MB2020 D3.5 - PORTING OF (MINI) APPLICATIONS

VERSION 1.0

Document Information

Contract Number	779877
Project Website	www.montblanc-project.eu
Nature	Document
Authors	Adrià Armejach (BSC), Bine Brank (JUELICH), Nam Ho (JUELICH), Miquel Moreto (BSC), Dirk Pleiter (JUELICH)
Reviewers	Guillaume Colin de Verdière (CEA), Timothy Hayes (Arm)
Keywords	Scalable vector extension, HPC applications, gem5 simulations

Notices: This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779877.

Change Log

Version	Description of Change
v0.1	Initial version of the deliverable
v0.2	Draft complete for internal review
v0.3	Incorporated changes from first internal review
v0.4	Incorporated changes from second internal review
v1.0	Finalized report

Contents

Executive Summary

This document reports the results of the activities planned in Mont-Blanc 2020 WP3 under task T3.4:

> **Task 3.4: Porting of (Mini)Applications:** This task will ensure that all the necessary tools are available and functional for porting purposes, and will measure the impact of the various design options on the applications. Once ported an invaluable infrastructure will then be available, so that the hypothesis or choices made in WP4 and/or WP5 can be validated globally. Furthermore, this task will make sure that it is possible to use the applications as inputs for task 7.1 of WP7 which will validate and/or debug the emulator.

This deliverable details the porting efforts for the MB2020 applications. During our porting efforts we have relied on emulation (ArmIE) and simulation (gem5) environments, and we present set of results obtained with the latter in this report. Finally, we conclude that it is easy to port and understand code that uses SVE's vector length agnostic paradigm, which enables to write simpler and shorter code.

1 Introduction

In order to evaluate the main IP blocks being developed in MB2020 a comprehensive evaluation environment is necessary. This environment is being deployed in WP7 as part of the final demonstrator, which is an emulation platform that will enable integration and evaluation of the RTL designs developed by different project partners. In order to evaluate the performance of the system a set of applications was selected in D3.1. These applications will help test the requirements defined in deliverables D3.2 and D3.3 when evaluated on the demonstrator.

In this deliverable we detail the porting efforts undertaken for selected applications. These efforts include: (i) to generate appropriate binaries for the Arm Instruction Set Architecture (ISA) by using available tools like the Arm Compiler for HPC; (ii) to leverage the Scalable Vector Extension (SVE) to achieve competitive performance; and (iii) to ensure correct execution of ported applications by using available emulation and simulation tools such as ArmIE and gem5.

Most of the effort has been devoted to port the applications to exploit SVE capabilities. This has been achieved in different ways depending on the application, from lower to higher level of effort: (i) auto-vectorisation, (ii) using Arm Performance Libraries, (iii) using Arm C Language Extensions (intrinsics), or (iv) hand-tuned assembly code. Applications based on simple loops that have regular or contiguous memory access patterns can rely on compiler auto-vectorisation, while low level kernels that require high-performance are likely to benefit from hand-tuned assembly implementations. A significant effort has been made to achieve good SVE performance, however, performance fine-tuning has not been a main objective due to the lack of a target architecture to optimize for.

Finally, we also provide the details for the gem5-based simulation environment that allows us to validate the correct execution of the ported applications. This environment has also been used to evaluate application performance under different vector lengths and memory hierarchies, with similar studies than those employed for the definition of requirements in deliverable D3.2.

2 Porting of Applications

This section describes the effort to port selected MB2020 applications to the Arm ISA and SVE.

An initial effort was necessary to have a porting environment with the needed set of tools and compilers to generate SVE-enabled Arm binaries. In particular, we have mostly used the Arm HPC Compiler suite, which includes compiler front-ends for `fortran`, `c`, and `c++`. In addition, the suite also contains the Arm Performance Libraries, which initially lacked SVE support, but such support has been added in recent versions of the libraries.

Most of the selected applications lacked the build system support to target Arm architectures. Therefore, an effort to adapt the application's building infrastructures to use the compilers and libraries mentioned above was necessary.

Finally, most of the effort has been devoted to port the applications to efficiently use SVE. Applications that present simple and regular memory accesses can rely on compiler auto-vectorisation, while low level kernels that have to achieve high-performance require hand-tunned assembly implementations. Therefore, for each application we have employed a particular porting strategy, depending on its characteristics and the target performance level. We have used four strategies:

- Auto-vectorisation: For certain applications the compiler is able to auto-generate high quality vectorised code. This is particularly true for applications that present simple loop bodies with regular memory access patterns. For these applications we have visually inspected generated code to guarantee that all relevant loops are vectorised, and that the generated code is inline with what is expected.
- Arm Performance Libraries (ArmPL): We have used an SVE-enabled version of the ArmPL to link applications that were already prepared to rely on external optimised libraries.
- Arm C language extensions (intrinsics): Certain applications require more effort to obtain the desired performance by refactoring code and coding specific key routines using intrinsics. This allows to take full advantage of the capabilities of the target ISA.
- Hand-tuned assembly: Low level kernels that are widely used are highly optimised using assembly code.

Figure 1 shows the main vectorisation method employed for each of the selected MB2020 applications (see Deliverable D3.1 for more information). As can be seen in the figure, we have employed the four methods described above depending on the characteristics of each application.

Method	Arbor	HACCKernels	HPCG	miniKKR	Grid	MiniAMR	RAJAPerf	SWFFT	XSBench	BLIS (gemm lib.)	
Intrinsics	✓			✓	✓						3
Assembly						✓				✓	2
Auto-vectorization		✓					✓		✓		3
ARM Perf. Libs.			✓					✓			2

Figure 1: Main vectorisation method employed for selected MB2020 applications.

The following sections categorize the applications by porting method and describe in detail the porting efforts.

2.1 Auto-Vectorisation

In this section we detail the work done for three of the selected applications in order to enable the compiler to vectorise code efficiently. This includes the use of hints via pragmas and, in some cases, code transformations.

2.1.1 HACCKernels

The HACC [HPF+14] framework uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. It can simulate gravity forces produced between particles at cosmological scale. We use the HACCKernels mini-application that focuses on the N-body force calculations.

This benchmark presents a rather simple loop that should be auto-vectorised by the compiler. However, the first time we compiled the program we found that, while the Arm HPC Compiler was able to vectorise the code, GCC was unable to do it. To enable both compilers to vectorise the benchmark we modified the source code of the benchmark, see Listing 1. We found that the *continue* clause was the problem and removed it using boolean variables. Arm HPC Compiler made this transformation automatically by using a bit mask approach. The employed modifications are shown in Listing 2. With the modified code both compilers were able to vectorise the benchmark and the generated code was the same.

```
#pragma omp simd \
    reduction(+:lax,lay,laz)
for (int i = 0; i < n; ++i) {
  float dx = x[i] - x0;
  float dy = y[i] - y0;
  float dz = z[i] - z0;
  float r2 = dx * dx +
             dy * dy +
             dz * dz;

  if (r2 >= MaxSepSqrd || r2 == 0.0f)
    continue;


  float r2s = r2 + SofteningLenSqrd;
  float f=PolyCoefficients[PolyOrder];
  for (int p=1; p<=PolyOrder; ++p)
    f = PolyCoefficients[PolyOrder-p]
      + r2*f;

  f = (1.0 / (r2s * sqrt(r2s)) - f) *
      mass[i];

  lax += f * dx;
  lay += f * dy;
  laz += f * dz;
}
```

Listing 1: Original HACCKernels code

```
#pragma omp simd \
    reduction(+:lax,lay,laz)
for (int i = 0; i < n; ++i) {
  float dx = x[i] - x0;
  float dy = y[i] - y0;
  float dz = z[i] - z0;
  float r2 = dx * dx +
             dy * dy +
             dz * dz;

  bool bigger, zero;
  bigger = r2 < MaxSepSqrd;
  zero = r2 != 0.0f;

  float r2s = r2 + SofteningLenSqrd;
  float f=PolyCoefficients[PolyOrder];
  for (int p = 1; p <= PolyOrder; ++p)
    f = PolyCoefficients[PolyOrder-p]
      + r2*f;

  f = (1.0 / (r2s * sqrt(r2s)) - f) *
      mass[i];

  lax += f * dx * zero * bigger;
  lay += f * dy * zero * bigger;
  laz += f * dz * zero * bigger;
}
```

Listing 2: Modified HACCKernels code

Once we verified that both compilers were able to vectorise the loop correctly by inspecting the generated code, we used ArmIE to verify that the program was behaving as expected by comparing its output to native scalar executions.

2.1.2 RAJA Performance Suite

The RAJA performance suite (RAJAPerf) [raj] consists of 29 kernels that range from simple synthetic loops to representative loops extracted from real HPC applications. By using the Arm HPC Compiler and ArmIE, we were able to confirm that almost all the loops auto-vectorise with coverages above 93%. We found that 4 out of the 29 kernels failed to auto-vectorise, due to conditional branches and the use of the *continue* keyword. We provided the necessary feedback to Arm so that the compiler team could look into it.

Table 1 lists the 10 benchmarks from the RAJAPerf suite that we have looked into to ensure generated code by the compilers follows the expected behaviour. These selected benchmarks present good auto-vectorisation and provide different characteristics in terms of flops per byte read.

Benchmark	**Description**
MULADDSUB	Simple synthetic loop for quick testing
HYDRO_1D	Main computational loop of the HYDRO application
EOS	Calculates the equation of state
INT_PREDICTOR	Integral predictor
ENERGY	Kernel to calculate energy states, extracted from an LLNL application
PRESSURE	Kernel to calculate pressure in a system, extracted from an LLNL application
FIR	Finite impulse filter, extracted from an LLNL application
LTIMES	Kernel extracted from an LLNL application
Stream-DOT	Stream benchmark, dot-product version
VOL3D	Kernel to calculate a 3D volume, extracted from an LLNL application

Table 1: Selected benchmarks from the RAJAPerf suite for porting purposes.

For all these benchmarks we have added the *simd* clause to the existing OpenMP *parallel for* clause in order to force the compiler to vectorise the loops automatically. For some of the kernels we have also added the *schedule(dynamic,CHUNK)* clause in order to partition the workload in blocks of *CHUNK* iterations, which helped balance the workload across all cores when performing simulations.

For all these benchmarks the compiler vectorised the codes as expected and no further modifications were required.

Note to code developers: favor the omp SIMD clause as often as possible to guide both the compiler and future readers of your code.

2.1.3 XSBench

The XSBench proxy app [TSIS14] models the most computationally intensive part of a typical Monte-Carlo neutron transport algorithm, i.e., the calculation of macroscopic neutron cross sections, a kernel which accounts for around 85% of the total runtime of OpenMC. The essential computational conditions and tasks of fully featured neutron transport codes are retained in the mini-application.

This benchmark stresses the memory hierarchy both in terms of capacity and bandwidth, and is sensitive to memory latency. Thereby it is suitable to explore memory hierarchy as well as network-on-chip requirements. However, the benchmark does not vectorise well as it has an irregular memory access pattern with a randomized distribution. Therefore the porting efforts focused on enabling the correct compilation of the code under Armv8 platforms.

2.2 Using SVE-enabled Arm Performance Libraries

This section details the use of Arm Performance Libraries in two of the selected applications.

2.2.1 HPCG

The High Performance Conjugate Gradients (HPCG) Benchmark [DHL15] was developed to create a new metric for ranking HPC systems. HPCG is intended as a complement to High-Performance Linpack (HPL) and is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, as well as, to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications. HPCG is a complete, stand-alone code that measures the performance of basic operations such as: sparse matrix-vector multiplication, vector updates, global dot products, and sparse triangular solvers.

Prior work developed during the Mont-Blanc 3 (MB3) project [hpc, RMC+18] observed that most of the time the OpenMP threads are idle in this benchmark. The analysis measured that 86% of the whole execution time the code runs on a single thread. This is because the symmetric Gauss-Seidel, the most time consuming kernel of the benchmark, is a serial algorithm, therefore no OpenMP parallelization has been implemented in the reference version of HPCG. To overcome this limitation we introduced two coloring techniques for the symmetric Gauss-Seidel pre-conditioner, which allowed us to increase the performance of the OpenMP version of HPCG by 9.5×, matching the performance of the MPI-only version.

In MB2020 we have been using this modified version of HPCG. In addition to these modifications, HPCG is also a good candidate to use the proprietary Arm Performance Libraries (ArmPL). Most of the benchmark kernels can be mapped directly to BLAS or LAPACK functions, for example the *dot-product* and the *axpy* operations. Therefore, we decided to port this application to SVE by leveraging an SVE-enabled version of the ArmPL. Initially with a pre-release facilitated by Arm, and later by using the latest release of the Arm HPC Compiler that already comes bundled with an ArmPL version that features SVE implementations for most kernels.

The main computational functions that perform ArmPL calls are *ComputeDotProduct*, *ComputeWAXPBY*, and *ComputeSPMV*. We also use the ArmPL version that has been optimized with further OpenMP calls by invoking them with `-armpl=sve,parallel`.

2.2.2 SWFFT

SWFFT is another mini-application extracted from the well-known HACC [HPF+14] framework. It focuses on the 3D distributed memory discrete Fast Fourier Transform (FFT), which depends on external FFT libraries and is typically compute limited.

Because there is a dependence with an external FFT library this is a great fit to employ the ArmPL and leverage the available SVE-enabled implementation that follows the FFTW API, for which this application is already prepared. FFTW [fft] is a popular FFT library that has defined an standard API that other libraries follow to ensure code portability.

Since this application is parallelised employing just MPI, we have used the appropriate version of the libraries, and linked using the flag `-armpl=sve`.

2.3 Using Arm C Language Extensions for SVE

In this section, we present the porting of three applications using Arm C Language Extensions for SVE, commonly referred to as intrinsic functions.

2.3.1 Arbor

Arbor [ACK+19] is a high-performance library for network simulations of multi-compartment neuron models. In this model, the only interaction between neuron cells is mediated by voltage spikes. Each cell is modeled as a one dimensional electrical system (governed by the cable equation), which branches to other cells.

To exploit advantages of different architectures, Arbor uses specific optimizations for a CPU implementation. A dedicated backend vectorisation library is implemented that separates the interface from architecture-specific intrinsic functions. The library additionally supports vectorised implementations of transcendental functions as well as gather/scatter memory operations. It is written in C++14 and features support for AVX, AVX2, AVX512, SSE and NEON. Different architectures are implemented as templated structs within the *simd* namespace.

Due to constraints of SVE intrinsic types[1], it is not possible to implement a vector-length-agnostic SVE support in the current library framework, without changing the default API of SIMD vectorisation library. At the time of writing this deliverable, developers are aware of this problem and are working to fix it[2]. Despite this limitation, support for SVE was added by fixing the SVE vector length at compile time. Two different approaches were used:

- GCC 10 has support for fixed size SVE types, which can be used as struct members. This implementation completely avoids the problem with sizeless types. Thus, this implementation is the same as for other architectures.
- Another option is to leverage upon generic implementation and use *std::array* as a field to store data. Intrinsic functions are then only used in arithmetic operations between arrays. This option is suboptimal because there is additional data copying before SVE operations are used.

Listing 3 shows an example of GCC 10 implementation for double precision floating point number and its multiply-add operation. (Some of the code is abbreviated for clarity.)

```
// Checking if run-time SVE length is equal to the length defined at compile-time
#if __ARM_FEATURE_SVE_BITS==SVE_SIZE
// Defining fixed-size SVE vector and predicate
typedef svfloat64_t vec_double __attribute__((arm_sve_vector_bits(SVE_SIZE)));
typedef svbool_t vec_pred __attribute__((arm_sve_vector_bits(SVE_SIZE)));
#endif

template <>
struct simd_traits<sve_double> {
    // Using width equal to pre-defined SVE length
    static constexpr unsigned width = SVE_LENGTH;
    using scalar_type = double;
    using vector_type = vec_double;
    // sve_mask is a struct which uses a fixed-size predicate to create a mask
    using mask_impl = sve_mask;
};
// multiply-add implementation using types defined above
    static vec_double fma(const vec_double& a, const vec_double& b, const
    vec_double& c) {
        return svmad_z(svptrue_b64(), a, b, c);
    }
```

Listing 3: Arbor SVE implementation

[1] Intrinsic datatype corresponding to SVE vectors and predicates are defined as *sizeless types*. Among other restrictions, sizeless type cannot be used for members of unions, structures and classes. https://static.docs.arm.com/100987/0000/acle_sve_100987_0000_00_en.pdf

[2] https://github.com/arbor-sim/arbor/issues/1021

ArmIE together with Arbor's comprehensive unit testing suite was used to verify correctnes of implementation.

2.3.2 Grid

Grid is a data parallel quantum chromodynamics library [BCYP16]. One of its main SIMD strategies is the abstraction of architecture dependent vector instructions. Similarly to Arbor, such interface enables the majority of code to be written in a platform independent way.

SVE optimization [MGP+18] does not follow a vector-length agnostic programming model. Instead, the implementation is bound to the vector length of the target hardware. To fix the SVE implementation to the desired SIMD size, compile-time constant SVE_VECTOR_LENGTH is introduced. In addition, a templated C++ structure *vec*<*T*> is added. It contains an ordinary array v as member data and its operations are specialized for each supported typename T (64-bit, 32-bit and 16-bit floating point numbers, and 32-bit integers). ArmIE emulator and about 40 representative tests were used to verify the implementation.

2.3.3 MiniKKR

MiniKKR is a mini-application that represents the core operation of the Density Functional Theory (DFT) application KKRnano [TZB+12]. KKRnano's implementation of DFT truncates the long interaction between atomic cells and thus enables linear scaling. The main computational task of MiniKKR is a matrix inversion. Due to short-range forces between atoms, matrix A is block-sparse. This suggests iterative inversion over direct solution and we therefore deal with an iterative linear system $A \times x = b$. To reduce memory consumption, this problem is solved for all columns in a block and for different source blocks simultaneously. In other words, vectors x and b become matrices X and B and we are solving $A \times X = B$ (where B is a block-diagonal matrix). Typical block size is 16×16 or 32×32, depending on which atomic orbitals are considered. The system is solved with Transpose Free Quasi Minimal Residual method (TFQMR).

The main part of TFQMR algorithm is a block-sparse matrix-matrix multiplication with double precision complex numbers. This involves many calls to BLAS level-3 *zgemm* routine with small matrix dimensions (16 or 32). This specific kernel was ported to SVE using intrinsic functions. Multiplication of complex numbers is implemented with *svld2* instructions which load an array of two-element structures into two vectors. Real values are gathered in one simd register and imaginary values in the other. We construct complex multiplication with 3 *fmla* and one *fmls* instructions as shown in Listing 4. Alternatively, one could also link MiniKKR to an SVE enabled version of BLAS.

```
svbool_t prj = svwhilelt_b64(0UL,K);
for(size_t j = 0; svptest_any(svptrue_b64(),prj);)
{
    svfloat64x2_t b     = svld2(prj,(const double*)(B+i*ldb+j));
    svfloat64x2_t ab_ji;

    // complex multiplication
    ab_ji.v0 = svmul_lane(b.v0, alphavec, 0);
    ab_ji.v0 = svmls_lane(ab_ji.v0, b.v1, alphavec,1);
    ab_ji.v1 = svmul_lane(b.v0, alphavec, 1);
    ab_ji.v1 = svmla_lane(ab_ji.v1, b.v1, alphavec,0);

    svst2(prj, (double*)(abcache_ptr+j), ab_ji);
```

Listing 4: MiniKKR SVE implementation

In the end, the implementation of MiniKKR was verified with ArmIE.

2.4 Using Hand-Tuned SVE Assembly

Finally, we present two applications that were ported with the use of inline assembly. Although this strategy is most effortful, it is necessary for applications that require absolute control over performance critical parts of code.

2.4.1 BLIS

BLIS [VZvdG15] is a portable software framework for high-performance BLAS dense linear algebra routines. Its main feature is that almost all level-2 and level-3 BLAS operations can be expressed in terms of a few simple kernels, called *microkernels*. These microkernels need to be highly optimized for a target architecture and are therefore typically implemented in assembly or with intrinsic functions.

Here, we focus on level-3 BLAS routines, more specifically on general matrix-matrix multiplication (gemm), which computes $C = \alpha AB + \beta C$. (α and β are scalars and C, A, B are matrices of size $m \times n$, $m \times k$ and $k \times n$ respectively.) BLIS implements *gemm* with six nested for-loops[3]. Three outer loops partition matrices C, A and B into smaller blocks C_c, A_c and B_c of sizes $m_c \times n_c$, $m_c \times k_c$ and $k_c \times n_c$. Simultaneously, these matrices are packed in specific order, which enables sequential access in the microkernel. Loop 4 and 5 further partition block-matrices into micro-tiles and micro-panels. The innermost loop (inside the microkernel) computes the product of a micro-panel of A_c and micro-panel of B_c as a sequence of k_c rank-1 updates, accumulating the result to a micro-tile of C_c. Micro-tile has a size of $m_r \times n_r$.

Sizes m_c, k_c, n_c, m_r, n_r can be derived from architecural parameters of the target machine [LISQO16]. Here we explain only the methodology for *dgemm* microkernel. By implementing and optimizing the *dgemm* microkernel, all level-3 operations on double precision numbers are fully optimized except those in the TRSM family (in which one matrix operand is triangular).

Two different approaches were conducted while porting to SVE.

- Vector-length-agnostic microkernel: In this approach, the microkernel does not have a fixed size. While n_r is constant, m_r scales with SVE vector size. This is in line with theoretical analysis, which states that size of micro-tile depends linearly on the underlying SIMD vector size. Prefetching is implemented through vector prefetch instructions. m_c, k_c and n_c are all dependant on m_r and n_r and are therefore computed at run-time.

- Fixed-size microkernel: Specific microkernels for SVE lengths of 256, 512 and 1024 bits were written. This follows the same approach as for traditional fixed-vector-length architectures. It only works for the respective SVE size.

Beside *dgemm* microkernel, many other kernels (*zgemm*, *sgemm*, *daxpy*, *zaxpy*) have also been ported to SVE. It is still a work in progress to have the entire SVE enabled BLIS library.

ArmIE was used to verify correctnes of the different implementations. Furthermore, gem5 with Konata visualizing tool has been used to investigate the pipeline behaviour of the underlying microkernel.

[3]The microkernel corresponds to the innermost loop.

```
mov     x1, #1                  // k:=1
loop:
cmp     x1, z_size
b.eq    end
ldr     d4, [A, x1, lsl #3] //A[i][j][k]
add     x2, x1, offs_north
ldr     d8, [A, x2, lsl #3] //A[i][j][k+1]
fadd    d4, d4, d8
sub     x2, x1, offs_north
ldr     d8, [A, x2, lsl #3] //A[i][j][k-1]
fadd    d4, d4, d8
add     x2, x1, offs_front
ldr     d8, [A, x2, lsl #3] //A[i][j+1][k]
fadd    d4, d4, d8
sub     x2, x1, offs_front
ldr     d8, [A, x2, lsl #3] //A[i][j-1][k]
fadd    d4, d4, d8
add     x2, x1, offs_east
ldr     d8, [A, x2, lsl #3] //A[i+1][j][k]
fadd    d4, d4, d8
sub     x2, x1, offs_east
ldr     d8, [A, x2, lsl #3] //A[i-1][j][k]
fadd    d4, d4, d8
fmul    d4, d4, constant
str     d4, [B, x1, lsl #3] //B[i][j][k]
add     x1, x1, #1
b       loop
end:
```

(a) Scalar version

```
mov     x1, #1
loop:
whilelt p0.d, x1, z_size
b.eq    end
ld1d    z4.d, p0/z, [A, x1, lsl #3]
add     x2, x1, offs_north
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
sub     x2, x1, offs_north
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
add     x2, x1, offs_front
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
sub     x2, x1, offs_front
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
add     x2, x1, offs_east
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
sub     x2, x1, offs_east
ld1d    z8.d, p0/z, [A, x2, lsl #3]
fadd    z4.d, z4.d, z8.d
fmul    z4.d, p0/m, z4.d, constant
st1d    z4.d, p0, [B, x1, lsl #3]
incp    x1, p0.d
b       loop
end:
```

(b) SVE version (VLA)

Figure 2: Code fragment of the 7-point stencil computation loop - only showing the innermost loop.

2.4.2 MiniAMR

MiniAMR [HDC+09] applies a stencil calculation on a unit cube computational domain, which is divided into blocks. The blocks all have the same number of cells in each direction and communicate ghost values with neighbouring blocks. With adaptive mesh refinement, the blocks can represent different levels of refinement in the larger mesh. MiniAMR allows the use of 7-point and 27-point stencils.

During the MB3 project we ported a serial version of the MiniAMR stencil kernel code to SVE assembly. Figure 2 shows the *scalar* and SVE baseline code of the innermost loop of a 7-point stencil. SVE uses the *whilelt* instruction to iterate over the for-loop. This instruction allows operating over the loop independently of the vector length (VL) and the number of iterations. *whilelt* constructs the predicate register, *p0*, by evaluating the condition *lt* (less than) on the content of registers *x1* and *z_size*. *p0* can be seen as a mask that tells the architecture if a specific vector lane is enabled ('1'), or disabled ('0'). Instructions *ld1d*, *st1d*, *fadd* and *fmul* are the vector equivalent of the scalar instructions *ldr*, *str*, *fadd* and *fmul*, respectively. These new instructions operate on vector registers (*z4.d* and *z8.d*), which contain a set of double precision floating point elements. The *incp* instruction increments the content of register *x1* by the number of active elements in *p0*. SVE executes the code inside the loop z_size/VL times, with an additional predicated iteration if $(z\_size \ mod \ VL) \neq 0$.

Using this SVE implementation as a starting point we implemented a number of different

code optimizations such as loop unrolling, loop fusion, data reuse, and load trading by leveraging SVE's vector length agnostic programming paradigm. We showed that for the serial version used in the MB3 project, loop unrolling combined with loop fusion performed the best on the 7-point stencil, while loop unrolling alone was best for the 27-point stencil [ACC+18].

During the MB2020 project we ported all our SVE-enabled code to a version of MiniAMR that supports MPI. This allowed us to evaluate this workload on a more realistic environment, that is, taking advantage of multicore architectures. By using as many MPI processes as cores are available, the workload changes significantly as memory contention now plays a major role. In fact, the obtained results differ significantly and the data reuse optimization now outperforms all the others in the 27-stencil implementation. Intuitively, data reuse is now of paramount importance as the memory subsystem is under more stress compared to the serial version. The amount of reused data and partial computations this optimization exposes is significant, i.e., two-thirds per iteration [ACC+20].

In Section 4 we evaluate the best performing set of optimizations for the 7- and 27-point stencils on the MPI version of MiniAMR. That is, for the 7-point stencil we do 3 levels of unrolling and loop fusion, and for the 27-point stencil we employ the data reuse optimization.

3 Methodology to Evaluate Ported Applications

This section describes the main tool we have employed to evaluate the ported applications, gem5. We have used gem5 to validate that application porting has been performed without introducing software bugs and to obtain initial performance results for the ported applications employing different parameters, such as: different memory technologies and varying SVE vector lengths.

The same gem5 infrastructure has also been used to generate application traces that will feed the WP7 emulation platform for the final MB2020 project evaluation.

3.1 Benchmarks and Inputs

Table 2 contains a list of all the evaluated benchmarks and their input parameters. We evaluate a subset of the selected MB2020 applications from deliverable D3.1. From MiniAMR, we evaluate a 7-point and a 27-point stencil input. From RAJAPerf suite, we evaluate a subset of the most interesting available benchmarks, namely: Stream-DOT, HYDRO-1D, EOS, VOL3D, and LTIMES. In Arbor, we focus on the ring benchmark, which connects cells in a simple ring topology. The MiniKKR benchmark uses an input generated for solving a poisson equation. Grid is evaluated for Wilson-sweep benchmark. Finally, from the BLIS library, we perform experiments with three forms of general matrix multiplication: single precision (sgemm), double precision (dgemm), and using complex numbers (zgemm).

Benchmark	**Input**
HPCG	32x32x32
HACCKernels	128
MiniAMR-7pt	–npx 2 –npy 2 –npz 2 –nx 64 –ny 64 –nz 64 –num_objects 1 –object 2 0 -1.10 -1.10 -1.10 0.030 0.030 0.030 1.5 1.5 1.5 0.0 0.0 0.0
MiniAMR-27pt	–stencil 27 –npx 2 –npy 2 –npz 2 –nx 64 –ny 64 –nz 64 –num_objects 1 –object 2 0 -1.10 -1.10 -1.10 0.030 0.030 0.030 1.5 1.5 1.5 0.0 0.0 0.0
XSBench	-s small -p 80000
Stream-DOT	–sizefact=0.002 –repfact=100
HYDRO-1D	–sizefact=0.001 –repfact=250
EOS	–sizefact=0.0005 –repfact=500
VOL3D	–sizefact=.005 –repfact=5
LTIMES	–sizefact=0.02 –repfact=30
Grid	–grid 4.4.4.4
Arbor	num_cells=30
MiniKKR	r_source=1 r_target=3
BLIS - dgemm	1024
BLIS - sgemm	1024
BLIS - zgemm	1024

Table 2: Benchmark inputs for the evaluated applications.

Processor size	8 cores
Cores	3-wide issue/retire, 92-entry instruction queue, 192-entry ROB, 48 LDQ + 48 STQ, 2 vector processing units (VPU), 2GHz
Private Caches	L1I: 64KB, 4-way, 2 cycle, 8MSHRs LID: 64KB, 4-way, 2 cycle, 24MSHRs L2: 256KB, 8-way, 9 cycle, 24MSHR, stride prefetcher
Last-level Cache	16MB, 16-way, 64B lines, 8 banks, 32MSHRs per bank Data bank access latency of 20 cycles.
NoC	Coherent crossbar, 128-bit wide, 2 cycles
Main Memory	4 DDR4-2400 channels, 2 ranks/channel, 16 banks/rank, 8KB row-buffer 128-entry write and 64-entry read buffers per channel 76.8GB/s peak bandwidth. Bank conflicts and queuing delays modeled
	HBM stack with 8 channels of 128 bits each. 128GB/s peak bandwidth. Bank conflicts and queuing delays modeled

Table 3: Parameters for full-system simulations.

3.2 Simulation Infrastructure

We detail the gem5 platform used to obtain traces for the final emulator evaluation in WP7, and to run simulations to check correctness and obtain performance estimations of ported applications.

We have configured our simulation infrastructure to look like the architecture envisioned in the MB2020 project. We use clusters of 4 cores modeled after the architectural parameters that resemble a Cortex-A75 core. Each core has private L1 and L2 caches, and there is a shared last-level L3 cache across all cores. The off-chip memory technologies can then be modeled on top of this multi-core setup and we plan to use both pin-based technologies like DRAM (DDR4) as well as High-Bandwidth Memory (HBM) stacks with wider and abundant memory channels that are based on silicon interposer technologies, bypassing the physical limitations of pin-based off-chip memory. Table 3 details the architectural parameters we use.

3.3 Experiments

We perform simulations for each benchmark described in Table 2 on the gem5 infrastructure described in Table 3. For each benchmark we evaluate, if possible, a scalar binary (without any vectorisation, `-fno-vectorize`), and the SVE binary for all available vector lengths (128 to 2048 bits).

In addition, we evaluate two different memory technologies. One based on DDR4 pin-based interfaces and one based HBM with through silicon interposer.

4 Evaluation of Ported Applications

Figures 3 and 4 show the roofline models for the target system when employing 4 DDR memory channels. We have split the benchmarks over two charts to ease readability.

A roofline model ties together floating point performance, operational intensity, and memory performance in a two-dimensional graph. The Y-axis is GFlops per second (performance). Theoretical ceilings can be derived using the hardware specifications; in our case, the simulated system can achieve 64GFlops/s for a 128-bit vector length: 2 vector processing units × 2 double floating-point operations per unit × 2GHz × 8 cores. The X-axis is operational intensity, i.e., operations per byte of DRAM traffic. Therefore, we measure traffic between the caches and main memory (including prefetched data), rather than between the processor and the caches. Thus, operational intensity suggests the DRAM bandwidth needed by a kernel on a particular computer. We can then plot memory performance by calculating the maximum floating-point performance that the memory system of that computer can support for a given operational intensity. This formula drives the two performance limits in the roofline model:

$$attainable\ GFlops/s = min(peak\ FP, peak\ memory\ bandwidth \times operational\ intensity) \quad (1)$$

The two lines intersect at the point of peak computational performance and peak memory bandwidth, i.e., the ridge point. This point denotes the minimum operational intensity required to achieve maximum performance. As can be seen in the figures, we plot 2 memory bandwidth ceilings, one for the theoretical peak pin bandwidth (black line) and another one for the measured bandwidth the system delivers using Stream-DOT (red line). In addition, we plot multiple peak performance ceilings depending on the vector length employed. From the figures, we can see that even with the peak bandwidth of 76.8GB/s (or 9.8GB/s per core), which is

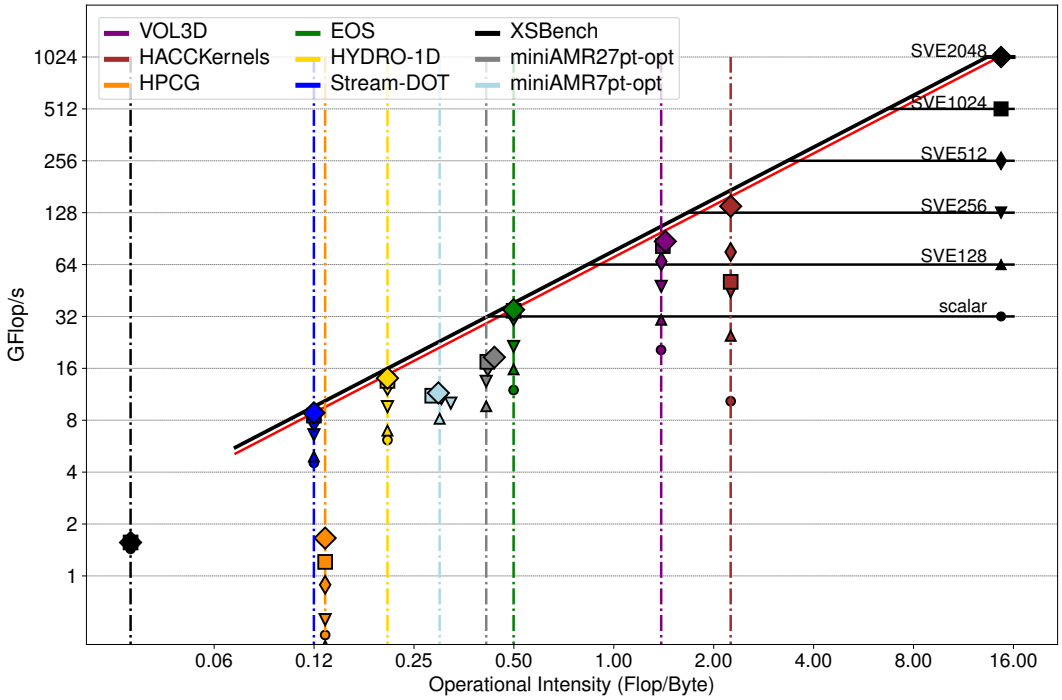

Figure 3: Roofline for selected benchmarks using 4 DDR memory channels. Its theoretical peak bandwidth is 76,8 GB/s.

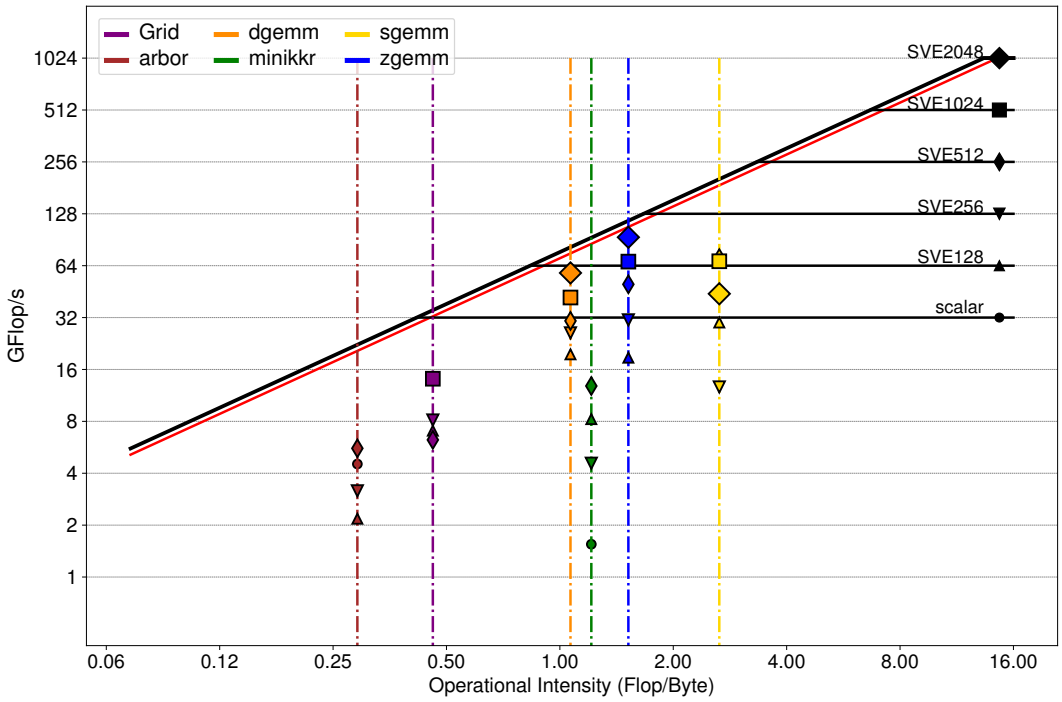

Figure 4: Roofline for selected benchmarks using 4 DDR memory channels. Its theoretical peak bandwidth is 76,8 GB/s.

typical from pin-based off-chip memory systems, an operational intensity of almost 1 Flop/byte is needed to be close to the peak performance a 128-bit SVE system can deliver. In addition, we plot multiple markers per benchmark, one for each vector length.

As can be seen in Figure 3, XSBench has a very low operational intensity and increasing the SVE vector length does not yield additional performance. This benchmark is meant to stress memory via random accesses that put pressure on the overall access latency. Therefore, it is not a good candidate for vectorisation and does little compute; it was selected as a good stress test for the network-on-chip. HPCG obtains poor performance as it is hard to vectorise and contains a serial section of code. Stream-DOT, HYDRO-1D, EOS, and the miniAMR benchmarks benefit from vectorisation; however, given their low operational intensity they quickly become memory bound even for narrow vector lengths. Vector lengths used in today's mainstream systems (e.g., 256 or 512-bits) necessitate memory bandwidths that are difficult to achieve with pin-based off-chip memory systems, and silicon interposer technologies are necessary to attain close to peak floating-point performance at the operational intensities found in typical HPC applications. Finally, VOL3D and HACCKernels present a better ratio of flop per byte, which enables larger performance gains with wider vectors.

Remaining applications are shown in Figure 4. Performance of BLIS kernels sgemm, dgemm and zgemm increases with a bigger vector size with a few exceptions. For example, we have seen an ambiguous behavior in sgemm using SVE256 in gem5, where the number of scalar instructions increases drastically, penalizing performance. The number of SVE instructions scales as expected, which means this artifact is outside the main computational loop. We attribute this behaviour to a gem5 bug or to a corner case being triggered with the OpenMP runtime or other system libraries. We were not able to replicate this behaviour using ArmIE, where the application behaves as expected. In addition, BLIS has a complex mechanism for partitioning matrices and this mechanism depends heavily on both architectural parameters as

well as matrix size. Kernels also use SVE vector prefetch instructions which are not yet available in gem5. Performance of Arbor is relatively poor due to a large portion of simulation time spent in Hines solver, which cannot be vectorised. In case of Grid and MiniKKR, we attribute low performance to a very small input size, where we were limited by long simulation times of gem5. Note that some results are missing for particular vector lengths. For Arbor and Grid, runs failed when using 1024 and 2048-bit vector size. This was reported back to developers and is being investigated. In case of Grid, the 2048-bit vector was too big for a lattice size of 4x4x4x4.

Figure 5 shows a similar roofline model but using a memory subsystem that features HBM technology, which has a peak memory bandwidth of 128 GB/s, i.e., 16 GB/s per core. As can be seen in the figure, memory bound benchmarks are starting to experience modest performance improvements with wider vector lengths. This is specially noticeable in HYDRO-1D and EOS. In order to maximize performance of current vector architectures, having abundant memory bandwidth is necessary. Figure 6 also shows improvement for BLIS (particulary sgemm and dgemm) kernels. For the other applications, we again suspect serial code sections and small input sizes to be the main reasons for poor scaling.

Next, Figure 7 offers a direct comparison with the results of two typical applications, EOS and dgemm, for DDR and HBM memory hierarchies. This highlights the need for an HBM memory subsystem, as performance with wider vector lengths improves substantially. For example, in EOS performance increases from 34.5 for SVE1024 and 35.01 GFlop/s for SVE2048 when using DDR to 46.17 and 49.62 GFlop/s when using HBM, respectively. These are improvements of 1.34× and 1.44×, respectively. For SVE512 improvements are of 1.13×. These results motivate the use of HBM technology and motivate the effort done in MB2020 to investigate and target a system that includes an HBM memory subsystem.

Finally, Figure 8 shows the instruction reduction normalized to an execution with SVE 128 bits. We observe that VOL3D, EOS, HYDRO-1D, and Stream-DOT scale very well in terms

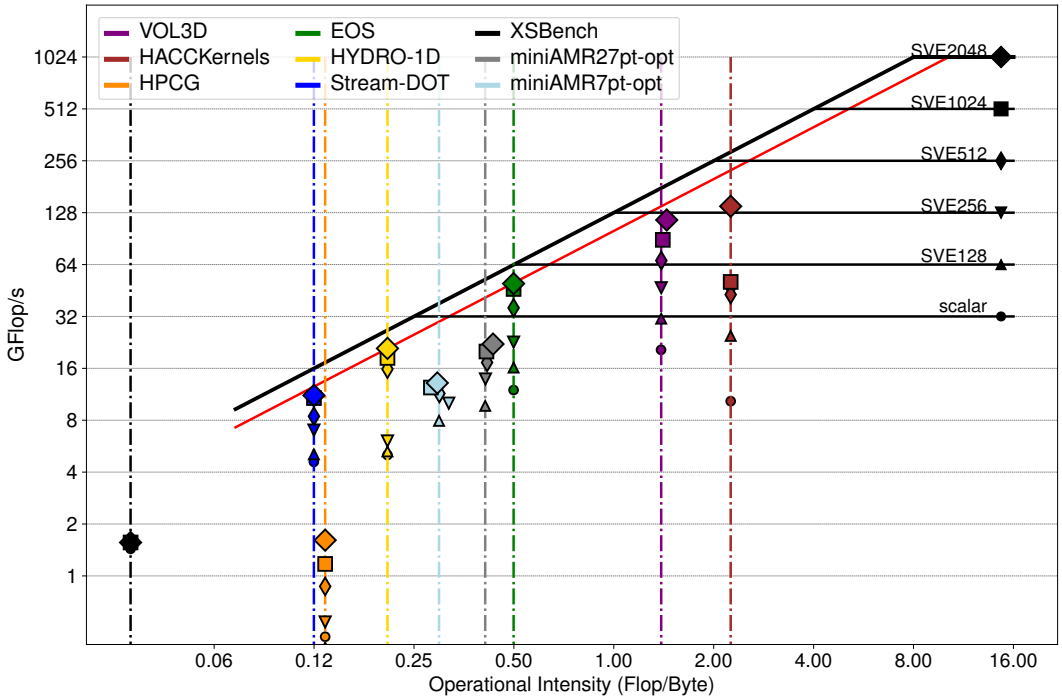

Figure 5: Roofline for selected benchmarks using 1 stack of HBM memory (8 channels of 128 bits each). Its theoretical peak bandwidth is 128 GB/s.

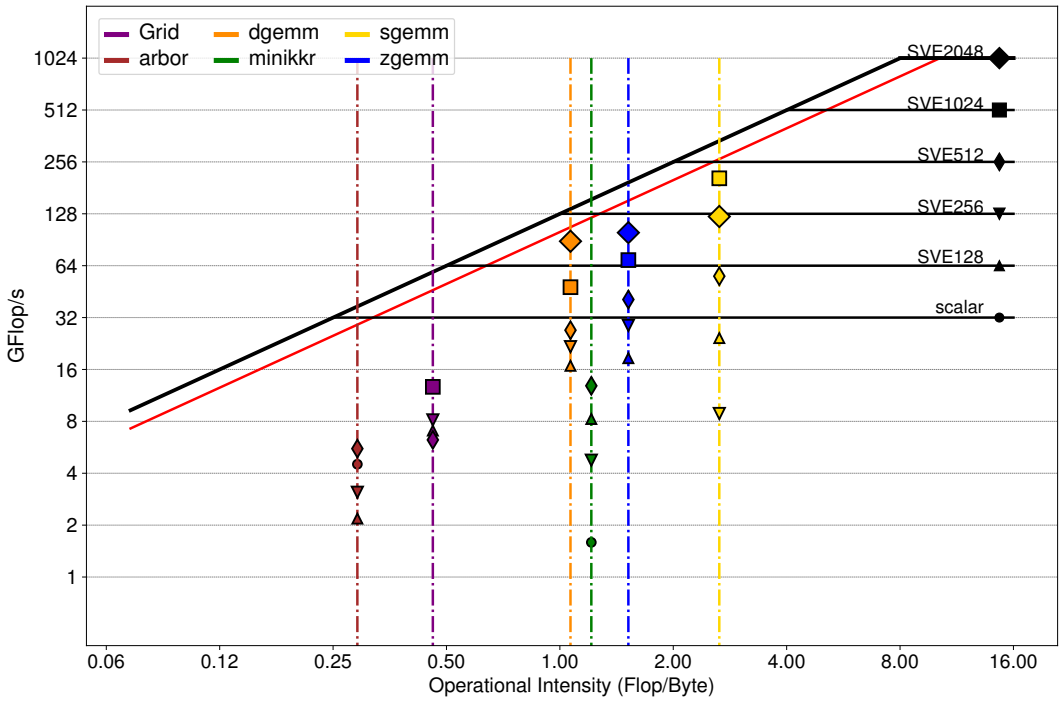

Figure 6: Roofline for selected benchmarks using 1 stack of HBM memory (8 channels of 128 bits each). Its theoretical peak bandwidth is 128 GB/s.

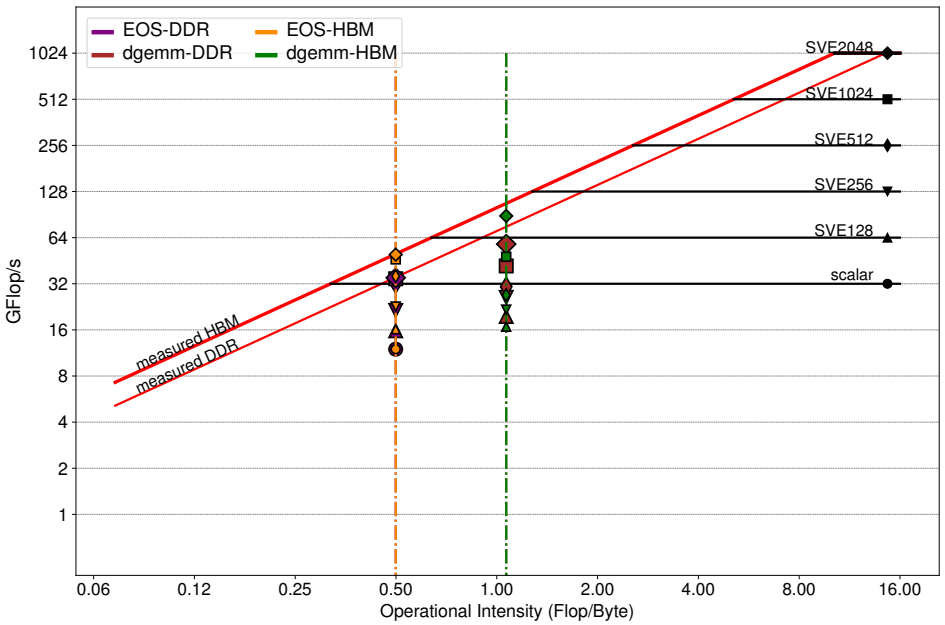

Figure 7: Roofline for two typical applications (EOS and dgemm) comparing DDR and HBM results.

of instruction reduction for all vector lengths; and also have a substantial reduction when compared to their scalar binaries. HPCG and XSBench, as mentioned before, have little potential for vectorisation, which leads to mild reductions compared to scalar. Finally, the miniAMR

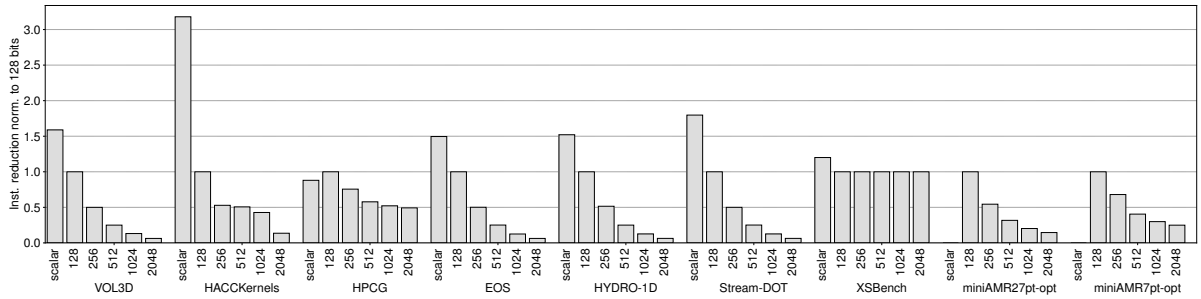

Figure 8: Committed instruction scaling normalized to SVE 128.

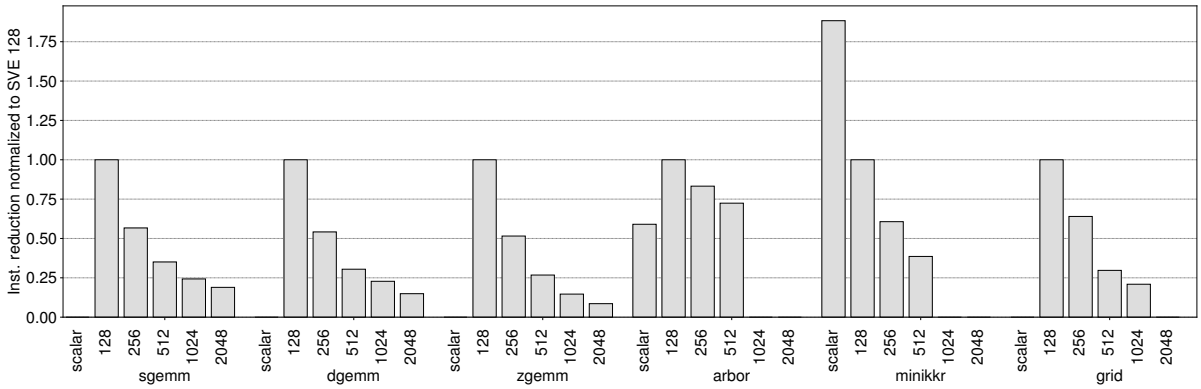

Figure 9: Committed instruction scaling normalized to SVE 128.

benchmarks also scale well, but lose a little bit of efficiency at the widest vector lengths, since the loop control instructions become dominant. Also note that for these benchmarks there is no scalar version available as these were written directly in SVE assembly.

Instruction reduction for remaining set of applications is shown in Figure 9. It can be seen that all applications scale well, when bigger vector sizes are used. For MiniKKR and BLIS kernels (sgemm, dgemm, and zgemm), both scalar and SVE part of instructions scales as a/L, where L is vector length. This is the expected behaviour. In case of Arbor, scaling is not so efficient due to already mentioned serial part of Hines solver. For Grid, we observe that scaling efficiency gets worse at bigger vector length, which can be attributed to a small lattice size.

5 Conclusions

Porting applications to SVE is a challenging task due to the lack of real hardware platforms to develop, debug, and test the applications. On top of this, the maturity of the software ecosystem toolchain, including compilers, libraries, and debuggers, presents additional challenges. For example, well optimized numerical and algebraic SVE-enabled libraries are still in its infancy and some of the functionality is not yet available. Moreover, SVE support in popular debugging tools like GDB is still lacking, as of the writing of this document.

During our porting efforts we have relied on emulation (ArmIE) and simulation (gem5) environments. While these environments have their own limitations, e.g., execution speed, they have proven to be invaluable to accomplish the level of porting presented in this deliverable.

As real-world implementations of SVE become available in commercial chips, and with the advancements we have seen in the software ecosystem tools; developing and porting SVE applications will become a much simpler task. In our opinion, SVE is ready to hit the market and is likely to see a quick adoption by the HPC community.

With respect to the SVE ISA, we have found that it is easy to reason about its vector length agnostic paradigm, which enables to write simpler and shorter code. More importantly, its C language extensions and datatypes are much more intuitive and comprehensive than other currently available solutions.

Acronyms and Abbreviations

- **ArmIE Arm I**nstruction **E**mulator
- **ArmPL Arm P**erformance **L**ibraries
- **DFT D**ensity **F**unctional **T**heory
- **DDR D**ouble **D**ata **R**ate memory
- **FIR F**inite **I**mpulse **F**ilter
- **GDB G**NU Project **D**e**B**ugger
- **HBM H**igh **B**andwidth **M**emory
- **HPC H**igh **P**erformance **C**omputing
- **IP I**ntellectual **P**roperty
- **ISA I**nstruction **S**et **A**rchitecture
- **LLNL L**awrence **L**ivermore **N**ational **L**aboratory
- **MSHR M**iss **S**tatus **H**olding **R**egisters
- **MPI M**essage **P**assing **I**nterface
- **OpenMP O**pen **M**ulti-**P**rocessing
- **RTL R**egister-**T**ransfer **L**evel
- **SVE S**calable **V**ector **E**xtension
- **TFQMR T**ranspose **F**ree **Q**asi **M**inimal **M**ethod
- **VLA V**ector **L**ength **A**gnostic

References

[ACC+18] Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. Stencil codes on a vector length agnostic architecture. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*, 2018.

[ACC+20] Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rubén Langarita, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. Using arm's scalable vector extension on stencil codes. *The Journal of Supercomputing*, 2020.

[ACK+19] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 274–282, feb 2019.

[BCYP16] Peter A. Boyle, Guido Cossu, Azusa Yamaguchi, and Antonin Portelli. Grid: A next generation data parallel c++ qcd library. *Proceedings of The 33rd International Symposium on Lattice Field Theory PoS(LATTICE 2015)*, Jul 2016.

[DHL15] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. HPCG benchmark: a new metric for ranking high performance computing systems. 2015.

[fft] The FFTW library. `https://www.fftw.org`.

[HDC+09] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[hpc] MB3 D6.9 Performance analysis of applications and benchmarking on the project test platforms. `cs://www.montblanc-project.eu/wp-content/uploads/2019/02/MB3_D6.9_Performance-analysis-of-applications-and-benchmarking-on-the-project-te v1.0.pdf`.

[HPF+14] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, Venkatram Vishwanath, Zarija Lukic, Saba Sehrish, and Wei-keng Liao. Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42, 10 2014.

[LISQO16] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016.

[MGP+18] Nils Meyer, Peter Georg, Dirk Pleiter, Stefan Solbrig, and Tilo Wettig. Sve-enabling lattice qcd codes. *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep 2018.

[raj] RAJAPerf. https://asc.llnl.gov/coral-2-benchmarks/downloads/RAJAPerfSuite_Summary.pdf.

[RMC+18] Daniel Ruiz, Filippo Mantovani, Marc Casas, Jesus Labarta, and Filippo Spiga. The hpcg benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform. Technical report, Universitat Politcnica de Catalunya (UPC), 2018.

[TSIS14] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XS-Bench – The development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[TZB+12] A. Thiess, R. Zeller, M. Bolten, P. H. Dederichs, and S. Blgel. Massively parallel density functional calculations for thousands of atoms: KKRnano. *Physical Review B*, 85(23), jun 2012.

[VZvdG15] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015.