Porting and tuning GROMACS on Arm-SVE

Gilles Gouaillardet
gilles@rist.or.jp

RIST – Research Organization for Information Science and Technology

- Japanese company with offices in Tokai, Tokyo and Kobe
- Involved in many aspects of High Performance Computer Initiative (HPCI)

  https://www.hpci-office.jp
  - Promotion
  - Project selection and resource allocation (everyone can apply!)
  - Single point of contact helpdesk: helpdesk@hpci-office.jp
    - First level support
    - Application installation and support
    - Advanced support (porting, tuning)
  - Co-host the Joint Seminars on Advanced use of Supercomputer Fugaku and Arm computer systems
    - https://www.hpci-office.jp/en/events/seminars
    - July 17th: Virtual Fugaku

GROMACS

- Molecular dynamics application

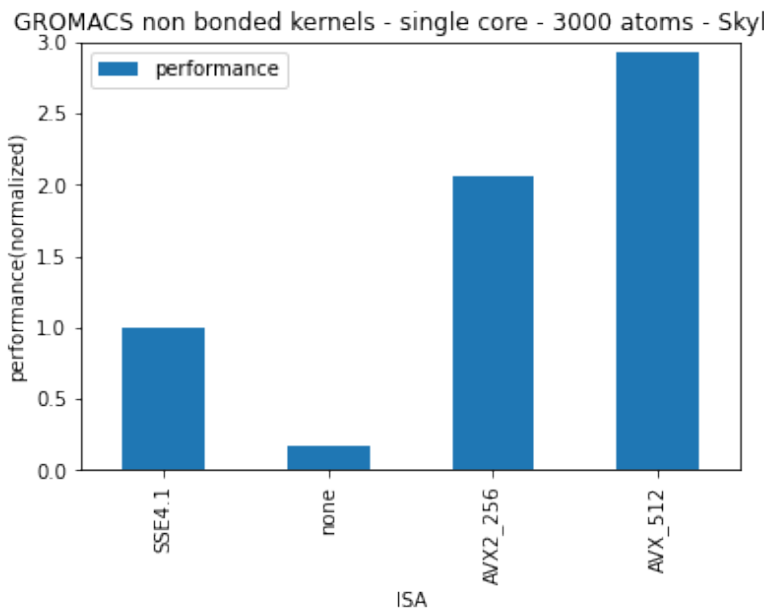
- Very popular
  - highest number of HPCI projects
  - Heavily used during the COVID’19 effort (team lead by Professor Okuno, Kyoto University)

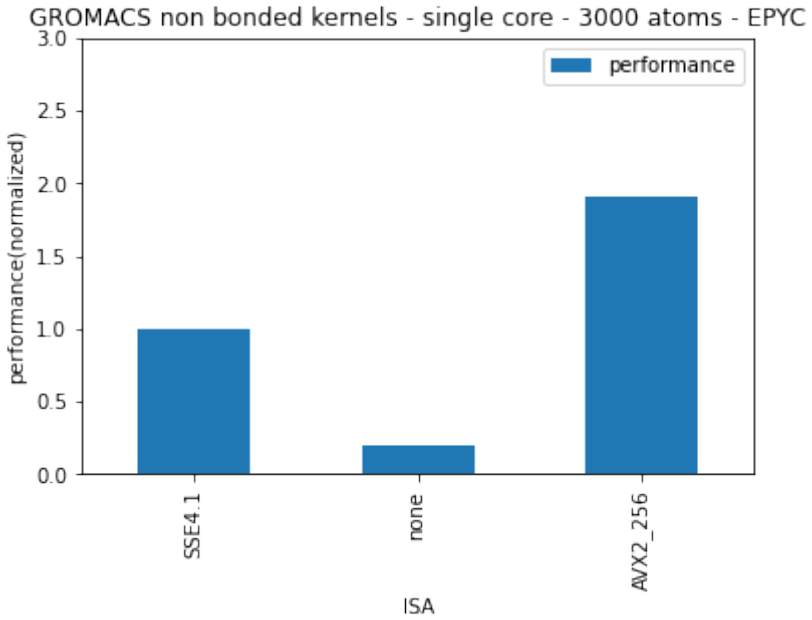
- Run on many platforms (Linux/Windows, CPU/GPU) and is heavily optimized

GROMACS has to be ported and optimized on Fugaku/A64fx

Vector Length matters (non bonded kernels)

- GROMACS was initially available on Arm-NEON (128 bits vectors)





- GROMACS was built on x86_64 platforms with all available architecture
  - No vectorization (e.g. the compiler will do everything) is highly suboptimal
  - Longest vectors means longest performances
  - Performance is improved by using the latest (and feature rich) ISA
- Naive expectation on A64fx (SVE 512 bits) is ~3x improvement vs the existing NEON architecture

GROMACS (from a software point of view)

- Modern C++ application
- Abstract vector registers
  - Vector operations are implemented by (inline) SIMD intrinsics
  - Operators are overloaded
  - Vector length is known at build time
- One implementation per architecture
  - x86_64 SSE
  - x86_64 AVX
  - x86_64 AVX512
  - Sparc HPC ACE
  - ARM NEON
  - **ARM SVE** had yet to be implemented
- Non bonded kernels single thread performance was identified as the primary target
  - Generally a hot spot in GROMACS simulations
  - Heavily rely on architecture specific implementation (e.g. SIMD intrinsics)
  - Most leverage

In a nutshell (ARM NEON)

```
#include <arm_neon.h>

class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};
```

In a nutshell (ARM NEON)

```
#include <arm_neon.h>

class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};

static inline SimdFloat
operator+(SimdFloat a, SimdFloat b)
{
    return {
                vaddq_f32(a.simdInternal_, b.simdInternal_)
    };
}
```

In a nutshell (ARM NEON)

```
#include <arm_neon.h>

class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};

static inline SimdFloat
operator+(SimdFloat a, SimdFloat b)
{
    return {
                vaddq_f32(a.simdInternal_, b.simdInternal_)
    };
}

static void dummy() {
    SimdFloat a, b, c;
    a = b + c;
}
```

Once upon a time...

- Porting started around ISC'19
  - Kickoff meeting with Professor Lindahl and team
- No SVE support in GROMACS at that time (only 128bits NEON vectors were supported)
- Very limited hardware and software environment
  - No processor yet (Fugaku was known as post-K)
  - Only ARM compilers were feature complete (ACLE intrinsics)
  - Compiler was VLA only
  - ArmIE (correctness) and gem5 (simulator)
- Main focus is the non bonded kernels
  - Heavily use vector abstraction framework
  - Ideally no changes are required

VLA or VLS ?

- SVE ISA enables writing Vector Length Agnostic (VLA) code
  - Same binary can run on any ARM+SVE processor
  - Use of masked instructions enable the code to be vectorized and to run efficiently on multiple architectures
- GROMACS is Vector Length Specific (VLS)
  - Vector lengths are hard coded
  - Some decisions are made at compile time
  - GROMACS supports selected vector lengths (2, 4, 8 or 16 elements per vector)
  - VLA code is at best on par with VLS code
  - VLA GROMACS would need involvement from the core developers
  - Only libgromacs.so uses SIMD instructions, binaries (e.g. gmx) do not
- Opted for Vector Length Specific GROMACS

Initial implementation (ARM compilers)

```
class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};
```

NEON

```
class SimdFloat
{
    public:
        SimdFloat() {}

        svfloat32_t simdInternal_;
};
```

SVE

Initial implementation (ARM compilers)

```
class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};
```

NEON

```
class SimdFloat
{
    public:
        SimdFloat() {}

        svfloat32_t simdInternal_;
};
```

SVE

```
typedef __sizeless_struct SimdFloat {
    SimdFloat() {}

    svfloat32_t simdInternal_;

} SimdFloat;
```

Initial implementation (ARM compilers)

```
class SimdFloat
{
    public:
        SimdFloat() {}

        float32x4_t  simdInternal_;
};
```

NEON

```
class SimdFloat
{
    public:
        SimdFloat() {}

        svfloat32_t simdInternal_;
};
```

SVE

```
typedef __sizeless_struct SimdFloat {
    SimdFloat() {}

    svfloat32_t simdInternal_;

} SimdFloat;
```

ACLE

```
SimdFloat vec[3];
```

NEON

```
SimdFloat vec[3];
```

Initial port

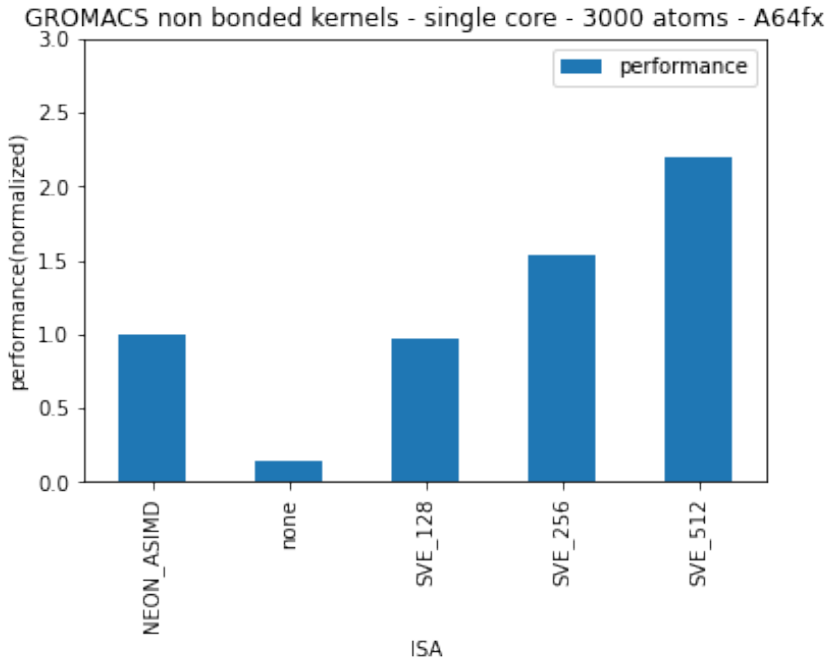
- ArmIE (SVE emulation) was extremely helpful
  - Initial port was developed on ThunderX2 (NEON only)
  - Builtin `make simd-test` was leveraged to validate the correctness of the port
  - Correctness only (e.g. no performance indication)
- SVE ISA (virtually all instructions take a mask) is a very good fit for GROMACS
- Gem5 simulator was used to start optimizing
  - BenchMEM benchmark (one iteration tooks a few ms on native hardware)
  - Takes > 7 hours on the simulator (virtually unusable)
  - Manually trim down to one function invokation and reduce the simulation down to 40 minutes
- Interesting tool to validate different implementations
  - No OS noise, so very easy to figure out what works and what does not

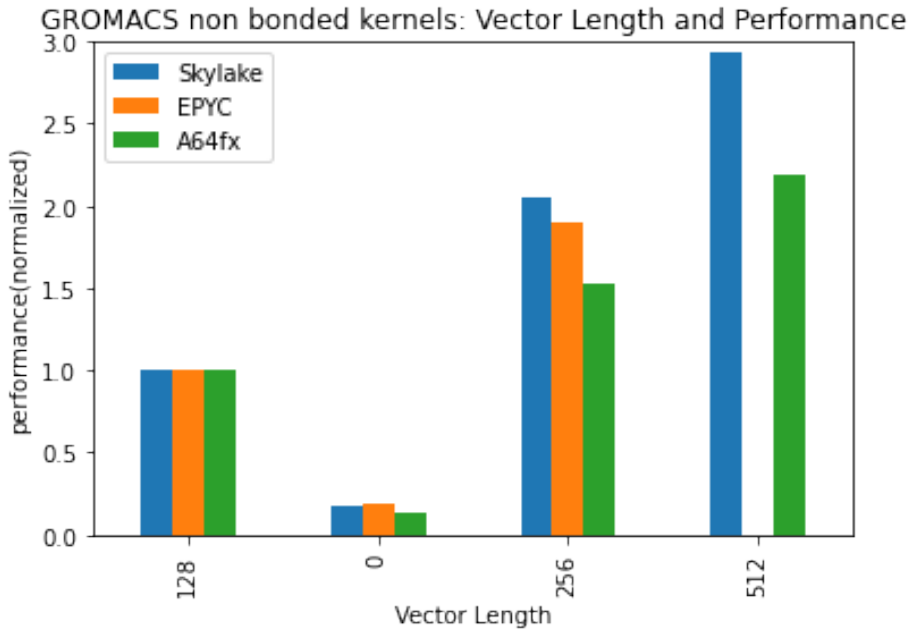
A long and bumpy road

- A working port was pretty straightforward
- Several attempts were made to optimize GROMACS on A64fx
- Try different GROMACS optimizations
  - Some can be hard to upstream (too much arch specific)
  - Outcome depends on both compiler and compile flags
- Try different compilers (ARM, Fujitsu, LLVM and GNU)
  - ACLE SIMD intrinsics was initially only supported by ARM compilers
  - Several bugs (quickly solved though) (GNU: 7, LLVM: 6, vendors: a few)
  - Performance wise YMMV
    - LLVM 12 (awful) vs LLVM 14 (great)
    - GNU compilers are good and stable
    - ARM compilers are now the best (and they are free now!)

Initial performance evaluation

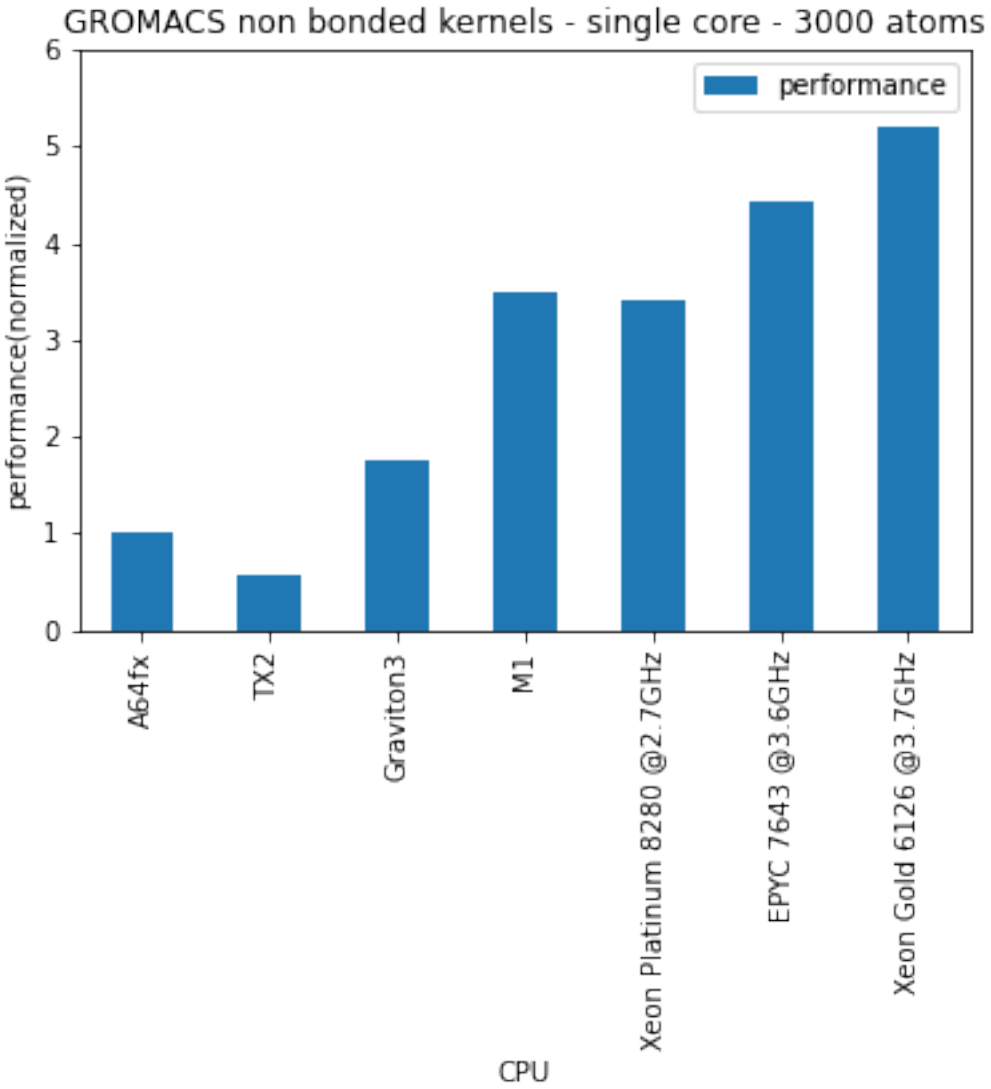
- NEON and 128 bits SVE are on par
- 512 bits SVE performance improvement was disappointing
- With a real world dataset (one node, 48 OpenMP threads)
  - NEON vectorization : 13.9 ns/day
  - SVE vectorization : 16.6 ns/day
  - SVE vectorization + GROMACS optimizations : 18.0 ns/day





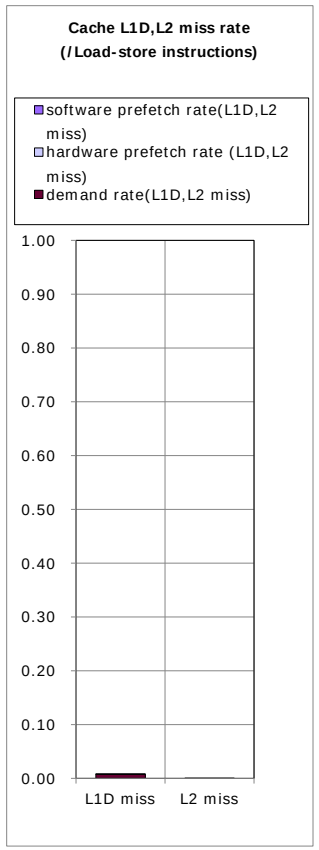
Single thread performance comparison

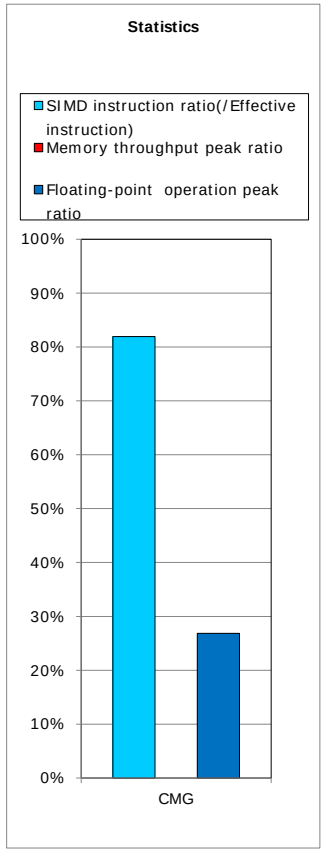
- GROMACS (non bonded kernels) performances were not very competitive on A64fx
  - Not an ARM vs x86_64 issue (M1 is pretty good)
- Need to understand what is limiting performances

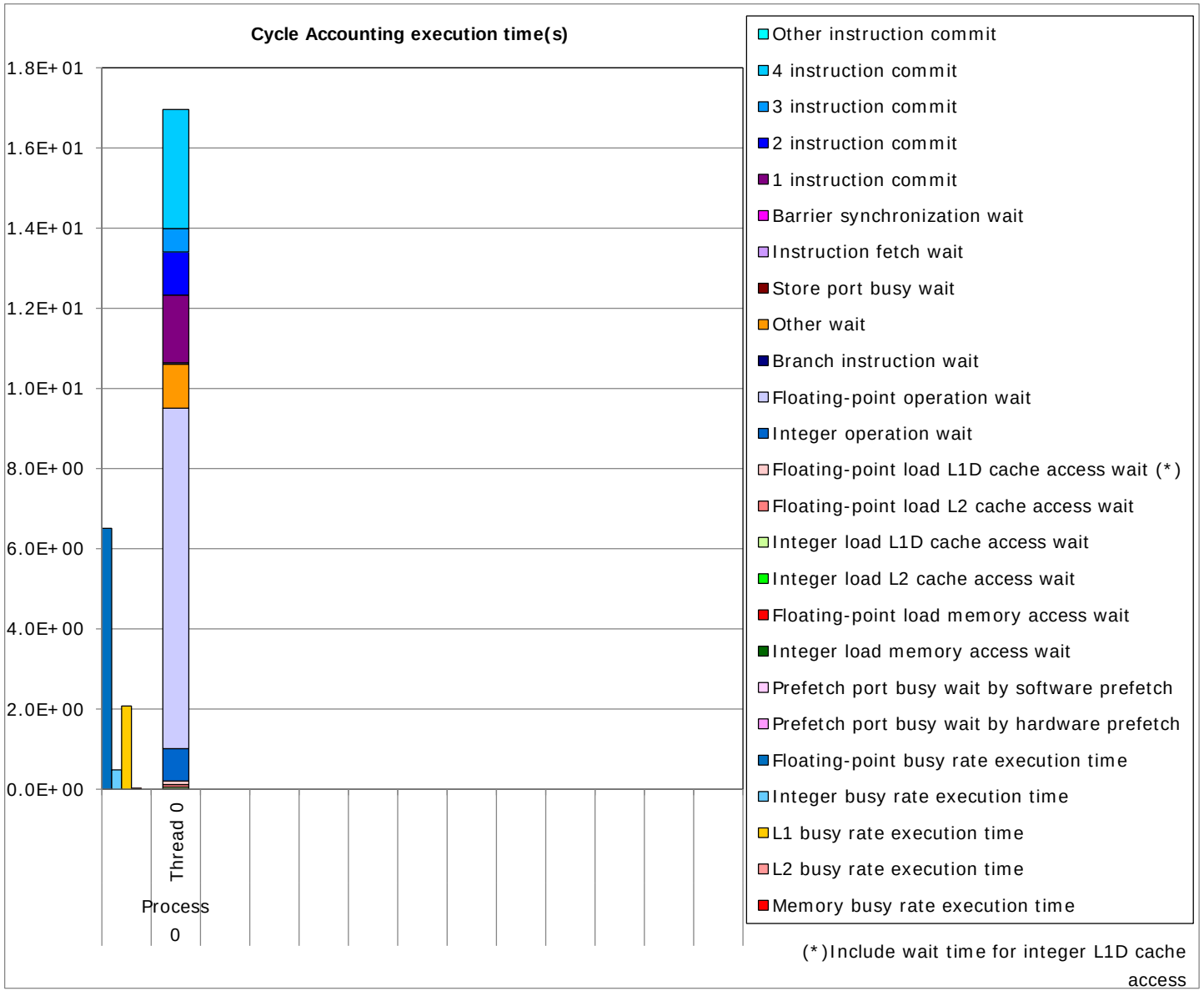


Nonbonded kernels performance on A64fx

SIMD instruction rate (%) (/Effective instruction)	SVE operation rate (%)	Floating-point pipeline Active element rate (%)	IPC
81.93%	99.99%	91.42%	**1.03**



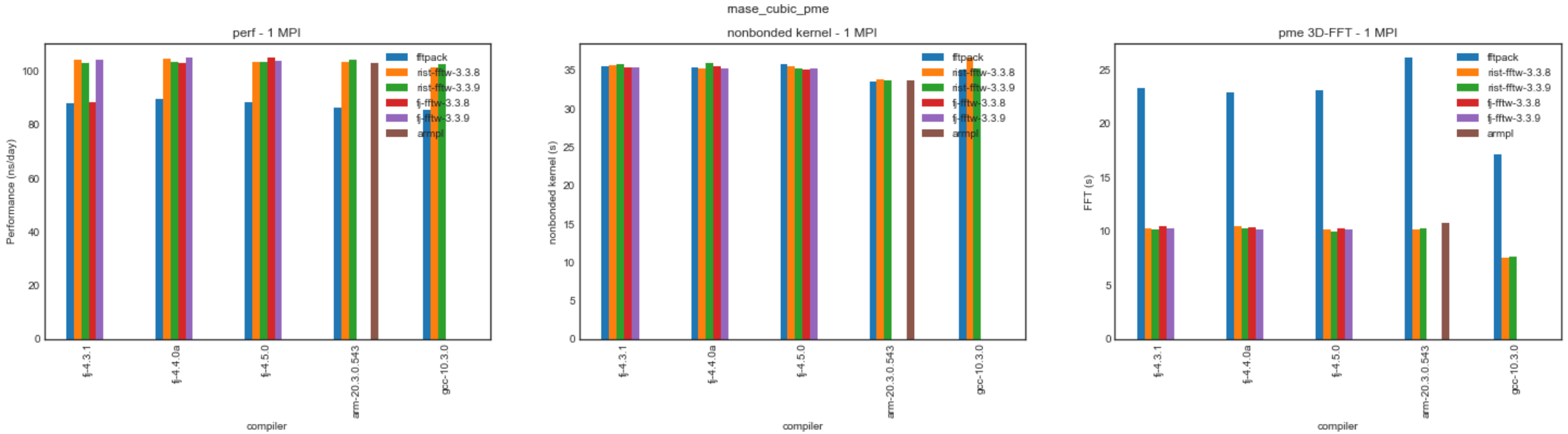




Performance analysis

- Some very good metrics
  - High SIMD rate
  - SVE only
  - No cache miss
- But
  - Low Instruction per Cycle (IPC): ~1 (ideally ~4)
  - More than half the cycles are spent waiting on the Floating Point Unit (FPU)
- Interpretation
  - GROMACS is a complex workflow mainly consisting of a long list of instructions depending of each other
  - The relative high instruction latency of A64fx severely limits performances
  - The relative short Out-of-Order (OoO) pipeline is helpless
  - Compiler does not seem to optimize “manually vectorized code” (e.g. no software pipelining)

A typical GROMACS benchmark (PP + PME)



- Overall performance (with optimized FFT) is pretty similar between various compilers
  - Optimized FFTW and upstreamed it https://github.com/FFTW/fftw3/pull/315
- A close look shown some room for improvement
  - ARM compilers are better for the non bonded kernels
  - GNU compilers are better for the PME kernels (including FFTW)
  - FJ compilers are not that bad and do not require any additional runtime dependencies
- Building an hybrid GROMACS allowed to squeeze some extra performances

Revisiting the low IPC issue

- Data is generally already in L1 cache
- Strong data dependency between instructions
- load/store is not much slower than compute => textbook software pipelining (SWP) not really helpful and would be hard to implement in GROMACS
- Loop unrolling does not help: short OoO pipeline does not unlock data parallelism

```
for(int i=0; i<N; i++) {
   int c = a[i];          // load
   c = c + 1;             // compute
   c = c + 2;             // compute
   c = c + 3;             // compute
   c = c + 4;             // compute
   a[i] = c;              // store
}
```

```
for(int i=0; i<N-1; i+=2) {
   int c0 = a[i];
   c0 = c0 + 1;
   c0 = c0 + 2;
   c0 = c0 + 3;
   c0 = c0 + 4;
   a[i] = c0;
   int c1 = a[i+1];
   c1 = c1 + 1;
   c1 = c1 + 2;
   c1 = c1 + 3;
   c1 = c1 + 4;
   a[i+1] = c1;
}
```

Unroll and interleave (1/2)

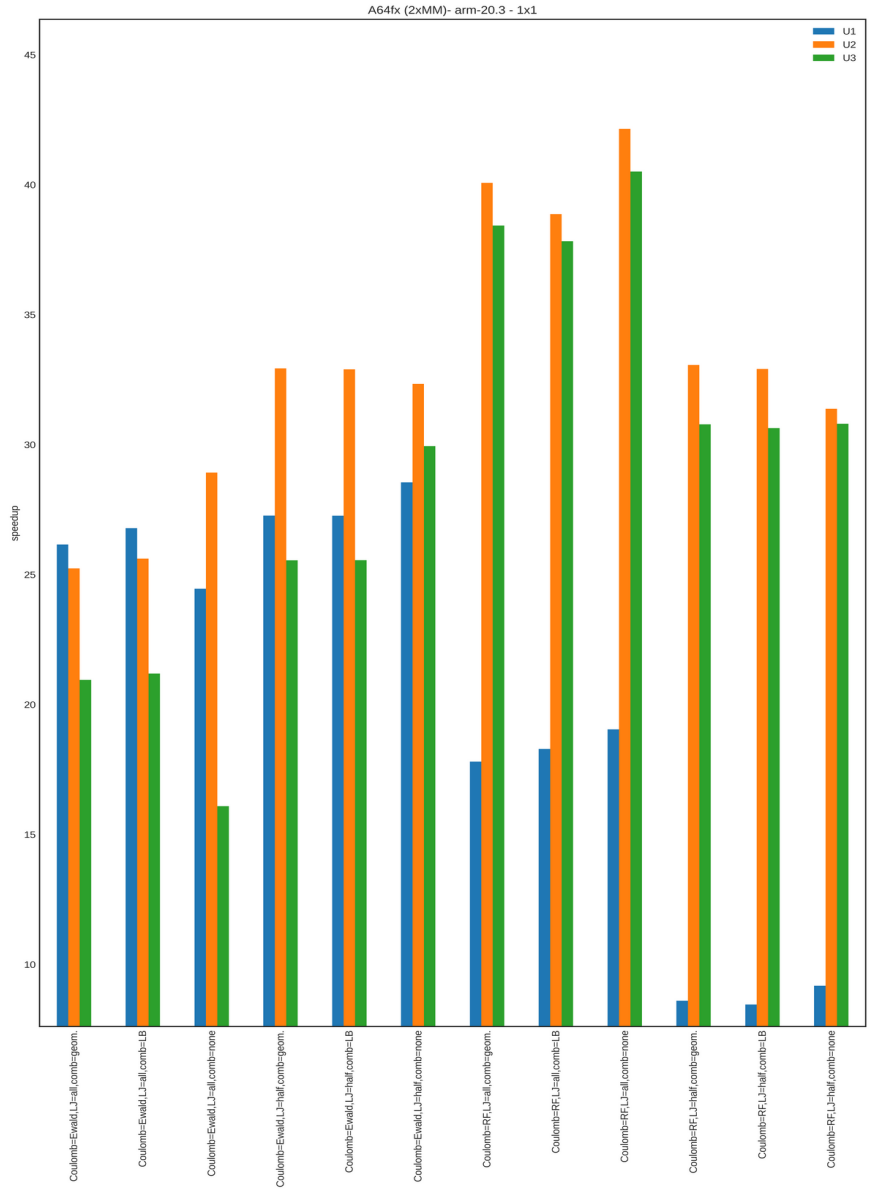
- Unroll and interleave instructions in order to evidence data parallelism, even with a short OoO pipeline

```
for(int i=0; i<N; i++) {
   int c = a[i];          // load
   c = c + 1;             // compute
   c = c + 2;             // compute
   c = c + 3;             // compute
   c = c + 4;             // compute
   a[i] = c;              // store
}
```

```
for(int i=0; i<N-1; i+=2) {
   int c0 = a[i];
   int c1 = a[i+1];
   c0 = c0 + 1;
   c1 = c1 + 1;
   c0 = c0 + 2;
   c1 = c1 + 2;
   c0 = c0 + 3;
   c1 = c1 + 3;
   c0 = c0 + 4;
   c1 = c1 + 4;
   a[i] = c0;
   a[i+1] = c1;
}
```

Unroll and interleave (2/2)

- Counter-intuitive since this requires more registers
  - Registers spilling before
  - Likely even more registers spilling
- Eventually quite effective with GROMACS on A64fx



Neoverse V2: SVE or NEON?

- NVIDIA Grace and Amazon Graviton 4 (based on the Arm Neoverse V2 architecture) support SVE with 128 bits vectors
- Should NEON or SVE be used?
- Ongoing discussion at https://gitlab.com/gromacs/gromacs/-/merge_requests/5147
- SVE gets selected by default
- NEON is currently faster than SVE (2% on non bonded kernels)
  - Currently best to force cmake -DGMX_SIMD=ARM_NEON_ASIMD)
- Not a NEON vs SVE strictly speaking
- Some discrepancies between NEON and SVE handling in GROMACS
  - Fixing these makes SVE 1% faster than NEON
- Using SVE with a few subroutines still using NEON improves performances
  - Making SVE 2.5% faster than NEON

Conclusions

- GROMACS has been successfully ported to A64fx (and the SVE extensions)
  - Port was relatively straightforward
  - Rich Arm software ecosystem was leveraged
- Optimization was challenging
  - Very sensitive to compilers and compilers options
  - GROMACS non bonded kernels are complex and CPU intensive, and hence not an ideal fit for the A64fx micro-architecture (short OoO pipeline and high latency instructions)
- SVE for FFTW is now upstream, will be available in FFTW 3.3.11
- Two not so common ways to improve performances:
  - Mix compilers and leverage the rich Arm ecosystem
  - Unroll and interleave improved the nonbonded kernels perfomances by ~50% on A64fx