over all matrices for each version.

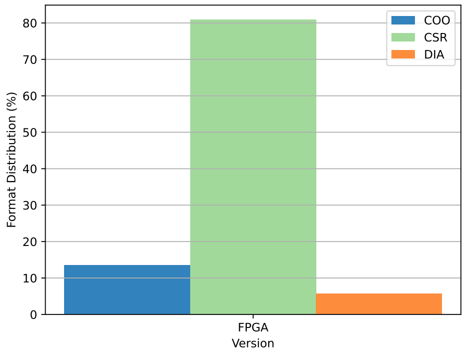


Fig. 7: Distribution of the optimal format for the SpMV multiplication operation in serial for over 2100 sparse matrices from SuiteSparse collection on Alveo U280. Distributions are shown for each version of the algorithm.

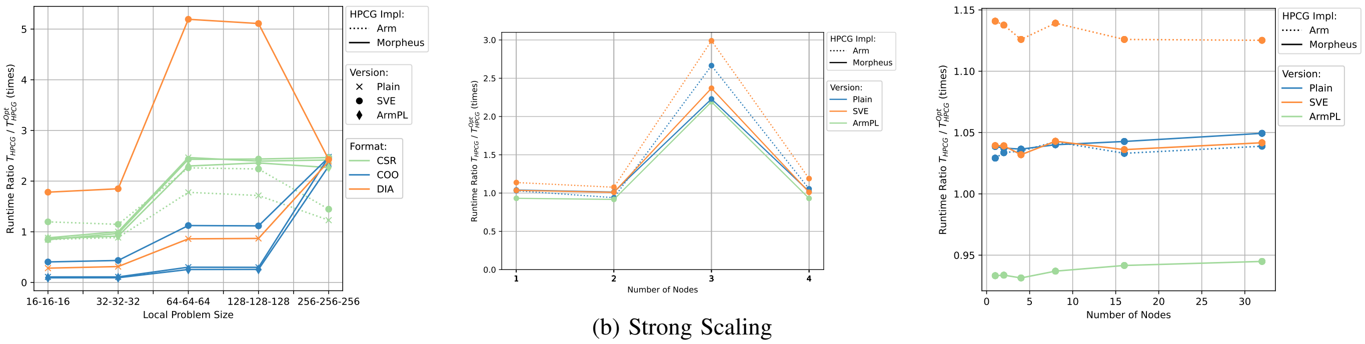
The experiment is configured as described in Section VII-A on the A64FX processors.

In Figure 8a, the single-core SpMV runtime performance of the two HPCG implementations (*Morpheus* and *Arm*) is measured against the original HPCG over a set of different problem sizes. For each format in the *Morpheus*-enabled implementation, we measure the runtime for the *Plain*, *ARMPL* and *SVE* versions of the SpMV multiplication routines. In a similar way, the runtime of the SpMV with (*SVE*) and without (*Plain*) SVE intrinsics is measured for the Arm implementation.

The system matrix in HPCG is generated using the FDM. As a result, the matrix is highly regular with non-zeros around the diagonals. It is expected therefore that the DIA format would perform optimally compared to the rest of the formats, a hypothesis which is confirmed by Figure 8a. The optimal performance difference between the two versions of the DIA SpMV i.e *Plain* and *SVE*, follows the average performance observed in Figure 5b, with a max speedup of $5\times$ compared to the reference HPCG. Note that for smaller problem sizes, the performance of both is impacted by the extra operations due to zero-padding. The *SVE* version for the Arm HPCG closely follows the performance of the *Morpheus*-enabled HPCG that uses CSR. However, the performance of the Arm HPCG version without *SVE* support diminishes at the problem size of 64^3. This can be attributed to the fact the matrix in Arm HPCG is reordered and to the lack of SVE extensions. Interestingly, for a problem size of 256^3 every implementation that was performing better, compared to the reference, now either sees a drop in performance or stays at par. However, implementations such as all versions of the *Morpheus*-enabled HPCG using COO and the *Plain* version that uses DIA now see a boost in performance, with the *SVE* version of COO achieving the optimal –but marginal– performance out of all. This result further motivates the need for runtime switching of different SpMV implementations for the same format.

The performance of the distributed HPCG for both Morpheus- and Arm-enabled HPCG is measured against the original HPCG implementation. For the strong scaling experiment the global problem size chosen is $192 \times 256 \times 192$ and for the weak scaling experiment the local problem size is $48 \times 64 \times 64$. For the distributed implementations the sparsity pattern of the matrix on each process differs from the one in the Serial case, due to the remote elements of the matrix added to the right. Whilst the matrix is initially structured, the remote part of it is highly unstructured. As a result, in the *Morpheus*-enabled HPCG we physically split this matrix into *local* and *remote* part in order to potentially select different storage formats for each. This is achieved by utilising a run-first auto-tuner where it finds the optimal format to use on every process. The formats selected on each process for each version of our implementation are shown in Table III. Notice that for the *SVE* version, the optimal formats chosen were DIA and COO, for both strong and weak scaling experiments.

Figures 8b and 8c shows the scaling SpMV performance for each version of *Morpheus*- and Arm-enabled HPCG implementations against the reference HPCG. Note that for both



(a) Serial Performance

(b) Strong Scaling

(c) Weak Scaling

Fig. 8: Performance of the Morpheus- and Arm-enabled HPCG implementations. The performance is measured as the SpMV runtime ratio of the reference HPCG w.r.t each optimal HPCG implementation and version on A64FX. For the serial version, High Performance Conjugate Gradients (HPCG) runs for a set of problem sizes. For the distributed Morpheus-enabled HPCG a run-first auto-tuner is used in order to determine the optimal format to use at each process. A ratio above 1 indicates a speedup over the performance achieved when using the original HPCG.

TABLE III: Optimal format used for each version of the *Morpheus*-enabled HPCG across the different processes for both strong and weak scaling.

Version	Local Format	Remote Format
Plain	CSR	CSR
ARMPL	CSR	CSR
SVE	DIA	COO

strong and weak scaling, the *Morpheus*-enabled HPCG closely tracks the performance of the optimal *SVE* version of Arm-enabled HPCG. In Figure 8b, the boost in performance on 3 nodes happens due to the fact the size of the system matrix strikes a balance between the benefits achieved from vectorisation and the overheads that are associated with it. Furthermore, the weak scaling results in Figure 8c, eventhough they show marginal improvement over the original HPCG, still show that our contributions match the performance of the optimal Arm implementation. Note however, with a local problem size somewhere closer to the region we have previously noticed performance improvements (i.e between 64^3 and 128^3 as shown in Figure 8a) more noticeable speedups would have been expected, although due to memory limitations such runs weren't feasible at the distributed level.

VIII. Related Work

The research efforts into optimising sparse computations largely fall in two categories, namely (i) in the creation of novel storage formats that better capture the characteristics of particular architectures or sparsity patterns, such as CSR5 [3] and Sell-C-σ [5], and (ii) in the use of auto-tuners such as Morpheus-Oracle [35] and SMAT [36] to automatically determine the optimal format to use for a computation.

A few recent works have examined the performance of different sparse formats on AArch64 targets. In particular on A64FX processors, the SELL-C-σ matrix storage has been shown to achieve performance and memory-bandwidth saturation superior to the standard CSR format, reaching performance on par with NVIDIA's V100 GPUs for large, memory-bound SpMV datasets [37]. An optimised variation of the CSR format—"Bitmap-based CSR (BCSR)"—that extracts edge information more efficiently and has a smaller memory footprint than the classical CSR format has also been proposed for the A64FX-based Fugaku supercomputer, enabling it to reach rank 1 in the Graph500 benchmark in November 2020 [14]. Other variations of the CSR and ELLPACK (ELL) formats, namely aligned CSR (ACSR) and aligned ELL (AELL), respectively, have also been proposed and implemented with Neon instructions for AArch64 targets [38].

For sparse matrix multiplications, FPGA vendors such as Intel implemented, among others, the COO, CSR and DIA formats in their Sparse BLAS libraries as part of the Intel oneAPI Math Kernel Library [39], whereas Xilinx have developed an implementation of the CSC format in their Vitis Sparse libraries [40] and the COO format in their General Matrix Operation (GEMX) [41] engine library which provides building blocks for constructing matrix operation accelerators on FPGA. Other work focused on implementing a modified CSR (MCSR) format for SpMV multiplication in HLS [42], or customised sparse matrix formats to leverage the available HBM2 on recent generation FPGAs [43].

IX. Conclusions and Further Work

In this paper, we have shed light on the challenges implementing prototypes of the SpMV kernels for the three sparse storage formats COO, CSR and DIA on the two emerging architectures AArch64 CPUs and FPGAs. Optimising the three kernels on the respective architectures, we highlight the performance advantages of individual approaches and show that our Morpheus implementations are competitive. Moreover, we describe potential integration targets of our prototypes to be implemented in the Morpheus library. While our results prove performant implementations especially on AArch64 CPUs with SVE and compared to ARMPL, accelerators such as

FPGAs exhibit larger performance portability gaps compared to architectures that do not build on a host-device model.

In terms of future work, a full integration of the FPGA prototype as novel backend to Morpheus is of interest but will require further engineering around abstractions for memory management, build process integration, smart container/layer translations due to restricted availability of dynamic memory and data type support. As vendors such as Intel and AMD-Xilinx have come up with architecture specific implementations of HPCG, an evaluation of these against the Morpheus-HPCG version will be possible, especially with AMD-Xilinx's closely to the HPCG problem size generation tied CSR implentation on FPGA. Moreover, the FPGA prototypes of the storage formats could benefit from implementation on the newest generation of AMD-Xilinx FPGAs, the Versal ACAP.

While our focus in this paper is on the three core sparse matrix storage formats supported by Morpheus i.e. COO, CSR and DIA, there exist a plenitude of other storage formats such as ELL, HYB or HDC that are widely used and from which the Morpheus extensions on AArch64 CPUs and FPGAs could provide further benefit to HPC users.

In terms of theoretical evaluation, future work includes the application of a roofline model to understand and validate the performance of our implementations and compare theoretical performance on our target systems to achieved performance.

Of interest is also the multi-threaded (OpenMP) implementation of the current formats with SVE intrinsics and benchmark on other Arm systems. Finally, extending *Morpheus* to support a dynamic selection and dispatch mechanism that adapts to the optimal algorithm given a sparsity pattern could be beneficial.

Acknowledgment

This research is part of the EPSRC project ASiMoV (EP/S005072/1). We used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1). We also acknowledge the ExCALIBUR H&ES FPGA testbed and AMD Xilinx HACC program for access to compute resource used in this work.

References

[1] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse Matrix-Vector Multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3017994

[2] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 94–108. [Online]. Available: https://doi.org/10.1145/3178487.3178495

[3] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 339–350. [Online]. Available: https://doi.org/10.1145/2751205.2751209

[4] E. Coronado-Barrientos, M. Antonioletti, and A. Garcia-Loureiro, "A new axt format for an efficient spmv product using avx-512 instructions and cuda," *Advances in Engineering Software*, vol. 156, p. 102997, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997821000260

[5] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, 07 2013.

[6] C. Stylianou and M. Weiland, "Exploiting dynamic sparse matrices for performance portable linear algebra operations," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov 2022, pp. 47–57. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/P3HPC56579.2022.00010

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[8] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016. [Online]. Available: https://doi.org/10.1177/1094342015593158

[9] "Morpheus-HPCG Benchmark," Morpheus, Nov. 2022. [Online]. Available: https://github.com/morpheus-org/morpheus-hpcg

[10] "HPCG for Arm," Arm Software, Nov. 2022. [Online]. Available: https://github.com/ARM-software/HPCG_for_Arm

[11] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc Web page," https://petsc.org/, 2023. [Online]. Available: https://petsc.org/

[12] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing," *ACM Transactions on Mathematical Software*, vol. 48, no. 1, pp. 2:1–2:33, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3480935

[13] A. Jackson, M. Weiland, N. Brown, A. Turner, and M. Parsons, "Investigating Applications on the A64FX," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. Kobe, Japan: IEEE, Sep. 2020, pp. 549–558.

[14] M. Nakao, K. Ueno, K. Fujisawa, Y. Kodama, and M. Sato, "Performance of the Supercomputer Fugaku for Breadth-First Search in Graph500 Benchmark," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, vol. 12728, pp. 372–390.

[15] R. Jesus, T. Oliveira e Silva, and M. Weiland, "Vectorising and distributing NTTs to count Goldbach partitions on Arm-based supercomputers," May 2021, Cray User Group (CUG) 2021.

[16] Masoud Koleini, "Graviton3 outperforms x86 on Machine Learning," Oct. 2022. [Online]. Available: https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/xgboost-lightgbm-aws-graviton3

[17] "Arm Architecture Reference Manual Supplement, The Scalable Vector Extension," Arm Limited, DDI 0584B.a, May 2021.

[18] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[19] "Arm Performance Libraries Reference Guide," Arm Limited, Issue 2210-00, Sep. 2022.

[20] N. Brown, "Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations," in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2021, pp. 9–20.

[21] N. Brown, "Accelerating advection for atmospheric modelling on Xilinx and Intel FPGAs," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 767–774.

[22] N. Brown, M. Klaisoongnoen, and O. Brown, "Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*.

United States: Institute of Electrical and Electronics Engineers (IEEE), Oct. 2021.

[23] M. Klaisoongnoen, N. Brown, and O. Brown, "I Feel the Need for Speed: Exploiting Latest Generation FPGAs in Providing New Capabilities for High Frequency Trading," in *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, ser. HEART '21. New York, NY, USA: Association for Computing Machinery (ACM), 2021.

[24] M. Klaisoongnoen, B. Nick, and O. Brown, "Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs," in *HEART2022: International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. United States: Association for Computing Machinery (ACM), Jun. 2022.

[25] N. Brown, "Exploring the Versal AI engines for accelerating stencil-based atmospheric advection simulation," in *31st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2023.

[26] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf

[27] "Arm C Language Extensions," Arm Limited, Version 2022Q4, Nov. 2022.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[29] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[30] M. Klaisoongnoen, N. Brown, and O. Brown, "Fast and energy-efficient derivatives risk analysis: Streaming option Greeks on Xilinx and Intel FPGAs," *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, Dec. 2022.

[31] N. Brown and D. Dolman, "It's All About Data Movement: Optimising FPGA Data Access to Boost Performance," in *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2019, pp. 1–10.

[32] GW4 and UK Met Office, "Isambard 2 UK National Tier-2 HPC Service," 2022. [Online]. Available: http://gw4.ac.uk/isambard/

[33] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2015, pp. 68–84.

[34] A. Zeni, K. O'Brien, M. Blott, and M. D. Santambrogio, "Optimized implementation of the hpcg benchmark on reconfigurable hardware," in *European Conference on Parallel Processing*. Springer, 2021, pp. 616–630.

[35] C. Stylianou and M. Weiland, "Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning," in *Eighteenth International Workshop on Automatic Performance Tuning (iWAPT2023)*. St. Petersburg, FL, USA: IEEE Computer Society, (to appear).

[36] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 117–126. [Online]. Available: https://doi.org/10.1145/2491956.2462181

[37] C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig, "Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, Sep. 2022.

[38] Y. Zhang, W. Yang, K. Li, D. Tang, and K. Li, "Performance analysis and optimization for SpMV based on aligned storage formats on an ARM processor," *Journal of Parallel and Distributed Computing*, vol. 158, pp. 126–137, Dec. 2021.

[39] Intel. Sparse Matrix Storage Formats. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/appendix-a-linear-solvers-basics/sparse-matrix-storage-formats.html

[40] Xilinx. (2019) Sparse Libraries, Vitis Accelerated Libraries. [Online]. Available: https://github.com/Xilinx/Vitis_Libraries/tree/main/sparse

[41] Xilinx. General Matrix Operation (GEMX). [Online]. Available: https://github.com/Xilinx/gemx

[42] M. Hosseinabady and J. L. Nunez-Yanez, "A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1272–1285, 2020.

[43] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery (ACM), 2022, p. 54–64. [Online]. Available: https://doi.org/10.1145/3490422.3502368