LLVM and the automatic vectorization of loops invoking math routines: `-fsimdmath`

1st Francesco Petrogalli
Arm Ltd.
Cambridge, United Kingdom
francesco.petrogalli@arm.com

2nd Paul Walker
Arm Ltd.
Manchester, United Kingdom
paul.walker@arm.com

***Abstract*—The vectorization of loops invoking math function is an important optimization that is available in most commercial compilers. This paper describes a new command line option, `-fsimdmath`, available in *Arm Compiler for HPC* [1], that enables auto-vectorization of math functions in C and C++ code, and that will also be applicable to Fortran code in a future versions.**

The design of `-fsimdath` is based on open standards and public architectural specifications. The library that provides the vector implementation of the math routines, `libsimdmath.so`, is shipped with the compiler and based on the *SLEEF* library `libsleefgnuabi.so`. *SLEEF*[1] is a project that aims to provide a vector implementation of all C99 math functions, for a wide variety of vector extensions and architectures, across multiple platforms.

This feature is very important for HPC programmers, because the vector units of new CPUs are getting wider. Whether you are targeting Intel architectures with the AVX512 vector extension, or Arm architectures with the Scalable Vector Extension, good quality auto-vectorization is of increasing importance.

Although `-fsimdmath` has been implemented in a commercial compiler, it has been designed with portability and compatibility in mind, so that its use is not limited only to the vector extensions of the Arm architectures, but can be easily introduced as a major optimization for all the vector extensions that LLVM supports.

If accepted upstream, this new feature will enlarge the set of loops that LLVM will be able to auto-vectorize.

***Index Terms*—compilers, vector math routines, SIMD, Arm, SVE, fsimdmath, SLEEF**

I. Introduction

Arm Compiler for HPC is the LLVM-based compiler that Arm distributes to enable the development of High Performance Computing applications on supercomputers based on the ARMv8 architecture.

The compiler is packaged together with a set of libraries and tools that allow programmers to achieve the best performance on the target machine.

One key optimization that HPC compilers like Intel C Compiler and PGI Compiler have added to their tool-chain is the auto-vectorization of loops that invoke the math routines defined in the C99 standard. The same optimization is available for *Arm Compiler for HPC* when targeting the Advanced SIMD vector extension (NEON) and the Scalable Vector Extension (SVE) [3] of ARMv8.[2]

This feature is very important in the HPC domain, because many HPC applications rely on expensive computation that use the math routines.

The automatic vectorization of the math routines performed by *Arm Compiler for HPC* is completely transparent to the user, and requires only the addition of the command-line option `-fsimdmath`.

This article will cover the following topics:

1) The user interface.
2) How the functionality is implemented in the compiler.
3) The library used to provide the math routines.
4) Application speedup examples.
5) Results and analysis of the implementation.

A section on the future development of the feature describes:

1) How Arm would like to to contribute the functionality back to the open source version of `clang`, in collaboration with the community.
2) How the SLEEF project can enable the same functionality for most of the architectures supported by LLVM.

II. Usage

The automatic vectorization of math routines is enabled by invoking the compiler with the command-line option `-fsimdmath`, together with the standard command-line options to enable auto-vectorization, `-O2` and above.

The complete syntax of the option is shown is figure 1. The optional value assigned to `-fsimdmath` is used to specify which library the linker needs to use, to find the symbols of the vector functions that are generated by the compiler. The value of `-fsimdmath` is set by default to `simdmath`, so that a plain invocation of `-fsimdmath` passes `-lsimdmath` to the linker, and therefore links the user code against `libsimdmath.so`. Any other value `-fsimdmath=<X>` is passed to the linker as `-l<X>` for linking the code against `lib<X>.so`, which must be visible in the search path of the linker.

When the compiler is invoked with the math function auto-vectorization feature on a loop that invokes a math function

[1]https://sleef.org

[2]https://developer.arm.com/products/software-development-tools/hpc/documentation/vector-math-routines

```
$ armclang[++] -fsimdmath{=[<vector math library> | simdmath]}
```

Fig. 1. Command-line interface for the `-fsimdmath` option. `simdmath` is the default value if no value is specified.

```
#include <math.h>

double foo(double * restrict x, double * restrict y, unsigned n) {
  for (unsigned i = 0; i < n; ++i)
    y[i] = sin(x[i]);
}
```

Fig. 2. Scalar loop with a call to `sin`.

like the one in figure 2, it converts the original scalar loop to a vector loop, inserting an appropriate call to the vector function that corresponds to the original scalar function. The function call that the compiler generates depends on the vector extension that the compiler is targeting, whether NEON or SVE, plus the usual parameters that are involved in auto-vectorization, such as the number of concurrent lanes on which the vector loop will operate, or whether or not masking is needed.

For example, when targeting NEON vectorization of the code in figure 2, the function call that the compiler generates in the auto-vectorized loop is `_ZGVnN2v_sin`, which operates concurrently on two elements loaded from the array `x`. The NEON assembly code of the loop block generated by the compiler can be seen in figure 3.

When targeting SVE vectorization, the function call that the compiler generates is `_ZGVsMxv_sin`, which is a vector-length agnostic (VLA) vector function that operates on up to as many elements as an SVE implementation can carry.[3] The SVE assembly code generated by the compiler in this case can be seen in figure 4. Notice that the vector-length agnostic instruction set architecture means that there is no loop tail or hard-coded vector width, and the ABI chosen for these vectorized math functions must be usable in this context.

The name of the vector functions generated by the compiler in the vector loop conforms to the mangling scheme of the *Vector function ABI specification for AArch64 (VABI)* [2]. An explanation of the rules of the VABI is given in the appendix at the end of this paper.

All the vector function calls that the compiler generates are provided via `libsimdmath.so`, a shared library that is packaged with the tool-chain.

III. Design and implementation

The extensions to LLVM to support the auto-vectorization of math functions have been designed around two standards:

- The `declare simd` directive of the OpenMP[4] standard is used to inform the compiler about the availability of vector math functions.
- The *Vector function ABI specifications for AArch64* is used to define the interface between the programs generated by the compiler and the vector math library.

The implementation required a set of changes in LLVM and `clang`, together with the selection of an appropriate vector math library, which is included with the compiler.

A. Changes in `clang` and LLVM

The implementation of `-fsimdmath` required modifications to several software components of the compiler:

1) The front-end `clang`.
2) The Target Library Info (TLI) in LLVM.
3) The Loop Vectorizer (LV) pass in LLVM.

The interaction of the three components is as follows.

1) A wrapper around the system header file `math.h` informs `clang` of the availability of the vector math routines in `libsimdmath.so`. The header is shipped with the compiler and is added to the sources of `clang` in `<clang>/lib/Headers/math.h`. It contains a set of additional declarations of the math functions decorated with the appropriate `declare simd` directive. The new declarations are added after loading the standard `math.h` header file available in the system. The `declare simd` directives reflect the vector version available in the library.
2) The OpenMP code generator module of `clang` populates the IR module with a list of global names that carry the information about the available vector functions, their mangled name and the signature. This information is generated according to the *Vector function ABI specification for AArch64*. The scalar-to-vector mapping rules of the ABI have been added to the code OpenMP code generator module of `clang`.
3) The list of vector names and vector signatures is passed to the TLI via the BackendUtils module of `clang`. The

[3]More information on SVE and the VLA vectorization techniques can be found at https://developer.arm.com/products/software-development-tools/hpc/sve.

[4]The `declare simd` directive of the OpenMP standard is supported since version 4.0. For more information, see https://www.openmp.org/.

```
$ armclang -O2 -fsimdmath -S -o - foo.c

// ... loop header ...
.LBB0_9:                                // vector loop body
        ldp     q0, q16, [x24, #-16]
        bl      _ZGVnN2v_sin
        mov     v17.16b, v0.16b
        mov     v0.16b, v16.16b
        bl      _ZGVnN2v_sin
        stp     q17, q0, [x25, #-16]
        add     x24, x24, #32
        subs    x26, x26, #4
        add     x25, x25, #32
        b.ne    .LBB0_9
// ... loop tail ...
```

Fig. 3. Auto-vectorized `sin` call for NEON generate from the code in figure 2.

```
$ armclang -mcpu=armv8-a+sve -O2 -fsimdmath -S -o - foo.c

// ... loop header
.LBB0_5: // Vector loop body
        ld1d    {z0.d}, p0/z, [x20, x22, lsl #3]
        str     p0, [x29, #8, mul vl]
        bl      _ZGVsMxv_sin
        ldr     p0, [x29, #8, mul vl]
        st1d    {z0.d}, p0, [x19, x22, lsl #3]
        incd    x22
        whilelo p0.d, x22, x21
        b.mi    .LBB0_5
```

Fig. 4. Auto-vectorized `sin` call for SVE generated from the code in figure 2.

TLI uses it to populate a table that holds three fields that describe the mapping between the scalar function and the vector function. The table provides the original name of the scalar function, and the mangled name and the signature of the vector function associated to it.

4) The LV pass is provided with an interface that enables it to query the TLI based on the original scalar name and the expected signature of the vector function.

For example, figure 5 shows the re-declaration of `sin` in `lib/Headers/math.h`.

From the re-declarations of `sin` in figure 5, the compiler populates the dynamic table of the TLI as in table I.

The LV pass then determines if a scalar function can be vectorized by looking at the scalar-to-vector mappings in the TLI interface, searching for the scalar name and the expected vector signature. If a match is found, the vector name and the vector signature are used to generate the function call in the vector loop.

A diagram of the components of the implementation and their interactions is shown in figure 6.

B. The library `libsimdmath.so`

As stated previously, the default behaviour of `-fsimdmath` is to assume that a `libsimdmath.so` library is available that implements the required vector math functions. The implementation used in *Arm compiler for HPC* is based on *SLEEF*, an open source project developed by Naoki Shibata of the Nara Institute of Science and Technology (NAIST)[5] in Japan. The project is a joint collaboration between NAIST, Arm, and other industrial partners. The project is maintained by a small community, with some occasional contributors that are using SLEEF in other open source or commercial projects. The project is gaining more and more attention from the open source community, and it is now part of the *experimental* release of Debian and Ubuntu.[6]

[5]http://www.naist.jp

[6]The packages `libsleef3` and `libsleef-dev` are officially part of Debian unstable, for all the architectures that Debian supports. See https://packages.debian.org/unstable/libsleef3. The maintainer of the Debian package provides also an unofficial PPA for Ubuntu at https://launchpad.net/~lumin0/+archive/ubuntu/sleef.

```
/* system math.h inclusion */
#include_once <math.h>
#ifdef __cplusplus
extern "C" {
#endif

#if defined(__ARM_NEON) && !defined(__ARM_FEATURE_SVE)
  #pragma omp declare simd simdlen(2) notinbranch
  double sin(double);
#endif

#if defined(__ARM_FEATURE_SVE)
  #pragma omp declare simd
  double sin(double);
#endif

#ifdef __cplusplus
}
#endif
```

Fig. 5. Redefinition of `sin` in `math.h`.

Scalar Name	Vector Name	Vector Signature (`llvm::FunctionTy *`)
`sin`	`_ZGVnN2v_sin`	`<2 x double>(2 x double)`
`sin`	`_ZGVsMxv_sin`	`<n x 2 x double>(n x 2 x double)`

TABLE I
SCALAR FUNCTION TO VECTOR FUNCTION MAPPING.

Arm choose *SLEEF* to provide the vector math routines for a couple of reasons. First, it is very easy to add new vector extensions to it. In fact, it is just a matter of adding a header file that maps the generic C intrinsics used to program the math routines to the target specific C intrinsics. As an example of this, let's consider the function `vdouble vadd_vd_vd_vd(vdouble, vdouble);` that adds two vectors in the target independent C intrinsic language of the library. The mapping of this intrinsic to a target specific C intrinsic for SVE is done as shown in figure 7.

With this mechanism, adding a new target is just a matter of preparing a header file with all the mappings between the core operation of the library and the target C intrinsics, and including it in the sources of the library. Implementations are already available for:

Intel
: SSE2, SSE4.1, AVX, FMA4, AVX2+FMA3, AVX512F.

ARMv8
: Advanced SIMD (Neon), SVE.

ARMv7
: Neon.

IBM Power
: VSX.

Another advantage of choosing *SLEEF* is that its design fits the VLA programming model of SVE. In fact, the size of the vectors used in the library is not encoded anywhere in the sources, other than in the `typedef` re-mapping of the target-specific header file, and the size it is not exposed in the user interface.

Finally, as shown in figure 8, the library has performance numbers comparable to a number of other industrial projects, like *SVML*,[7] and has been successfully used in the *Blue Gene/Q* compiler `bgclang` compiler [5] developed at the Argonne National Lab.[8]

Our feature requires that the vector math library conforms to the *GNU* ABI, and *SLEEF* provides the `libsleefgnuabi.so` variant that does this.

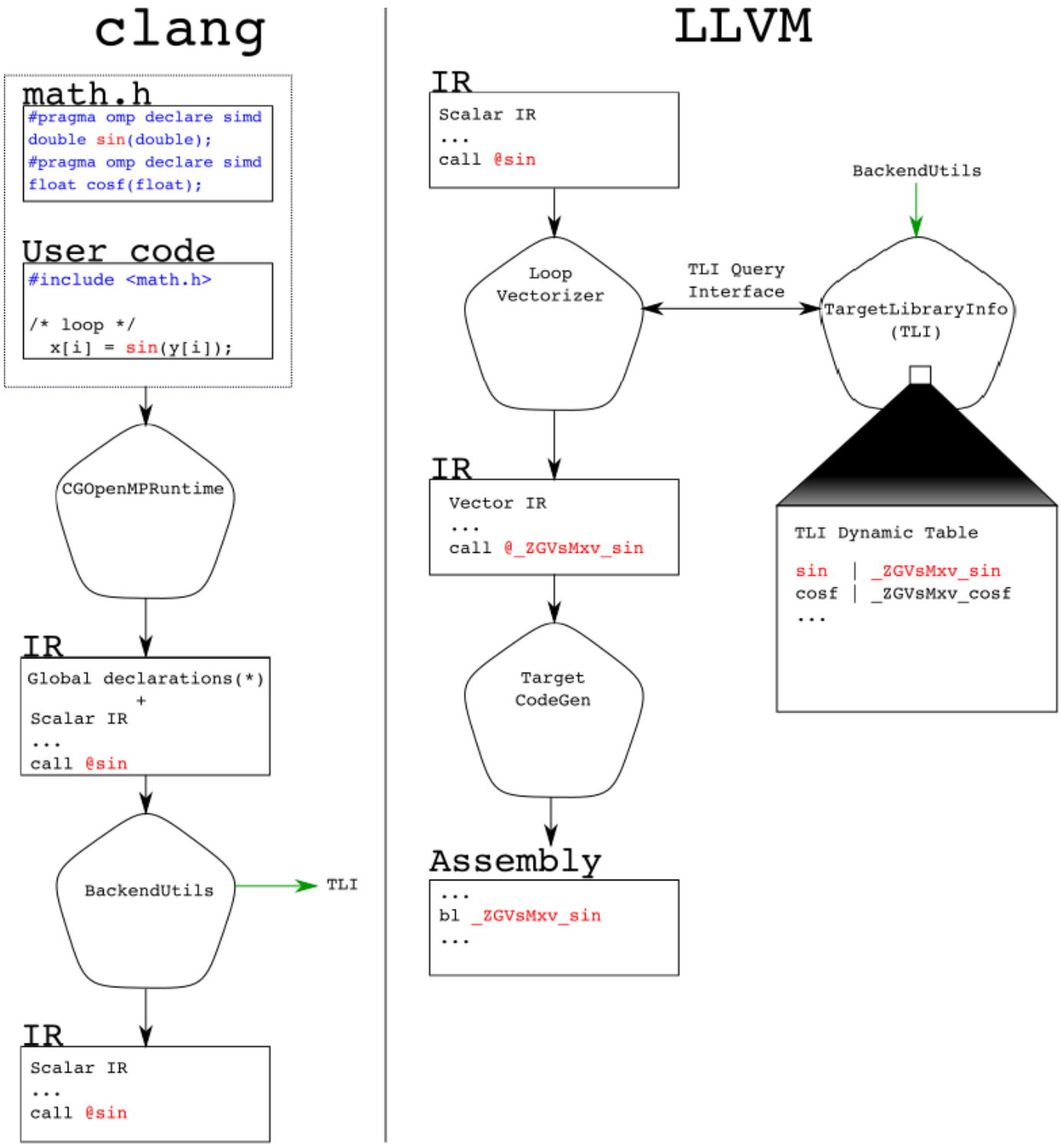
IV. RESULTS

The implementation of this functionality in *Arm compiler for HPC* is in its early stages. The focus has been to provide a functional end-to-end implementation of the capability, rather than on performance. Moreover, the runtime library `libsimdmath.so` is not tuned to fully exploit the capability of the vector extensions of AArch64. We expect sensible improvements on a wide range of programs when we start to make performance improvements in the future.

Additionally, the cost model of the LV needs to be tweaked so that it is able to decide whether or not the vectorization of a function call is going to be beneficial.

[7] ⟨https://software.intel.com/en-us/node/524289

[8] https://www.alcf.anl.gov/user-guides/bgclang-compiler.




```
(*)
declare <n x 2 x double> @vec_prefix__ZGVsMxv_sin_vec_midfix_sin_vec_postfix(<n x 2 x double>, <n x 2 x i1>)
declare <n x 4 x float> @vec_prefix__ZGVsMxv_cosf_vec_midfix_cosf_vec_postfix(<n x 4 x float>, <n x 4 x i1>)
```

Fig. 6. Diagram of the components of `-fsimdmath` in `clang` and LLVM, and their interactions. The independence between the frontend and the backend is broken by the link between the `BackendUtils` and the `TargetLibraryInfo` modules (in green). The signature of the mappings in the `TLI Dynamic Table` has been omitted to save space, but it is visible in the declaration of the functions in `(*)`.

```
#include <arm_sve.h>
typedef vdouble svfloat64_t;
static vdouble vadd_vd_vd_vd(vdouble x, vdouble y) {
  return svadd_f64(svptrue_b64(), x, y);
}
```

Fig. 7. Example C intrinsics mapping in SLEEF for SVE.

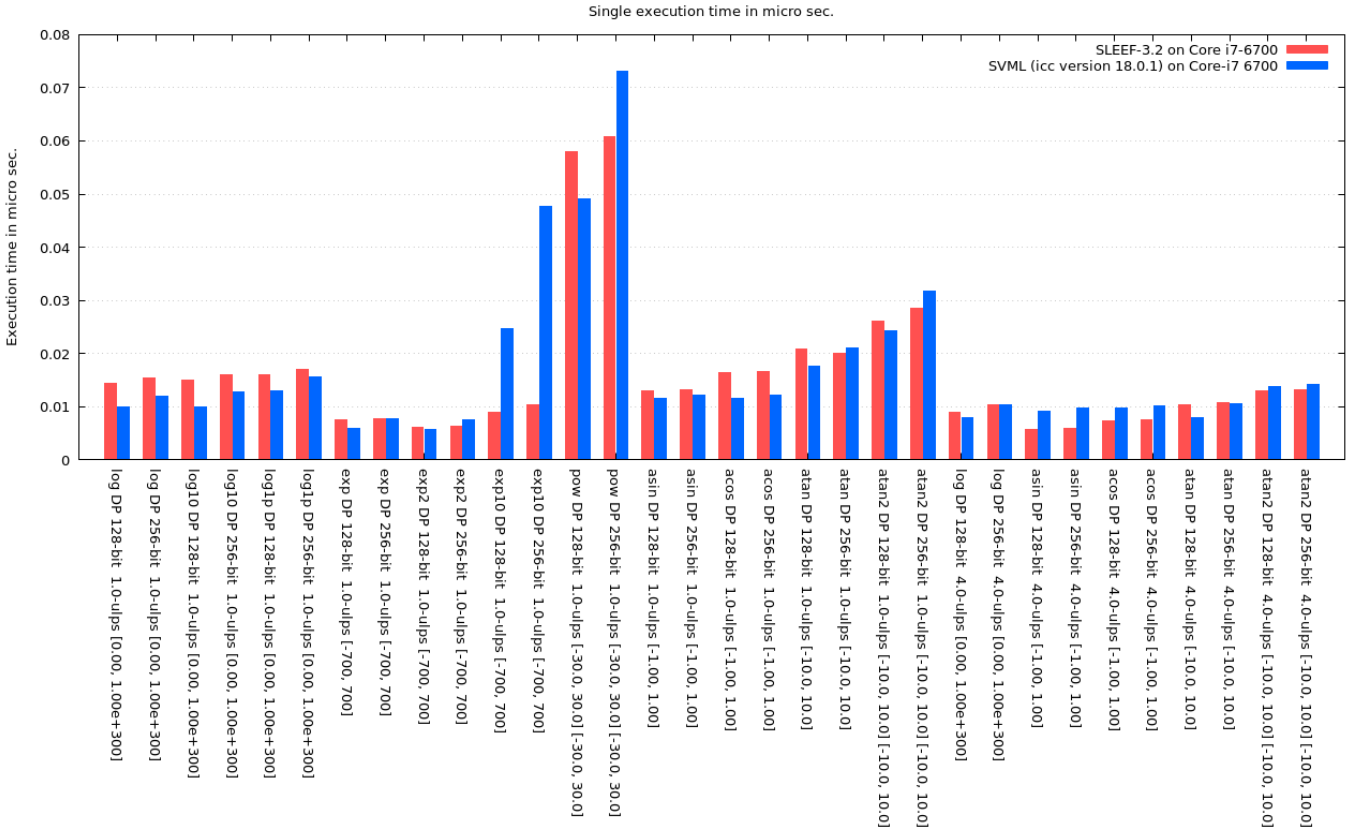


Fig. 8. Comparison of SVML versus SLEEF 3.2 on x86. Image credit: Naoki Shibata, Nara Institute of Science and Technology. Image retrieved from http://sleef.org/benchmark.xhtml, on September 9th, 2018.

However, even at these early stages, some of our customers using `-fsimdmath` have already reported a significant speedup on some HPC applications. For example, the Quantum Monte Carlo simulation code `qmcpack`[9] [4] experiences a speedup of 1.2x on NEON when using `-fsimdmath` on real-life input data (see fig 9).

V. Analysis of the Implementation

This section presents an analysis of the implementation of `-fsimdmath` in Arm Compiler for HPC.

A. Advantages

The mappings between scalar functions and vector functions are based on an open standard, the `declare simd` directive of OpenMP. Relying on standard specifications in the implementation of a compiler is beneficial as it guarantees portability. Both the compiler and the library are implementing the *Vector function ABI specification for AArch64*, with the results that compiler and library can be de-coupled from each other. In fact, any compiler compliant with the specifications can use `libsimdmath.so` as a target for auto-vectorization of math functions, and at the same time any other vector math library compliant with the *Vector function ABI specifications* can be used by *Arm Compiler for HPC*, as described in section II. This is an important feature because *GCC* plan for auto-vectorization of math routines relies on a vector version of `libm`, called `libmvec`, which provides vector functions with names and signatures based on the Vector function ABI specifications of the target architecture.

The implementation in *Arm compiler for HPC* is easily maintainable and extendible. In fact, adding or removing math

[9]https://www.qmcpack.org/

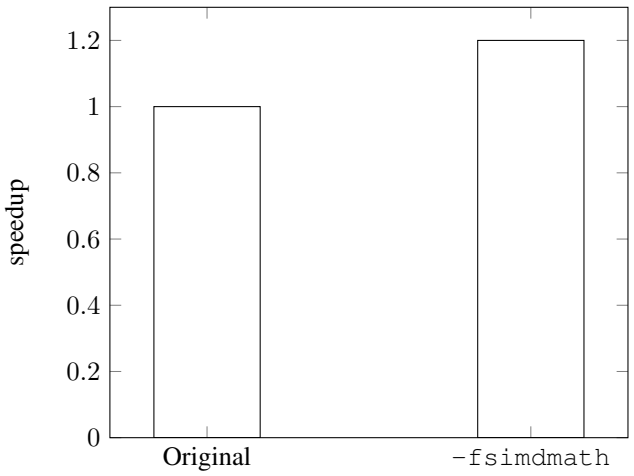


Fig. 9. Speedup on `qmcpack` when compiling with `-fsimdmath`.

functions that need to be considered for auto-vectorization by the compiler is just a matter of tweaking the header file shipped with the compiler. The lists of such functions are not expected to be the same in each library that provides vector math routines for a specific architecture, but pre-processor macros can be used to make sure the compiler knows which one to select. This seems to be a major improvement over the current auto-vectorization capabilities of upstream LLVM that are implemented via the `veclib` command-line option, which rely on maintaining a set of static arrays in the sources of the TargetLibraryInfo (TLI).

One additional advantage of relying on names for the vector functions that are generated according to a Vector function ABI is that it enables the linker to generate scalar code in situations where the vector function generated by the compiler is not available in a library that is visible at link time. As an example of this situation, suppose that the code in figure 2 is used in the development of an application that is being compiled with `-fsimdmath`. Suppose that the user has access to a machine that provides a vector math library that presents a subset of the symbols available in `libsimdmath.so`, or that the user wants to invoke the compiler with `-fsimdmath=X` to link the executable to `libX.so` instead of `libsimdmath.so`. These situations could generate a failure in the compiler, but such failure could be prevented by a *smart* linker that could demangle the name of the vector call planted by the compiler, and understand that a call to a missing symbol like `_ZGVsMxv_sin` could be replaced by a series of calls to `sin` in `libm` that loops over the values of the vector registers passed to `_ZGVsMxv_sin`.
.

One key advantage of this implementation is the choice of the vector math library shipped with the compiler. Although *Arm compiler for HPC* provides only a Linux build of `libsimdmath.so`, targeting the vector extensions of AArch64, *SLEEF* supports a rich variety of operating systems and platforms, therefore the compiler can be easily extended to support `-fsimdmath` on such systems. *SLEEF* works on *Linux*, *OSX*, *Windows*, and *FreeBSD*. It can be built using `clang`, *GCC*, *Intel C Compiler* and *Microsoft Visual Studio*, and it targets all major vector extensions available on `x86`, `ARMv7`, `ARMv8` and powerpc.

B. Limitations

The scalar-to-vector-function mechanism used by the compiler in this implementation of `-fsimdmath` via the OpenMP directive `declare simd` works very well with the functions of the math libraries, but it cannot be extended to fully support the directive for user-defined functions. in particular, there is limited support for the `linear` and `uniform` clauses of the `declare simd` directive. Those directives can still be used, as for example, the `linear` clause is needed for functions like `sincos[f]` (see section IV), but the interface with the LoopVectorizer (LV) would not be able to distinguish the *linear step* of the clause, and its modifiers, because such informations can not be encoded in the signature of the vector functions. This would require adding new fields to the dynamic table of the TLI - a solution that, although possible, it does not conform to the plan the open source community has for implementing the `declare simd` directive.[10]

The code in `libsimdmath.so` is not fully optimized to achieve the best performance on AArch64, especially for SVE. In the current implementation, the masking interface of the function is simulated with wrappers around the non-masked functions. Moreover, the algorithms used in the core of the library do not utilize the accelerators that the instruction set of SVE specifically introduces for improving the computation speed of some of the math functions.

Finally, one of the drawbacks of this implementation of `-fsimdmath` is that it requires the whole tool-chain for testing, because the TLI is populated via the pass manager and not via metadata stored in the IR module. Therefore, it is not possible to unit test the LV functionality in the middle-end of LLVM.

VI. The status of math functions in LLVM ands its interaction with `-fsimdmath`.

Auto-vectorization of math routines requires a link between scalar function names and their vectored counterpart. One of the obstacles LLVM introduces is that the vector versions are new, whereas LLVM has had years to optimize the usage of the scalar versions. Typically, LLVM does this by converting the scalar calls into something that LLVM finds easier to work with. The problem with this is that it breaks the scalar-to-vector mapping and therefore killing the auto-vectorization.

This can be seen today with calls to `pow` where, when built with `-ffast-math`, LLVM replaces it with a call to

[10]The current plan for implementing the `declare simd` directive is outlined in the RFC from Intel at http://lists.llvm.org/pipermail/cfe-dev/2016-March/047732.html. This plan covers function vectorization in the generic case of function definitions, not in the specific case of vector math function declarations. The upstream development of `-fsimdmath` should be based on the generic functionality, because it could reuse some of the components described in the RFC.

an intrinsic with a different name, `llvm.pow.f64` in this case. At a higher level it means that loops calling `pow` call vectorized functions at `-O3` but not at `-Ofast` (at this level the loop might not even vectorize).

Another example is during code generation. It is at this phase of compilation that LLVM will combine calls to `sin` and `cos` into `sincos`. Again, it does this by converting the scalar call into something easier to work with (`ISD` nodes in this case) with the two nodes later combined into a `SINCOS` node. As the vectorized versions of these routines are new, the code generator does not yet know how to combine them, which results in vectorized loops with `-fsimdmath` being slower than using a highly optimized scalar library (`libamath.so`) when the loop contains `sin`/`cos` that can be combined as described above.

We think that there are two routes to success:

1) Stop the special case conversion of math routines and instead teach LLVM how to work with the original calls efficiently.
2) Alternatively, teach the special case handling about the vectorized versions that now exist in the backend.

VII. Conclusions and future work

In this paper, we have shown that the auto-vectorization of math functions is beneficial for real HPC applications.

The functionality is based on open standards, and therefore we think that the LLVM community might be interested in adopting `-fsimdmath` as a new command-line option for the front-end.

At the time of writing, an implementation of the `declare simd` directive is based on a proposal from Intel[11] is being discussed upstream. The proposal is based on the `declare simd` directive, and it will be easy to extend it to the other architectures supported by LLVM.

In this direction, Intel is contributing to a new pass, called the *Vector Clone Pass*,[12] that generates IR metadata to describe the available vector function generated from user code decorated with the `declare simd` directive. The IR meta-data is exposed to the vectorizer pass to enable function vectorization.

Once the *Vector Clone Pass* will be part of LLVM, the work that needs to be done to support `-fsimdmath` will have to be based on it.

An implementation of `-fsimdmath` based on the *Vector Clone Pass* has the advantage to enable unit testing without the need to use a front-end, as the list of the available vector functions is stored in the IR.

The underlying mechanism of `-fsimdmath` has also shown that the `declare simd` of OpenMP has an important role in enabling users to provide their own vector libraries, as it applies to any function, not only to the math functions. For this reason, we believe it is a superior solution to the current vector library interface of `clang` which relies on the `-veclib` command-line flag. For example, many machine learning algorithms rely on computations that are not listed in any standard, such as rectifier or sigmoid functions, which as often used as activation functions. Users could in fact interface their own libraries with vector implementation of such functions, and achieve auto-vectorization of loops invoking them, by simply adding the appropriate scalar declarations in header files, without needing to hack in to the compiler to enable them via `-veclib`.

In the long term, the introduction of the directive `declare variant` in the upcoming release of *OpenMP 5.0* will make it possible to use the `declare simd` mechanism in conjunction with vector math libraries that do not follow the naming conventions that a Vector Function ABI specification mandates for a vector extension.

As a final note, it is worth mentioning that *SLEEF* can also be compiled in the form of a bit-code library when compiling it with `clang`. This has the potential to further improve the performance of the code that relies on vectorizable loops containing math function calls, as it would enable the compiler to access not only the code outside the vector function call, but also the target-independent IR instructions of the vector routine.

Appendix: Vector Function ABI name mangling scheme

The *Vector function ABI specification for AArch64* mandates that the vector function names generated by a compiler are mangled according to set of rules. The vector name is created so that it is possible to reconstruct the signature of the vector function, and trace its origin back to the scalar declaration with which it is associated.

The name mangling scheme for AArch64 is based on the *Itanium C++ ABI* mangling scheme described in [6]. In particular, the name mangling rules have been designed to be compatible with those defined for the x86 architecture in [7].

The vector names are in the form `_ZGV<isa><mask><len><parameters>_<name>`. Each token in the mangling is defined as follows:

`<isa>`
: This token specifies the vector extension used in the vector function, where `n` is for NEON and `s` is for SVE.

`<mask>`
: The value of this token is `M` for masked functions, which accept an additional mask parameter, and `N` for unmasked functions.

`<len>`
: This value represents the number of concurrent lanes on which the vector functions operate. In case of VLA vectorization for SVE, the value is set to `x`. Note that for SVE this can be set as a number when targeting vector-length specific vectorization.

`<parameters>`
: This is a list of the parameters expecteted in the signature of the vector function, according to the qualification of the scalar parameters given through

[11] http://lists.llvm.org/pipermail/cfe-dev/2016-March/047732.html

[12] https://reviews.llvm.org/D22792

the `declare simd` directive in the declaration of the scalar function. For example, `v` is for vector and `l2` is for a parameters that is associated with a `linear` modifier with a step of 2.

`<name>`
: This token is the original assembly name of the scalar function.

For example, the vector function `_ZGVsMxv_sin` associated to `sin` in figure 2 is an SVE (`s`) vector function, which accept a mask parameter (`M`), is in VLA form (`x`) and has a vector parameter as first input (`v`). From this information, it is possible to reconstruct the signature of the vector function, which can be written using the *SVE Arm C Language Extensions (SVE ACLE)* [8], and the original scalar declaration it comes from, as in figure 10.

```
// Scalar declaration.
#pragma omp declare simd
double sin(double);
// SVE vector function name
//  and signature.
svfloat64_t _ZGVsMxv_sin(svfloat64_t,
                         svbool_t);
```

Fig. 10. Scalar declaration of `sin` and associated SVE vector function.

Figure 11 shows the example of a vector function targeting NEON vectorization. In this case, the `<len>` and `masking` token of the vector function name have been set using the `simdlen`, `linear` and `notinbranch` clauses of the `declare simd` directive. The signature of the vector function is represented as a C declaration using the *Advanced SIMD (NEON) Arm C Language Extensions (ACLE)* [9].

```
// Scalar declaration.
#pragma omp declare simd simdlen(4) \
  linear(x:2) notinbanch
float foo(int x, unsigned short y);
// NEON vector function name and
// signature.
int32x4_t _ZGVnN4l2v_foo(int32x4_t,
                         int16x4_t);
```

Fig. 11. Example of vector function with a NEON signature generated with the Vector Function ABI rules.

ACKNOWLEDGMENTS

The authors would like to thank Geraint North, Will Lovett, and Ganesh Dasika for their help and valuable suggestions. Arm is grateful to Naoki Shibata for the work he has done on *SLEEF*.

The authors would like to express their gratitude to Julie Gaskin, for proofreading this article.

REFERENCES

[1] *Arm Compiler for HPC*, https://developer.arm.com/products/software-development-tools/hpc/arm-compiler-for-hpc

[2] *Vector Function ABI specification for AArch64*, https://developer.arm.com/products/software-development-tools/hpc/arm-compiler-for-hpc/vector-function-abi

[3] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, Paul Walker, *The ARM Scalable Vector Extension*, IEEE Micro (Volume: 37, Issue: 2, Mar.-Apr. 2017), https://doi.org/10.1109/MM.2017.35.

[4] Jeongnim Kim et al. *QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids*, Journal of Physics: Condensed Matter, Volume 30, Number 19, https://doi.org/10.1088/1361-648X/aab9c3.

[5] Hal Finkel, *bgclang: Creating an Alternative, Customizable, Toolchain for the Blue Gene/Q*. IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis, November 16 - 21, 2014. http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/tech_poster_pages/post119.html

[6] *Itanium C++ ABI*, Revised March 14, 2017, http://itanium-cxx-abi.github.io/cxx-abi/

[7] *Vector (SIMD) Function ABI*, Xinmin Tian, Hideki Saito, Sergey Kozhukhov, Kevin B. Smith, Robert Geva, Milind Girkar and Serguei V. Preis. Intel® Mobile Computing and Compilers, https://software.intel.com/en-us/articles/vector-simd-function-abi

[8] *Arm C Language Extensions for SVE*, https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve.

[9] *NEON Intrinsics Reference*, https://developer.arm.com/technologies/neon/intrinsics.