FFT W3: LEVERAGING THE SCALABLE VECTOR EXTENSION (SVE)

SEPTEMBER 2019
FRAMEWORK PARTNERSHIP AGREEMENT IN EUROPEAN LOW-POWER MICROPROCESSOR TECHNOLOGIES

THIS PROJECT HAS RECEIVED FUNDING FROM THE EUROPEAN UNION'S HORIZON 2020 RESEARCH AND INNOVATION PROGRAMME UNDER GRANT AGREEMENT NO 826647
“FASTEST FOURIER TRANSFORM IN THE WEST”

- FFTW, “Fastest Fourier Transform in the West”, is a library implementing Fourier Transform and approximations thereof and self-adapt to the hardware
- It has been around for over two decades
- Widely in use in the HPC community
  - Intel’s MKL include an FFTW-compatible interface
- Version 3 of the library, FFTW3, was designed in the early 2000s, and added support for abstract SIMD support
  - Originally SSE/SSE2 (x86) & AltiVec (PowerPC)
- Since then, support was added for NEON, AVX*, etc.
FTTS IN FFTW3

- FFTW3 user interface has two stages
  1. “Planning” the transform
  2. “Executing” the transform
- The “plan” is an executable data structure that accepts the input data and computes the desired DFT [1]


- This data structure includes
  1. Small, executable transforms called “codelets” whose code is generated by a specialized code generator, “genfft”
  2. Algorithms to combine those “codelets” in larger transforms, such as “Cooley-Tukey”

- In FFTW3, the “codelets” can be designed for SIMD
SIMD CODELETS

- “genfft” uses multiple algorithms to generate “codelets”
  - Small full FFTs, Cooley-Tukey, Bluestein, Rader, ...
- It can generate multiple variants for a single size of a single algorithm
  - e.g. favoring Fused-Multiply-Add (FMA) or not, etc.

- One possible option is to generate SIMD code
- This code is purely abstract, it doesn’t implement code for any architecture
- The code uses C macros as placeholder for actual computations
- An “implementation header” is needed to describe a specific architecture and implement the macros
THE IMPLEMENTATION HEADER

- The “implementation header” implements the required macro (or function) to support an SIMD architecture
- It specifies register width, how to compute/load/store vectors, etc.
  - That first one – register width – is going to haunt us later...
- What we need for SVE is “only” the implementation header

AVX-512

/* SIMD complex vector length – FP64, FP32 */
#define VL DS(4, 8)
/* permute the Real and Imaginary part of each complex */
#define FLIP_RI(x) SUFF(_mm512_shuffle)(x, x, DS(0x55, 0xB1))
/* element-wise FMA */
#define VFMA(a, b, c) SUFF(_mm512_fmadd)(a, b, c)
/* similar, with a == ‘i’ */
#define VFMAI(b, c) SUFF(_mm512_fmaddsub)(VLIT1(1.), c, FLIP_RI(b))
FFT W3 CODE FLOW

- The codelet generator “genfft” generates C codelets with SIMD macro
  - And regular C-only, scalar codelets
- The codelets include the implementation header file to produce SVE binary codelets
  - The regular C codelet are compiled normally
- The SVE binary codelets & other codelets are combined with the infrastructure library to produce the SVE-enabled library
USING SVE FOR FFTW3 SIMD

- SVE has one specificity not in SSE, AVX, AltiVec, ... : it is *scalable*
- That means, the vector width is not known at compile time – it can be anything from 128 to 2048 bits, in steps of 128 bits
- There is no way to define the VL macro the way it is done for other SIMD ISA
- And the value is hardwired in other places in the infrastructure as well...

- SVE includes specific instructions to support *complex arithmetic*, the basis for FFTW3’s computations
- It should help with the implementation of some of the implementation macro
- SVE also has lane masking, which is useful for auto-vectorization but can also be useful in our case...
  - But sometimes must use a mask as there is no unmasked version... and vice versa
SVE FOR FFTW3, FIRST VERSION (1)

- The easiest way to deal with the scalability problem is to ignore it...
  1. Assume a vector width, such as 512 bits
     - e.g. for the Fujitsu A64FX
  2. At runtime, in the infrastructure, only enable the codelets if the hardware vector width matches the assumed vector width

```c
/* FIXME: this hardwire to 512 bits */
#define VL DS(4, 8)
/* permute the Real and Imaginary part of each complex */
#define FLIP_RI(x)
TYPE(svtrn1)(TYPE(svtrn2)(x,x),x)
/* element-wise FMA */
#define VFMA(a, b, c)
TYPESUF(svmad,_z)(ALLA,b,a,c)
/* similar, with a == ‘i’ */
#define VFMAI(b, c)
TYPESUF(svcadd,_z)(ALLA,c,b,90)
```

- all-1 mask macro
- zeroing masked values to avoid false dependencies
- dedicated addition with 90° complex rotation, i.e. multiplications by ‘i’
SVE FOR FFTW3, FIRST VERSION (2)

- This code will only be enabled when the hardware register width is exactly X bits
- Cannot work on smaller register, as some macro loads load values from array – and the array content is register width-dependent
- Could work with larger register if we used masking...
- Implemented for 256 & 512 bits using Arm C Language Extension (ACLE)

- Code is tested using the Arm Instruction Emulator for validity
  - Needs to disable performance measurement as emulated SVE is much slower than NEON!
- Also somewhat tested using the QEMU & GEM5 simulators
  - More accurate but even slower
  - Timing is also somewhat questionable at the moment
- Hopefully, hardware access soon ;-)

https://github.com/rdolbeau/fftw3/tree/arm-sve
SVE FOR FFTW3, SECOND VERSION (1)

- Makes the implementation more tolerant to register width
  
1. Assume a vector width, such as 512 bits, and make the code works for register at least as wide
   - Masking is great!

2. At runtime, in the infrastructure, only enable the codelets if the hardware vector width is equal or larger than the assumed vector width

SVE, masked

/* FIXME: this hardwire to 512 bits */
#define VL DS(4, 8)
/* permute the Real and Imaginary part of each complex */
#define FLIP_RI(x)
TYPE(svtrn1)(TYPE(svtrn2)(x,x),x)
/* element-wise FMA */
#define VFMA(a, b, c)
TYPESUF(svmad,_z)(MASKA,b,a,c)
/* similar, with a == 'i' */
#define VFMAI(b, c)
TYPESUF(svcadd,_z)(MASKA,c,b,90)
SVE FOR FFTW3, SECOND VERSION (2)

- This code will only be enabled when the hardware register width is at least X bits
- Implemented for 128, 256 & 512 bits
- 512 bits hardware can use all three groups, in case shorter vector are somehow faster
  - Cache management, small dimension(s), etc.
  - FFTW3 is self-tuning, more options is better – but planning is slower
- Same testing as for the fixed-width version
- Requires Arm HPC Compiler 19.3 or newer
  - Small code generation bug in 19.2 and earlier
- Similar binary output to the first version, more versatile at runtime

https://github.com/rdolbeau/fftw3/tree/arm-sve-alt
LESSONS LEARNED

- The Scalable Vector Extension is nice to work with 😊
  - But ACLE so far only available in Arm HPC compiler

- All the required features are there: computations – including complex arithmetic, data management, masking, etc.

- Compiler are starting to auto-vectorize using SVE, no need to code by hand for many codes

- Scalability is new, and code infrastructure may need to adapt
  - e.g. FFTW3 would require changes at a higher level, as it expects fixed-amount-of-work codelet at this time

- Other algorithm much more amenable, e.g. Chacha20 crypto algorithm implemented in the Supercop benchmark

https://bench.cr yp.to/supercop.html
crypto_stream/chacha20/dolbeau/arm-sve
EPI PARTNERS