MORPHEUS UNLEASHED: FAST CROSS-PLATFORM SPMV ON EMERGING ARCHITECTURES

Christodoulos Stylianou, Mark Klaisoongnoen, Ricardo Jesus, Nick Brown, Michèle Weiland

EPCC, The University of Edinburgh

epcc

Introduction

- Sparse matrices essential concept in computational science and engineering
- Sparse matrix storage formats are different in-memory representations of sparse matrices
  - Each designed to exploit strengths of the different hardware architectures or sparsity pattern of the matrix
- More than 70 formats have been developed over the years - still no single one performs best across:
  - Different sparsity patterns
  - Different target architectures
  - Different operations
- Most code-bases today still use a single format (CSR)
  - Adapting the data structure at run-time offers new optimization opportunities

Sparse Matrix Storage Formats

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

(a) Dense Matrix

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 Representation

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 Representation

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

(d) DIA Representation

epcc

Emerging & Non-traditional Architectures

1. AArch64 CPUs
   - Newcomers in the HPC scene
   - Have already proven competitive against traditional x86 CPUs.
   - Scalable Vector Extension (SVE) key enabler for high performance
     - Compiler support often lacking

2. Field Programmable Gate Arrays (FPGAs)
   - Allow for hardware reconfiguration tailored to the code
     - Provide configurable logic components and interconnect
   - Historically lacking mature development ecosystem
     - Hardware and software ecosystems have become more capable
     - FPGAs can now be programmed using High-level Synthesis (HLS) in C/C++
   - Best performance obtained by reworking the algorithm into a dataflow style

Motivation

- New formats are proposed every time a new architecture emerges
  - Aim to exploit the new characteristics and features of the new hardware
- Switching formats dynamically offers new opportunities for optimisation and increased performance
- Adopting a new format can be a tedious process as it requires significant code changes
- Libraries offer multiple formats through various abstractions, capable for runtime switching
  - e.g. PETSc, GINKGO and Morpheus
- When it comes to the adoption of new hardware, libraries often require major changes in the interface
  - e.g. to support a new programming model
- Need to ensure software can adapt to the requirements of new hardware architectures

SpMV on AArch64 CPUs

- From an application programmer's perspective there are two main ways of optimising software for AArch64 targets:
  - Using target-specific libraries that implement core algorithms and routines efficiently for the targets
  - Writing efficient code targeting specific CPU (micro-)architectures, usually through **intrinsics** or **assembly**

- Each of these strategies has its advantages and downsides
  - **Libraries** → easy to use as they do not require writing target-specific code explicitly
    - But limited by the methods and functionality that they offer
  - **Intrinsics** (and assembly) → require a far deeper knowledge of the targets' specificities and of the methods being implemented
    - But do not pose limitations to what can be implemented

- We have explored the usage of these **two approaches** to augment Morpheus with AArch64-optimised SpMV routines
  - The same techniques can be utilised for other targets (e.g. x86)

SpMV on AArch64 CPUs with ArmPL

- The **Arm Performance Libraries** (ArmPL) are a set of core routines developed by Arm for HPC applications for AArch64 targets
  - BLAS, LAPACK, FFT, Sparse, libamath (a subset of libm) and libastring (a subset of libc for strings) routines
    - For single- and multi-threaded programs provided via both C and Fortran interfaces
  - Sparse routines support dense, CSR, CSC, COO and BSR matrices via an API (similar to FFTW)
    - The description of the problem is independent of its execution

- Interface for the sparse methods:
  - `armpl_spmat_create_*` – create a handle to a sparse matrix
  - `armpl_spmat_hint` – provided to attempt to speedup future SpMV calls
  - `armpl_spmv_optimize` – issue optimisation stage where the library tries to determine the best algorithms and implementations for the specific matrix and target
  - `armpl_*_exec_*` – issue SpMV and other sparse algebra computations
  - `armpl_spmat_destroy` – destroy the handle

- Key change in Morpheus: Add a workspace that keeps track of the handles created for each matrix and use it to issue SpMV calls

SpMV on AArch64 CPUs with SVE and ACLE (I)

- The **Scalable Vector Extension** (SVE) is one of the most disruptive extensions of the AArch64 architecture for HPC
  - Vector extension that, unlike other single instruction multiple data (SIMD) extensions such as Neon and the AVX extensions, is "vector-length-agnostic" (VLA)
  - In SVE, the length of the vector registers is not known at compile time
  - This makes the extension highly portable across SVE implementations of varying vector widths
  - Key HPC and ML features:
    - Per-lane predication (i.e. control on a per vector element basis)
    - Gather-loads and scatter-stores
    - Speculative vectorisation
    - Horizontal and tree-based reductions

- **Arm C Language Extensions** (ACLE) are a set of compiler intrinsics that aim to:
  - Expose advanced features of the Arm architecture
  - Enable the development of applications and libraries portable across compilers and Arm micro-architectures
  - ACLE can be leveraged to write portable SVE code

SpMV on AArch64 CPUs with SVE and ACLE (II)

- We have used ACLE to implement SVE-enabled SpMV kernels for COO, CSR and DIA matrices
  - They result mostly from a transliteration of the "schoolbook" algorithms
  - We highlight the two main nuances in our implementations for COO and DIA

- ***COO:***
  - Indirection in output vector hinders vectorisation
    - Addressed by leveraging SVE's predication and reduction features to work only on element that write to the same output index
    - This strategy seems to lead to significant speedups over compiler generated and ArmPL implementations

```
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);
}
```

- ***DIA:***
  - Inner loop tends to have a short trip count
    - Therefore, vectorisation is often not advantageous
    - Vectorised the *outer loop* instead, leading to:
      - Better cache utilisation (multiple cache lines can be filled at once)
      - Avoids horizontal reductions prior to writing to the output vector

```
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);
}
```

epcc

Porting code on FPGAs

- FPGAs provide a very large number of **configurable logic** components sitting **within a sea of configurable interconnect**
- Modern FPGAs also contain **hardened components**: BlockRAM (BRAM), High Bandwidth Memory (HBM2), DDR, and high-performance networking capabilities
- **A major challenge** with FPGAs: The historically **significant time investment required in programming the technology** and **need for detailed hardware-level knowledge** on behalf of developers
- Recently FPGA hardware and software development ecosystems have become far more capable:
  - High-Level Synthesis (HLS) toolchains such as Intel's Quartus Prime and Xilinx's Vitis
  - software developers can now program FPGAs by writing code in C or C++ using HLS

Programming FPGAs is now becoming more a question of software development than hardware design

SpMV on FPGAs (i)

- For each supported format (COO, CSR and DIA) we implement a kernel to be loaded on FPGA and deliver three different bitstreams
  - Each kernel is a direct transliteration of the original SpMV algorithms for each format
- Each bitstream configures the FPGA
- Host code to manage data transfers and kernel launching on the FPGA is done using OpenCL
- The HLS kernels are set for AMD-Xilinx Alveo U280 FPGA, using the AMD-Xilinx HLS toolchain (Vitis)

- Each high-level function follows the host-device model:
  1. Initialise the device in the host code
  2. Create the OpenCL buffers for input/output data
  3. Transfer the required input matrix and vector data on device
  4. Execute the kernel on device
  5. Transfer results back to host

SpMV on FPGAs (ii)

- FPGAs operate fundamentally different from traditional *Von-Neumann* architectures and algorithms have to be reworked into a *dataflow style.*
- The *dataflow style* is built around concurrently running stages (dataflow stages) that stream data between themselves and each stage comprises individual pipeline(s).
- This approach provides the potential to implement custom optimization techniques around memory accesses and data transfers, resulting in lower number of cycles before a result is produced.
- An example of such dataflow structure is shown in Figure 1 for COO kernel:
  - Purple box: Depict a connection to external high bandwidth memory where all reads and writes are packed in chunks of 512 bits.
  - Green box: A separate dataflow region running concurrently
  - Solid Arrows: Represent the streams of data that flow from one cycle to the next.
  - Dashed Arrow: Represents a ping-pong buffer (double buffering technique)
    - One cycle will concurrently write to one buffer whereas the subsequent stage is served with data from a previous copy of the buffer.
    - Switch occurs at a predefined point.

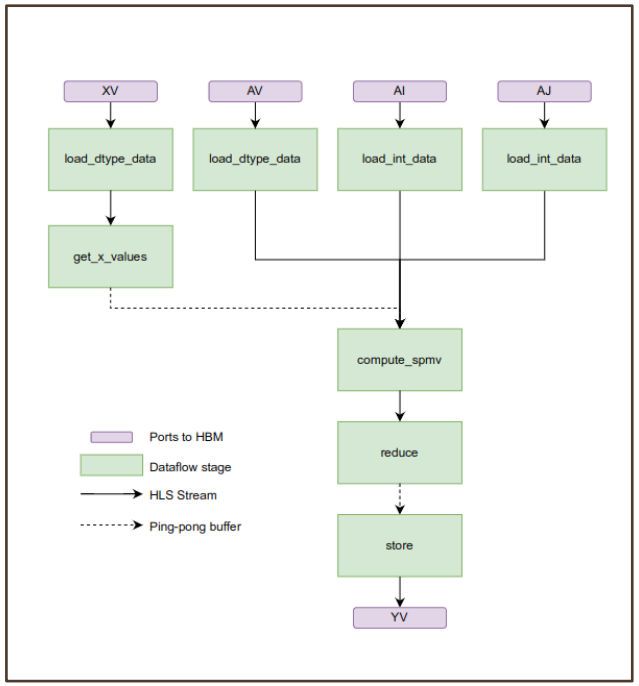


Figure 1: Separate dataflow region **running concurrently** to load the data from HBM2 and then **pass individual data elements to the next stages**, for COO SpMV kernel.

Integration

- *Morpheus* is a C++ header-only library that:
  - Provides an abstraction of sparse matrices and allows for efficient dynamic and transparent switching of formats
  - Currently supports *COO, CSR* and *DIA* and can target *Serial, OpenMP, CUDA* and *HIP* Backends.

- Proposed **Arm Optimizations** are integrated into **existing backends** using compile-time flags
  - ArmPL implementations for COO and CSR
  - SVE implementations for COO, CSR and DIA.

- Integration of new backends poses unique challenges that might require significant development efforts
  - and **changes to the existing interface**
- FPGAs, from the developer's perspective, fit in the *host-device model* as **accelerators**
  - Morpheus already supports GPUs *as a device*: potential for high-level interface to remain unchanged
  - Integration possible by developing an *FPGA* execution and memory space to be used by *Morpheus.*
- Challenges in the integration manifested in **the low-level implementation** of the algorithms:
  - Bitstream generation
  - Performance portability across different FPGA devices

Experimental Setup

- For AArch64:
  - HPE Apollo 80 partition on Isambard (hosted by Bristol)
    - 72 Nodes each with a Fujitsu A64FX processor (48 ARMv8.2 cores and 512-bit SVE)
    - 32GB HBM2 memory.
  - Compiler: GNU 10.2.0
  - Flags: `-O3 -ffast-math -ftree-vectorize -funroll-loops -mcpu=native`
  - For distributed experiments: OpenMPI 4.1.0

- For FPGAs:
  - ExCALIBUR H&ES FPGA testbed (hosted by EPCC in Edinburgh)
    - Xilinx Alveo U280
    - 32-core AMD EPYC 7502 CPU
    - 256GB DRAM
    - 8GB HBM2 and 32GB DDR DRAM on board.
  - Xilinx Vitis Version 2021.2

Evaluation of SpMV on AArch64 CPUs

- For each implementation we perform 100 iterations of SpMV multiplication
- Optimal format distribution per version differs significantly
- For most matrices in SuiteSparse CSR is optimal
- Almost 20% and 40% of matrices better off with COO for Plain and SVE implementations
- DIA almost never used by the Plain version
- Vectorization performed by SVE version makes DIA optimal for 10% of the matrices.

For the same hardware, operation and set of matrices, in the majority of cases CSR is optimal. Distribution of the optimal format can vary significantly given a different implementation or different optimizations.

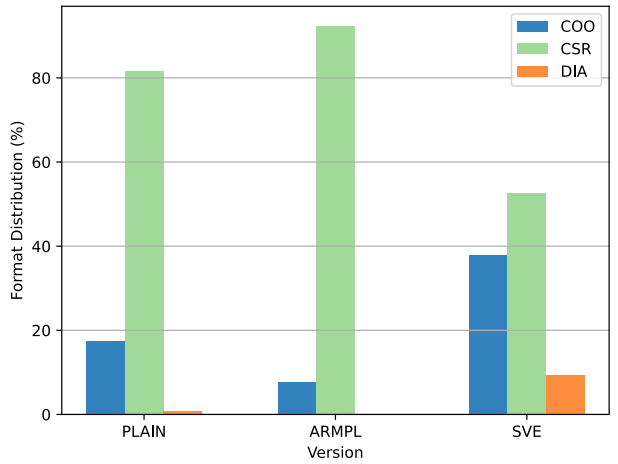
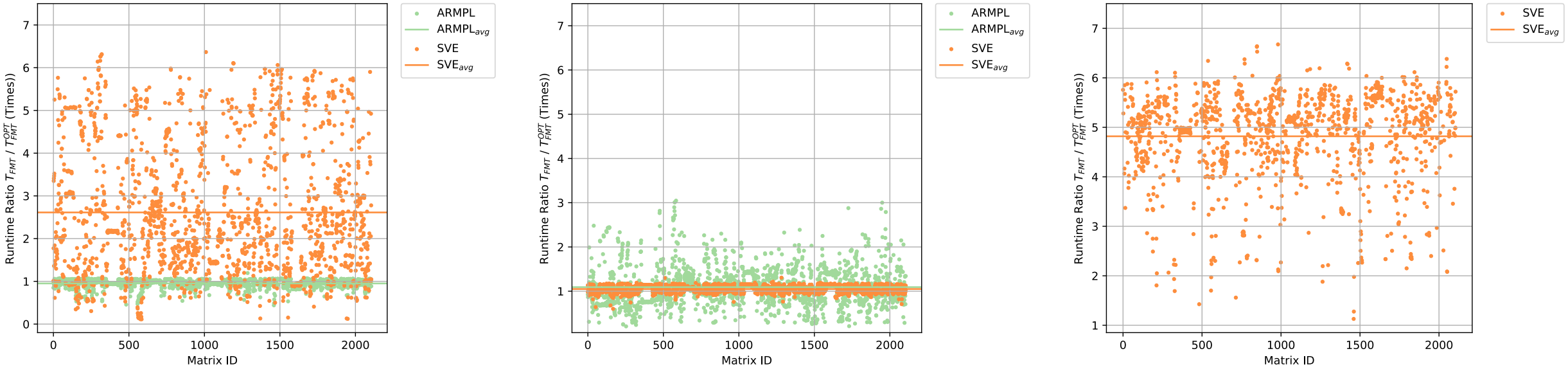


Figure 2: Distribution of the optimal format for the SpMV multiplication operation in serial for over 2100 sparse matrices from SuiteSparse collection on A64FX.

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

Table 1: Versions of each CPU-based SpMV implementation available in Morpheus along with the formats each version supports.

Evaluation of SpMV on AArch64 CPUs



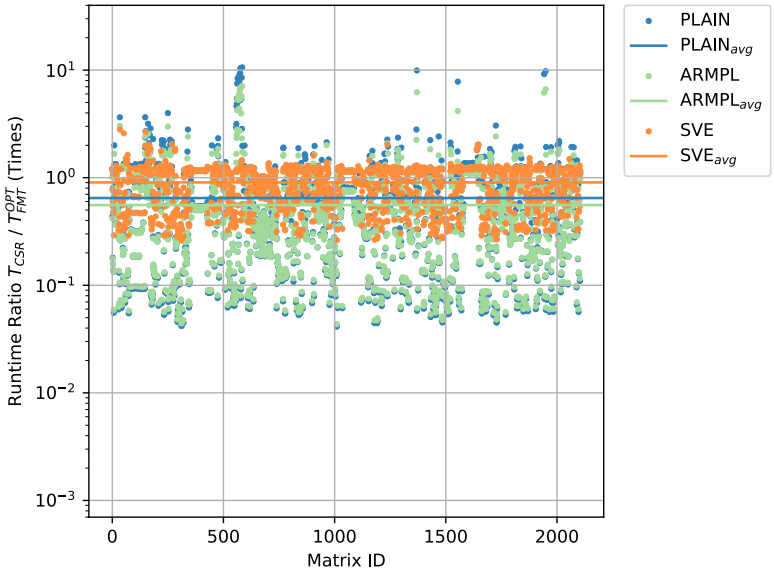
(a) COO (b) CSR (c) DIA

Figure 3: Serial performance of the SpMV multiplication over 2100 sparse matrices on A64FX for each format w.r.t. the equivalent plain implementation. 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.

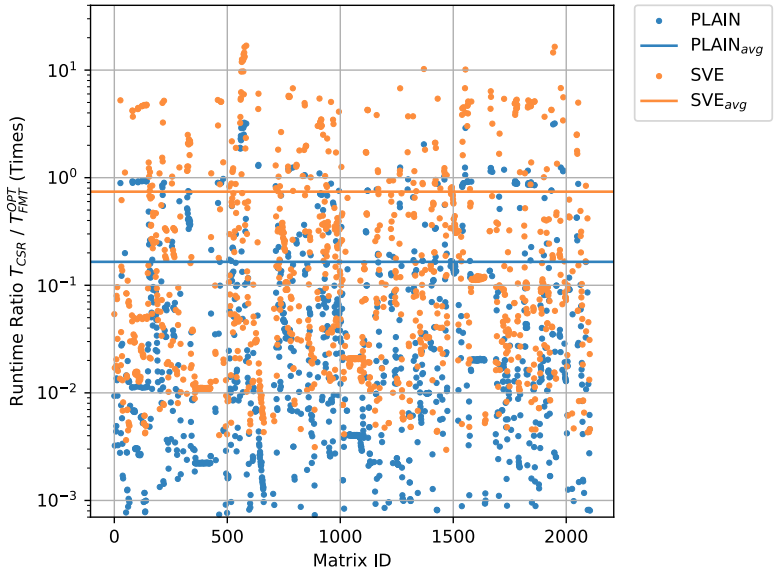
- No single implementation performs optimally across all matrices.
- Compiler not always capable for auto-vectorizing the sparse kernels.

epcc

Evaluation of SpMV on AArch64 CPUs



(a) COO



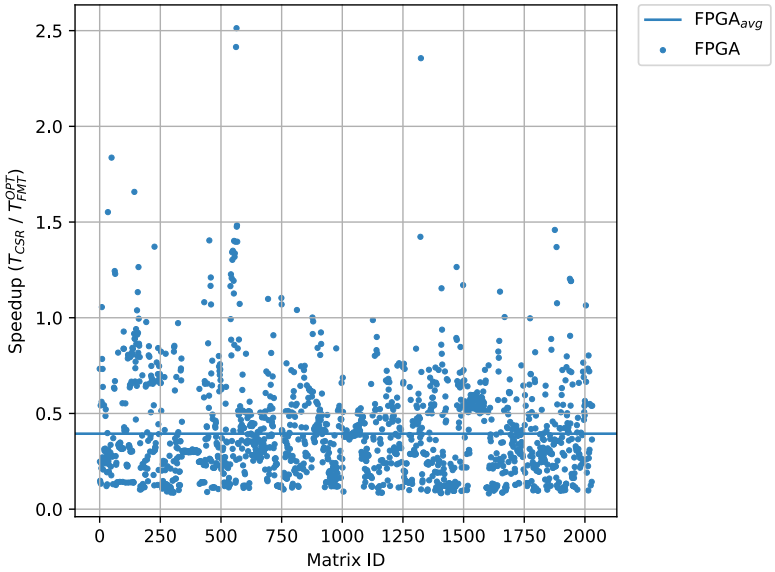
(b) DIA

Figure 4: Serial performance of the SpMV multiplication over 2100 sparse matrices on A64FX for each format w.r.t. the equivalent plain CSR implementation. 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.

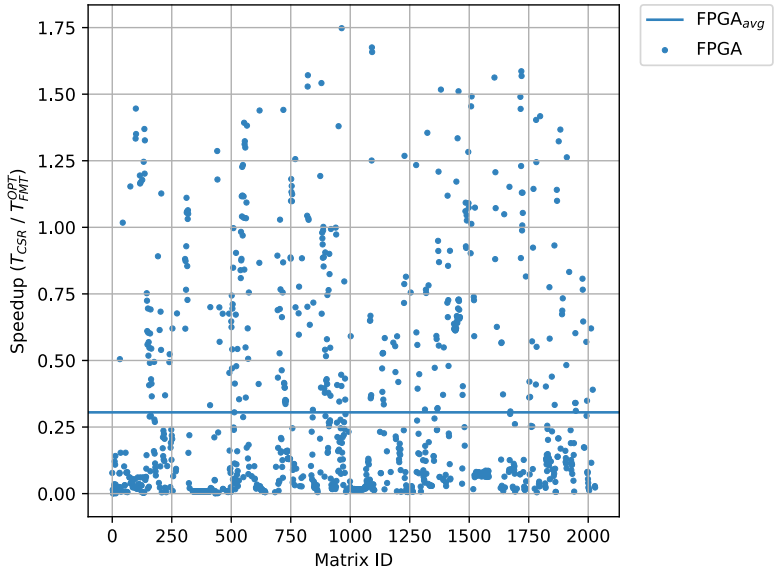
- For most matrices, plain CSR clearly outperforms other formats and implementations.
- SVE implementations for COO and DIA can offer noticeable speedup to runtime performance, sometimes by an order of magnitude higher.

epcc

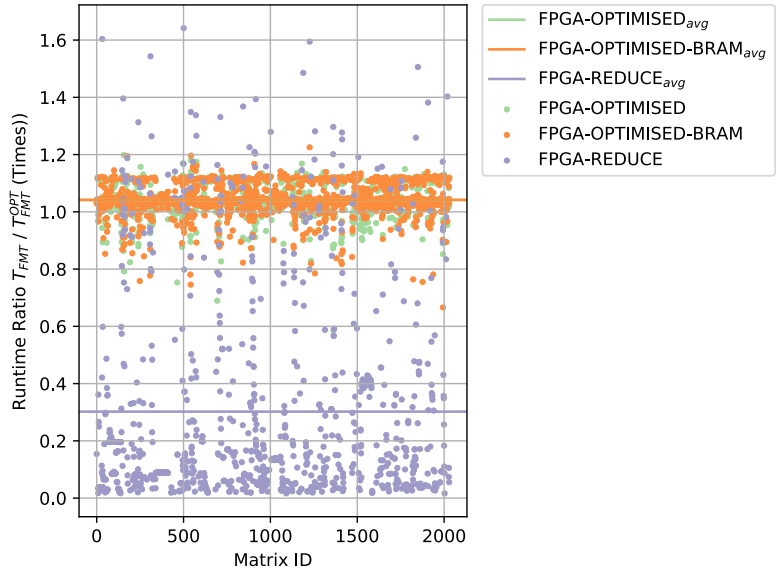
Evaluation of SpMV on FPGAs



(a) COO SpMV performance against CSR



(b) DIA SpMV performance against CSR



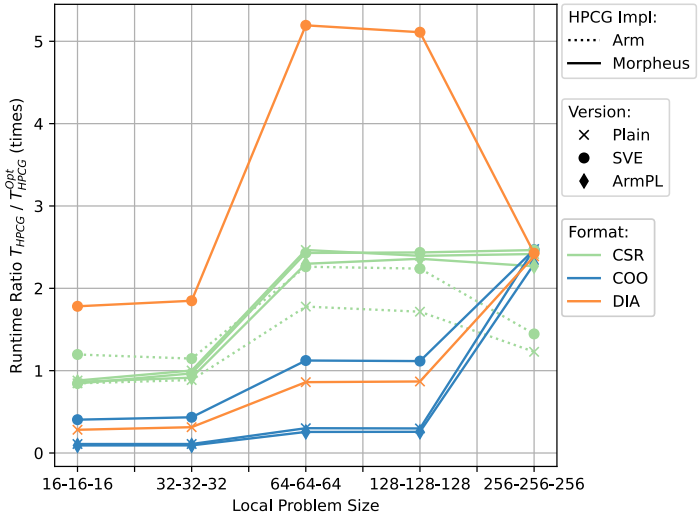
(c) Optimised COO SpMV versions against original COO

Figure 5: Serial performance of the SpMV multiplication over 2100 sparse matrices 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.

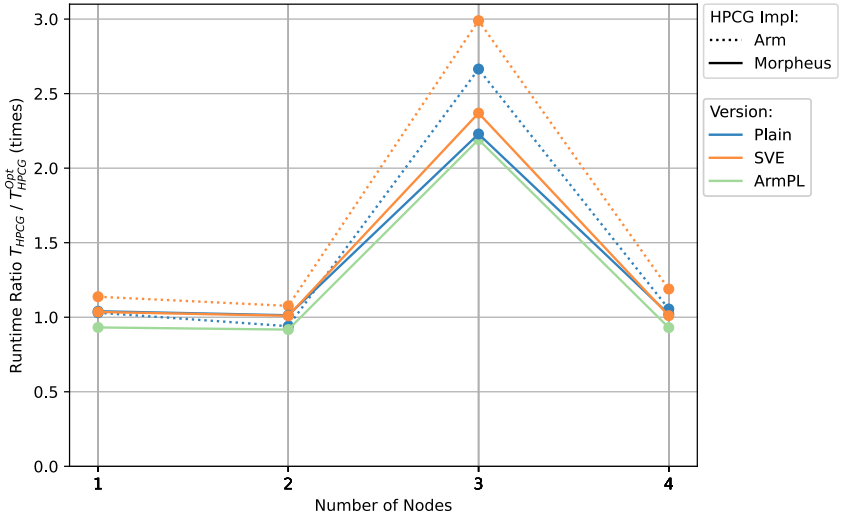
- 20% of sparse matrices on FPGAs perform optimally with COO and DIA format.
- Speedup less significant compared to CPU equivalents but still room for improvement.

epcc

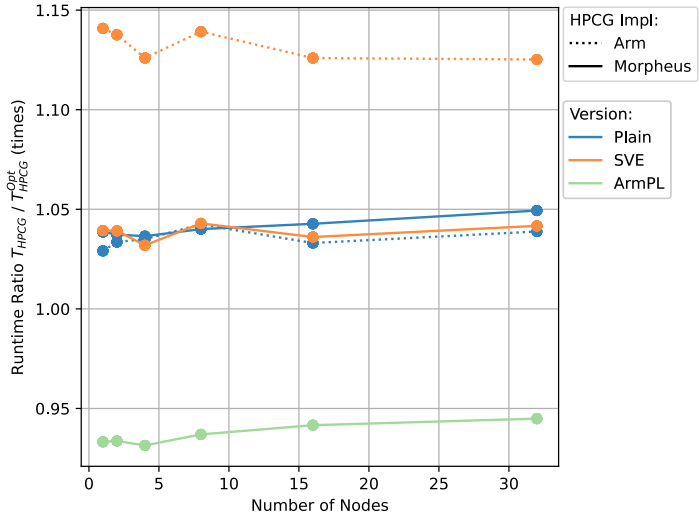
Comparison with ARM-HPCG



(a) Serial Performance



(b) Strong Scaling



(c) Weak Scaling

Figure 6: 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. A ratio above 1 indicates a speedup over the performance achieved when using the original HPCG.

- Serial Performance matches the observed performance from Figures 3 and 4 with Morpheus-enabled HPCG outperforming Arm-enabled HPCG (vendor implementation).
- Both strong and weak scaling performance of Morpheus-enabled HPCG closely tracks the performance of the optimal SVE version of the Arm-enabled HPCG.

epcc

Conclusions

- In this work, we explore the suitability of storage formats such as COO, CSR and DIA for emerging and non-traditional architectures, such as AArch64 CPUs and FPGAs.
- In addition, we detail hardware-specific optimisations to Aarch64 CPUs and evaluate the potential of each contribution to be integrated into *Morpheus.*
- Our findings for AArch64 CPUs show that no single format performs best, but also motivate the adoption of a dynamic selection mechanism for **selecting the optimal algorithm for each format**, since different optimizations can work best to different matrices.
- Furthermore, the adoption of SVE can significantly improve performance whenever the compiler struggles to auto-vectorize sparse kernels (e.g. COO and DIA SpMV).
- For the FPGAs, our naïve implementations showed similar optimal format distribution as their CPU equivalents, although on average performance was inferior.
  - Optimizing these implementations to use the dataflow style and also the available resources on the FPGA efficiently is an avenue of further work.
- New optimizations can be easily integrated and co-exist in *Morpheus.*
- It is possible to extend *Morpheus* to include new backends such as FPGAs, without any changes to the existing interface.