Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Optimizing ART: Radiative Transfer Forward Modeling Code for Solar Observations with ALMA

Marcin Krotkiewski[a*]

[a]*USIT, University of Oslo / SIGMA2, Norway*

Abstract

Various optimizations of the ART software package for the solution of the radiative transfer equation in three dimensions are discussed in this white paper. All critical path functions of the code have been optimized and vectorized using OpenMP directives. Several techniques have been used, amongst others the rearrangement of input data and internal data structures to facilitate usage of CPU vector units, vectorization of calls to the math library, explicit loop unrolling to allow vectorization of iterative loops with a convergence criterion, vectorization of data-dependent if-statements through enforced computations on all SIMD lanes and filtering of the final result. Several technical challenges had to be overcome to achieve the best performance. The OpenMPI stack needed to be compiled with a custom (non-native) Glibc library. In some cases, individual vectorized clones generated automatically by the compilers needed to be substituted with custom functions implemented manually using compiler intrinsics. Performance tests have shown that on the Broadwell architecture the optimized code works from 2.5x faster (RT solver) to 13x faster (EOS solver) on a single core. MPI implementation of the code scales with 95% efficiency on 2048 cores. Throughout the project, several GCC bugs related to automatic OpenMP vectorization have been reported, which shows that the support for the relatively new OpenMP vectorization features is still not mature. For some of those bugs effective workarounds have been developed. We also point to some shortcomings in the OpenMP `simd` vectorization framework and develop several new optimization techniques, which improve effectiveness of automatic code vectorization. Finally, some generally useful tutorials have been delivered.

* Corresponding author. *E-mail address*: *marcin.krotkiewski@usit.uio.no*

1. Introduction

The *Atacama Large Millimeter/sub-millimeter Array (ALMA)* is currently the world's largest astronomical observatory. The interferometric array is located on the Chajnantor plateau in the Chilean Andes and consists of 66 antennas. Since December 2016 ALMA has amongst others been used to observe the Sun at an unprecedented spatial resolution. ALMA's novel diagnostic capabilities need to be further developed and understood to fully exploit the instrument's potential. In this context detailed numerical simulations of the solar atmosphere and artificial "observations" of the Sun play a key role. Such numerical observations are produced as part of the SolarALMA project at the University of Oslo. An important step in this endeavor is the development of realistic numerical models of the solar atmosphere. Within SolarALMA a new code has been developed (*ART*), which solves the radiative transfer equation for a 3D numerical model, and thus reveals how the modeled part of the Sun would look like through the eyes of ALMA. The code is written in C++ with some features from the C++11 standard. The code consists of two major computational parts:

- An *equation of state (EOS)* solver, which - based on various types of input data - computes electron density, gas pressure, and density. These are quantities required by the second part of the code.
- A nonlinear solver for *radiative transfer (RT)*. This code is based on a FORTRAN code from 1970 by Robert Kurucz.

These computational kernels have modest memory requirements and are purely compute bound. Most of the time is spent inside calls to the math library functions (`exp,pow,log`). Input data is read from HDF5 files and composed into a set of 3D Cartesian grids. Each column in the atmosphere model requires independent calculations. In fact, the two kernels described above are executed independently for each grid point, with no communication required by the neighbor cells. Afterwards the resulting columns are collapsed into a single 2D image, which comprises the result of the program. In summary, the code is very suitable for efficient parallel execution:

- It is computationally heavy per grid point
- Relative to the computations it is very light on communication and IO
- The computations are trivially parallel, which facilitates both fine- and coarse-grain parallelization

The original code is parallelized using MPI. The total number of columns is divided by the number of processes, and the MPI ranks process one column at a time. The amount of work per column can vary depending on the input data.
The goal of this *Preparatory Access* project was to optimize the code's performance looking at both the per-core efficiency, and the parallel execution, to make it suitable for use in high-resolution massively parallel production runs. To achieve this target we have reorganized the code and the data structures to facilitate usage of CPU's vector units. The code has been vectorized using the OpenMP pragmas (`omp simd`), including the calls to the math library. For several reasons discussed in this report vectorization of the code using the chosen approach has proven to be challenging. Some general issues include the size (number of functions) of the code, both the flow complexity, and the computational complexity, compiler limitations. Finally, the rigid requirements imposed on both the code and the data structures by the OpenMP standard are not always easily met in practice, especially when adapting legacy code.
This report presents an analysis of the original code (*Chapter 2 Code Analysis*), discusses general vectorization methods (*Chapter 3 Vectorization Methods*), and shows how those have been applied to ART (*Chapter 4 Vectorizing ART*). Performance of the optimized code is discussed in *Chapter 5 Performance*.

2. Code Analysis

2.1. Data Structures

Input data is read from an HDF5 file, which contains 4-dimensional data sets (3D grids changing in time). The original input is stored in `(t,Y,X,Z)` order, e.g.

```
$ h5ls bifrost_dens.h5
dx                       Dataset {63}
dy                       Dataset {63}
temperature              Dataset {1, 63, 63, 496}
```

That is, the Z-coordinate is the fastest changing index. Parallel computations are assigned to processors by column, each rank reading a single column from the input file at a time. The result of the computations is one `nx*ny` picture per analyzed wavelength, where one pixel is obtained for one column of the 3D grid. The result is saved into an HDF5 file.

The original data and work distributions have some disadvantages. On any modern parallel file system, it is best for performance reasons to access large chunks of data. This maximizes the IO bandwidth (minimizes the latency) and limits the pressure on the IO servers. We have modified the code so that each MPI rank reads a block of grid columns at once. To improve IO performance further we have modified the original HDF5 input to support dataset chunking. This change is transparent to the user and makes sure that blocks of grid columns, accessed by each MPI rank, correspond to the data storage unit. As an added benefit, chunked datasets can be transparently (un)compressed by the HDF5 library, which further improves the IO performance on a parallel file system.

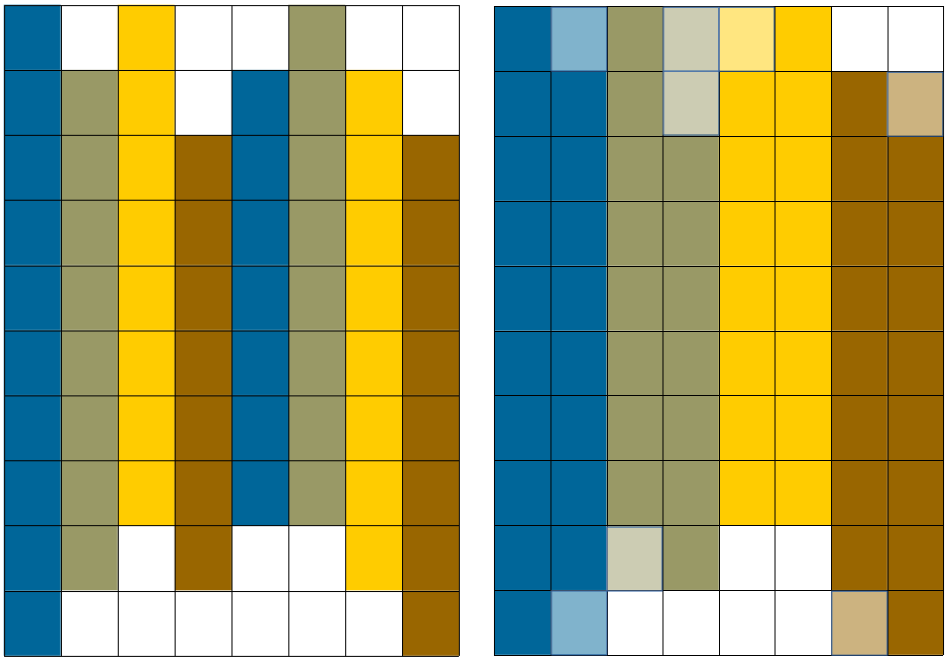

Figure 1 A schematic depth (XZ plane) cut through a 3D model. Colored squares indicate grid cells that require computations, white squares indicate grid cells without computations. *Left:* original work distribution. Individual columns are assigned to distinct MPI ranks. *Right:* computations on blocked input data with block size of 2.

The original computations are performed per column, i.e., the order in which the grid points are traversed can be expressed by the following loop structure:

```
for(j=0; j<ny; j++){
    for(i=0; i<nx; i++){
        // assign column to an MPI rank = f(i, j)
        for(k=0; k<nz; k++){
            result(j, i, k) = compute(j, i, k);
        }
    }
}
```

Left picture in Figure 1 shows an example work distribution on a simplified 2D grid, among 4 MPI ranks. The amount of work in each column can vary depending on the input data. For example, depending on the temperature threshold some grid points can be ignored, because their contribution to the result is negligible. This results in empty compute `bubbles` (white squares in the picture). Since computations within the columns are independent, the code can be vectorized in the K-direction, so that multiple points in the column are processed at the same time. Using a FORTRAN-like vector notation:

```
for(j=0; j<ny; j++){
    for(i=0; i<nx; i++){
        // assign column to an MPI rank = f(i, j)
        for(k=0; k<nz; k+=VEC_WIDTH){
            result(j, i, k:k+VEC_WIDTH-1) =
                      compute(j, i, k:k+VEC_WIDTH-1);
        }
    }
}
```

However, the compute 'bubbles' disturb the vectorization. Before each compute region there must be a scalar warm-up loop to assure correct data alignment of the vectorized code (the `active` cells start at arbitrary locations). In addition, since the length of the compute regions is not necessarily a multiple of the vector width, a tail scalar loop is required. These overheads significantly affect the efficiency of vectorization.
The above has been addressed by changing the loop order so that X is the fastest changing dimension. The right picture in Figure 1 shows a block vectorized approach with vector width of 2. Here computations must be performed for those compute `bubbles`, which are part of a vector (light-colored cells). Those contributions can either be filtered out when storing the result in the memory or ignored - since their impact on the result is negligible. In addition, this approach facilitates jamming of the two inner loops into one:

```
for(j=0; j<ny; j++){
    // beg = find Z index of the first active cell in the block
    // end = find Z index of the last active cell in the block
    // blk = size of the block in the X dimension
    for(i=blk*beg; i<blk*end; i++){
        result(i:i+VEC_WIDTH-1) = compute(i:i+VEC_WIDTH-1);
    }
}
```

The above vectorized loop is much longer than what would be possible to achieve with the original data layout, which makes it more efficient. In the optimized code, all compute intensive parts of the optimized code are oriented around such 2D slabs of data.

2.2. Vtune Analysis of Original Code

A Vtune performance summary for a single core run of the original code (Figure 2) indicates that it is purely compute bound. Half of the time is spent in the math library functions `exp`, `pow`. Only 13% of the time is reported to be spent on accessing data in the cache, and the global memory access does not contribute to the execution time. Also, 90% of the executed floating-point instructions are scalar. All these factors indicate that vectorization is a very promising direction to increase the single-core performance of the code.

Figure 3 shows the Vtune hotspot analysis, which confirms that the math library functions are the most time-consuming part of the code. In addition, some of the longest user functions are named, many of them also being internally bound by calls to `libm`. `pe_pg` is part of the EOS solver, while `COULX, HE1OP, HE2OP, HOP` come from the RT part of the code.

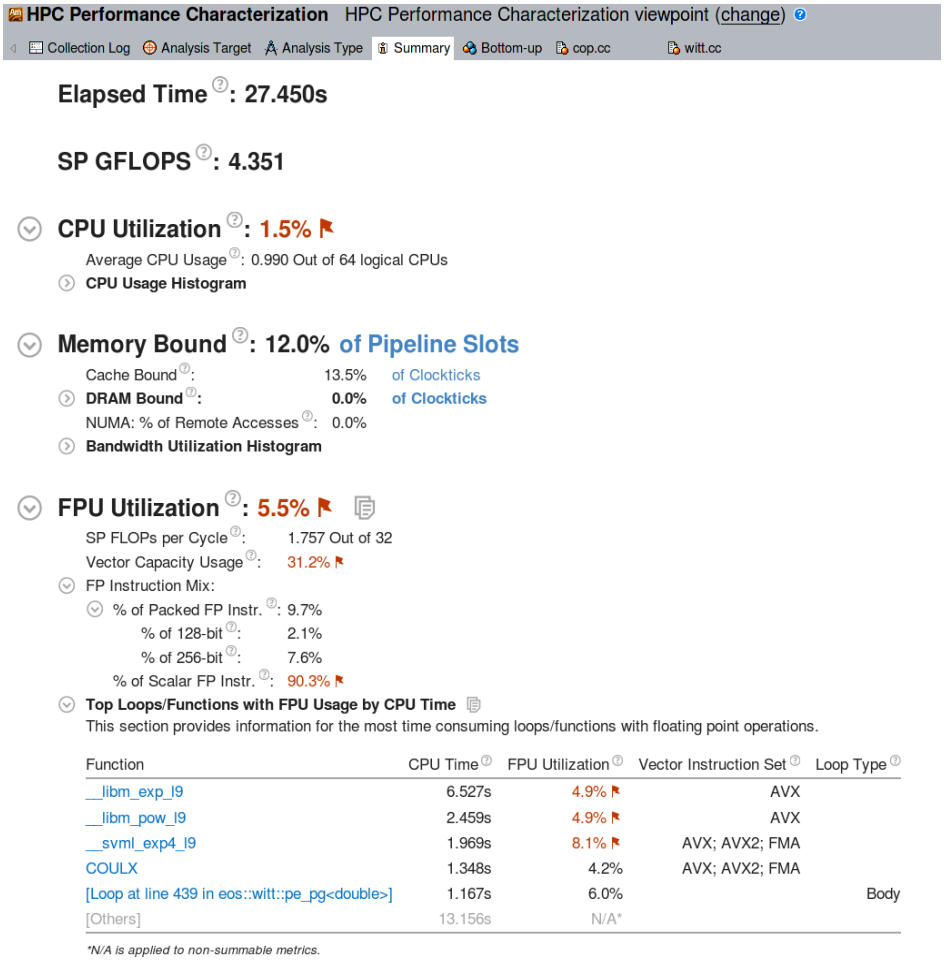

Figure 2 Vtune performance summary of the original code. Single core execution.

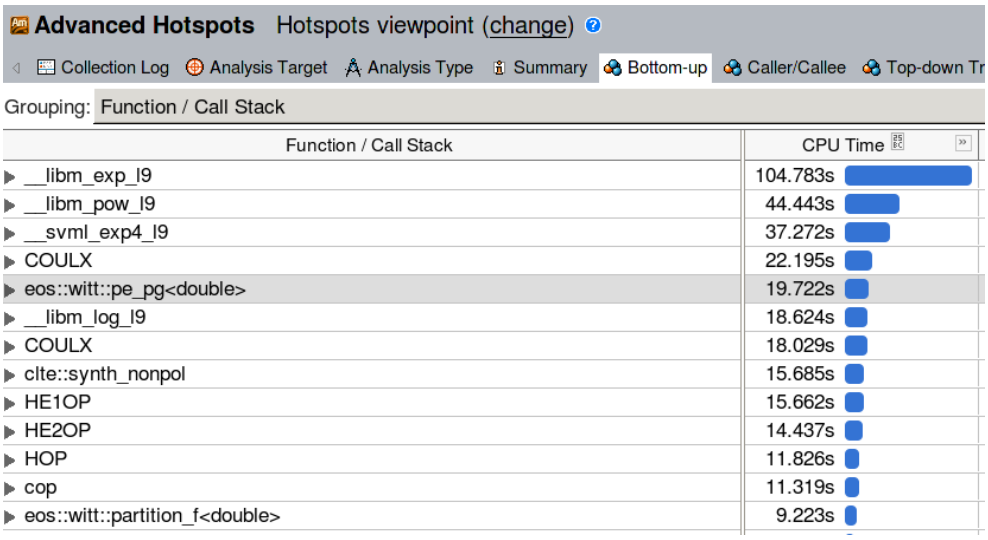

Figure 3 Vtune hotspot analysis of the original code. Single core execution.

3. Vectorization Methods

3.1. Compiler Intrinsics

Compiler intrinsics are widely used in vectorization of numerical codes. Intrinsics are called in the code like normal functions but are substituted with assembly calls internally by the compilers. Hence, they often directly correspond to individual assembly instructions and the programmer must have an in-depth knowledge of the architecture and the instruction set. Although they are easier to program than direct assembly, the learning and maintenance costs are large.

The computational kernels in ART comprise of roughly 60 computationally complicated functions. Re-implementation of the entire code from scratch using compiler intrinsics would be error prone and time consuming beyond the scope of this project. Instead, a solution is needed, which would allow a simple re-compilation of the code (with some minimal required changes) for automatic vectorization. While compilers are able to automatically perform this tasks for tight computational loops, calling functions that accept and compute on short vector arguments requires some explicit language support.

In practice, the intrinsics can be used to fine-tune small code fragments, which the compiler for some reason could not handle efficiently. An example applied in this project is described in Appendix C.

3.2. OpenMP `simd`

Starting from version 4.0 the OpenMP standard provides pragma constructs to request vectorization of loops (in C - `#pragma omp simd`), and to declare vectorized functions (`#pragma omp declare simd`). In the following C example `vfun` is a vectorized function:

```
#pragma omp declare simd
double vfun(double v1, double v2)
{
  return exp(v1-v2)/2;
}
```

Note that the function is declared using basic C types (double) and no additional change towards the scalar function call is needed. A supporting compiler automatically generates vectorized function variants (clones), which instead of a double take vectors of doubles (e.g., `m256d`) as arguments. The preferred vector width can be specified, but in general is characteristic of the given architecture. Consider another compilation unit, where `vfun` is called in an OpenMP simd-declared loop:

```
double test(int n, double *v1, double *v2, double *out)
{
  #pragma omp simd
  for(int i=0; i<n; i++){
    out[i] = vfun(v1[i], v2[i]);
  }
}
```

The intention is to process the elements of `[v1,v2,out]` arrays as short vectors of `double`s, and a supporting compiler should call a vector-enabled version of `vfun`. Function calls inside vector functions (in this case `exp` from `libm`) are vectorized as well, unless the compiler cannot find a suitable variant. Of course, vectorization can be inhibited by data / flow dependencies between loop iterations.

In practice care must be taken to assure that the compiler works as expected. In some cases, both ICC and GCC may fail to vectorize or do it in a non-optimal way. Throughout this project we have encountered many cases, in which vectorization of function bodies (or parts of it) is inhibited. Vectorization reports are not always enough to be certain about the quality of the optimizations, or to diagnose problems. Inspection of the generated assembly is sometimes advisable (see Appendix A.). The following sections show some of the pitfalls and explain how to correctly compile the above basic OpenMP codes using ICC and GCC.

3.2.1. Compiling with GCC

When the above `test` function is compiled with the following arguments, GCC reports the `omp simd` loop as vectorized:

```
$ gcc -march=broadwell -fopenmp -O3 -c test.c -fopt-info-vec
test.c:16:28: note: loop vectorized
```

In fact, this message is confusing and although GCC does generate a call to a vectorized `vfun` clone, the clone itself executes in scalar mode. GCC reports the loop as vectorized because 1) it does generate vectorized clones of `vfun` and 2) it does call one of the vectorized clones inside the loop in question (see Appendix A. on how to establish this). Internally, however, the clones compute on scalars, i.e., the code loops over the vector arguments and executes a scalar `exp`. The reason is that `vfun` calls `exp` from the math library, and in order to enable vectorized `libm` in GCC one must use the `-ffast-math` compiler flag (see *Section 3.3 Vectorized Math Library*).

GCC by default generates SIMD clones for all types of architectures, from SSE to AVX2. However, when compiling for the Skylake architecture GCC 8 by default uses 4-wide `double` vectors, while Skylake supports AVX512 and 8-wide vectors. To force the compiler to use AVX512 one must use the `-mprefer-vector-width=512` compiler switch. This is a change in behavior compared to version 7. This is due to increased power demands (and hence heat production), which may force the CPU to dynamically scale down the frequency [1]. As a result, it might be more efficient to run AVX2 code instead of AVX512 code on Skylake, especially when using all cores.

3.2.2. Compiling with ICC

A similar command line used with the ICC compiler also produces an unexpected result:

```
$ icc -march=broadwell -fopenmp -O3 -qopt-report-phase=vec
  -qopt-report=5 test.c
  icc: remark #10397: optimization reports are generated
  in *.optrpt files in the output location
```

Broadwell supports AVX2 instructions with vectors of 4 `double`s. However, the ICC vectorization report says the following:

```
LOOP BEGIN at test.c(15,3)
[...]
remark #15305: vectorization support: vector length 4
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
[...]
remark #15489: --- begin vector function matching report ---
remark #15490: Function call: vfun(double, double) with simdlen=4,
  actual parameter types: (vector,vector)   [ vtest.c(16,14) ]

remark #15492: A suitable vector variant was found (out of 2) with
  xmm,simdlen=2, unmasked, formal parameter types: (vector,vector)

remark #15350: The function ISA does not match the compilation target.
  For better SIMD performance, consider using -vecabi=cmdtarget
  compiler switch or "processor" clause in vector function declaration
LOOP END
```

The code has been vectorized, and at first glance the vector width is 4. However, vector width of 2 is used in the call to `vfun`. Even though the target architecture is Broadwell, ICC generated an SSE compatible vectorized function. This is unexpected, because the same compilation options used on a tight computational loop show the expected result, i.e., 4-wide vector instructions are used. As suggested by the ICC report we should add the `-vecabi=cmdtarget` compiler switch or add an ICC-specific `processor` clause to the `omp declare simd processor=core_5th_gen_avx`. As explained in the documentation of the `processor` clause [2]: *"The vector version of the routine that is created by the compiler is not affected by processor options specified on the command line."* In other words, `-march=broadwell` has no consequence for vectorization of `omp simd` functions.

Similarly, starting from version 18 ICC by default uses 4-wide `double` vectors when compiling for the Skylake architecture. To force the compiler to use AVX512 one must use the `-qopt-zmm-usage=high` compiler switch.

3.2.3. Declaration of vectorized functions

Vectorized C functions are declared using `omp declare simd` pragma, e.g.:

```
#pragma omp declare simd
double vfun(double v1, double v2);
```

Optional clauses are possible [3], some of them may be compiler specific, e.g., the `processor` clause used by ICC.

A large part of ART is written in C++. Vectorized member functions are declared by additionally including `this` inside the `uniform` clause:

```
class vclass {
  #pragma omp declare simd uniform(this)
  double vmember(double v1, double v2);
}
```

The above essentially means that `this` is the same for all calls to `vmember` [3].
To simplify the declaration of vectorized functions and members, and to unify the declarations for different compilers, suitable macros have been implemented:

```
#ifdef __ICC
#define OMP_ISA processor(core_5th_gen_avx)
#else
#define OMP_ISA
#endif

#define DO_PRAGMA(name) _Pragma(#name)
#define VECTORIZED_FUNCTION(mods) \
        DO_PRAGMA(omp declare simd notinbranch OMP_ISA mods)
#define VECTORIZED_MEMBER(mods) \
        VECTORIZED_FUNCTION(uniform(this) mods)
```

In the optimized code the declaration of vectorized C functions is done as follows:

```
VECTORIZED_FUNCTION() double vfun(double v1, double v2);
```

For vectorized member functions:

```
class vclass {
  VECTORIZED_MEMBER() double vmember(double v1, double v2);
}
```

For both macros additional clauses can be specified as the `mods` argument.

3.3. Vectorized Math Library

GCC and ICC can both use a vectorized math library with platform-optimized implementations of amongst others `exp,pow,log`, which are the relevant functions for this project. Intel compiler uses its own *Short Vector Math Library (SVML)*, which is also available as intrinsics. The library comes together with the compiler installation, and there is nothing system-specific that needs to be done to use it.
GCC on the other hand relies on `libmvec`, which is part of Glibc version 2.22 and higher. This means that on systems with an older version of Glibc the vectorized math library is not readily available, regardless of the GCC version. This is a significant practical problem, because OS vendors often lag a few years when it comes to Glibc (e.g., Centos 7.5 comes with version 2.17 released in the end of 2012).

While it is not safe to manually upgrade Glibc for a given OS release without recompilation of the entire software stack, it is possible to install a custom Glibc as a module and recompile a limited number of software packages against that Glibc version. For this project it is required to re-build the MPI library (OpenMPI is the preferred option for Mellanox-based systems). It is not as straight forward as compiling OpenMPI with a native Glibc - for details see Appendix B. . The implemented method is transparent to the user, so nothing needs to be changed in the `Makefile` - `mpicc` can be used in the usual way.
As noted in the documentation of SVML [4] *"[...] the intrinsics differ from the scalar functions in accuracy"*. Scalar implementations are not the same as the vectorized ones with vector width of 1. Instead they follow strict floating-point arithmetic and are more computationally demanding. GCC is by default conservative wrt. the floating-point optimizations: vectorized math library is only enabled with `-ffast-math`. ICC by default uses relaxed settings (`-fp-model fast=1`), which allow the compiler to make calls to low accuracy SVML functions. In addition, SVML also provides higher accuracy vectorized functions (`-fp-model precise`). Vectorization of `libm` calls can be prohibited with `-fp-model strict`. With ART the high accuracy is not required, hence we compile the code with the most relaxed settings.

4. Vectorizing ART

4.1. EOS solver

The EOS solver implemented in file `witt.cc` contains more than 20 functions that need to be vectorized. Because of the code structure and language features used, several different techniques have been developed to achieve this goal:

- Automatic code translation from monolithic `f2c` output to short, vectorizable inline functions (Section 4.1.1)
- Elimination of conditional code for constant arguments through function versioning and inlining (Section 4.1.2)
- Elimination of multiple reference and pointer output arguments Through inlining (Section 4.1.3)
- Explicit loop unrolling to allow vectorization of iterative loops with a convergence criterion (Section 4.1.4)

4.1.1. Generation of vectorizable code from `f2c` output

The partition function `eos::witt::partition_f` defined in `witt.cc` was automatically generated from a FORTRAN code using `f2c`. Around 1300 lines of critical path code consists of 92 `if` and `goto` statements, which choose the appropriate computations based on an integer input argument `nel_in`:

```
template <class T> void eos::witt::partition_f(int nel_in, T t_in,
T &u1_in, T &u2_in, T &u3_in, T &du1_in, T &du2_in, T &du3_in)
```

The original function was not vectorizable, but it also prevented vectorization of the parent caller functions. The optimized partition function has been implemented using the following steps:

- A Perl script (`utils/convert_partition_f`) was used to convert the monolithic partition function generated by `f2c` into 92 separate `inline` functions. The integer argument was eliminated and instead incorporated into the function name, e.g., `partition_f_0(...)`. The resulting individual functions are located in `src/part.h`.
- The original code called `partition_f` in a loop with constant bounds from 1 to 28, and passed the iterator as the integer argument:

```
for(int i=1; i<28; i++){
     partition_f(i, ...);
     // process the results of partition_f
     [...]
}
```

The optimized code unrolls the loop and calls `partition_f` explicitly 28 times inside the `SPEC_CONTRIBUTION()` macro, which includes processing of the results:

```
#define SPEC_CONTRIBUTION(sp)                \
  partition_f_##sp(temp, x, y, u0, u1, u2); \
  // process the results                     \
  [...]

  SPEC_CONTRIBUTION(1);
  [...]
  SPEC_CONTRIBUTION(27);
```

Calls to `SPEC_CONTRIBUTION()` have been inlined directly into methods, which use the partition function (`pe_pg, gasc`). The resulting code structure allows for inlining and automatic vectorization of the partition function. Note however that the loop has been unrolled for a fixed number of species (28). For a different number the code must be modified to include new/remove unnecessary calls to `SPEC_CONTRIBUTION()` in both caller functions.

4.1.2. Optional pointer arguments

A number of ART functions are implemented in a general way so that - depending on the input data - they might provide additional output to the caller, e.g.:

```
template <class T> T witt::pe_pg(T temp, T Pe, T Pgas, T *fe_out = NULL);
```

Depending on whether the user provides `fe_out` or not, the function behaves differently:

```
[...]
if(fe_out){
  fe_out[0] = value;
}
```

Such constructs prevent efficient vectorization. With `omp declare simd` one of `[uniform, linear]` clauses is applied to the arguments. In this case `fe_out` is a linear array:

```
VECTORIZED_MEMBER(linear(fe_out))
template <class T> T witt::pe_pg(T temp, T Pe, T Pgas,
                                 T *fe_out = NULL);
```

The `uniform` clause declares the arguments to have a constant value for all concurrent invocations of the function in the execution of a single SIMD loop (e.g., `this`). The `linear` clause declares arguments to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of the enclosing loop [3]. In the above example `fe_out` is declared linear, because it is an array, to which results of subsequent calls to `pe_pg` are written, one per call. The problem is that if `fe_out=NULL`, then it is in fact `uniform`. In this case the compiler will not find a correct vectorized function and call a scalar clone instead.

To address this problem and keep the generality without changing the API such functions have been split into three separate functions:

```
VECTORIZED_MEMBER(linear(fe_out))
template <class T> inline T witt::pe_pg_base(T temp, T Pe, T Pgas,
                                             T *fe_out){
   // ...
   if(fe_out){
     fe_out[0] = value;
   }
   // ...
}

VECTORIZED_MEMBER(linear(fe_out))
template <class T> T witt::pe_pg(T temp, T Pe, T Pgas, T *fe_out){
#pragma forceinline
   return pe_pg_base(temp, Pe, Pgas, fe_out); // always inlined
}

VECTORIZED_MEMBER()
template <class T> T witt::pe_pg(T temp, T Pe, T Pgas){
#pragma forceinline
   return pe_pg_base(temp, Pe, Pgas, NULL); // always inlined
}
```

Two dedicated functions are created for both cases of `fe_out=NULL` and `fe_out` being a non-null pointer. Their only purpose is to inline a call to the base function. Depending on the value of `fe_out`, during the inlining process the conditional code will be eliminated or emitted in a way that does not prevent vectorization.
Note that for this method to work inlining of the base function has to be enforced. This is done differently for different compilers. For ICC we have to use `#pragma forceinline`, while GCC needs the function to be declared with `attribute((always_inline))`.

4.1.3. Reference and pointer return arguments

A common case in ART are functions, which return values through arguments passed as pointers or references, e.g.:

```
template <class T> void witt::molecb(T X, T *Y, T *dy){
   Y[0]=[...];  Y[1]=[...];
  dy[0]=[...]; dy[1]=[...];
}
```

In the above case `Y` and `dy` are neither `uniform`, nor `linear`. They are temporary arrays of size 2 that need to be defined for each SIMD lane separately. While it's possible to write a vectorized intrinsics-based code similar to the above, no construct exists in OpenMP that would allow vectorization of the above function. To do this one needs to first split the arrays into separate arguments:

```
VECTORIZED_MEMBER()
template <class T> inline void witt::molecb(T X,
                   T &Y_0, T& Y_1, T &dy_0, T &dy_1)
{
   Y_0=[...];   Y_1=[...];
  dy_0=[...];  dy_1=[...];
}
```

The above function still cannot be vectorized, because OpenMP currently does not specify a way to return multiple values through reference arguments [5]. However, `molecb` can now be inlined into the caller, which eliminates the need to return multiple values and allows the function call to be vectorized:

```
VECTORIZED_MEMBER()
template <class T> inline T witt::pe_pg_base(T temp, T Pe, T Pgas,
                                              T* fe_out)
{
    T theta, cmol_0, cmol_1, dcmol_0, dcmol_1;
    [...]
#pragma forceinline
    molecb<T>(theta, cmol_0, cmol_1, dcmol_0, dcmol_1);
    [...]
}
```

In practice, in the current (4.5) specification of OpenMP any output argument, whether pointer or reference, must be declared `linear` for the function to be vectorizable [5]. And this means that such arguments must have size on the order of the iteration space of the enclosing loop, i.e., they cannot be pointers to temporary variables. This is an OpenMP limitation, which incurs unnecessary memory overhead and implementation complexity. Hence such constructs are best avoided, or such functions must be re-written and forcefully inlined.

4.1.4. Convergence criterion in loops

An interesting case for automatic vectorization are loops that iterate until a numerical convergence criterion is met, e.g.:

```
template <class T> T eos::witt::pe_from_pg(T temp, T Pgas,
                                            T *fe_out)
{
  float dif = 1.1;
  T Pe = 1.2 * init_pe_from_T_Pg<T>(temp, Pgas), oPe = Pe;
  int it = 0;

  while((dif > prec) && (it++ < 250)){
    if(dif > 0.5) Pe = (oPe + Pe)*0.5;
    oPe = Pe;
    Pe = pe_pg<T>(temp, Pe, Pgas, fe_out);
    dif = 2.0 * fabs(float(Pe - oPe)/(Pe + oPe));
  }

  return Pe;
}
```

The number of iterations in the above loop depends on the input data, and is in general different for each call to `pe_from_pg`. This limits the possibilities of automatic vectorization: to conserve the semantics of the program the compiler must check convergence of each vector element individually, and only iterate for those vector elements, which have not converged yet. In ICC the overhead of such masked computations is large and eliminates most of the vectorization speedup. On the other hand, due to a bug, GCC cannot vectorize loops, which are part of vectorized functions and make function calls [6], and always executes a scalar clone.

Optimization opportunities in this type of numerical codes come from the fact that performing more iterations than necessary is not going to deliver wrong results. Hence, we can unroll a hard-coded number of loop iterations without checking for convergence, and fine-tune the yet unconverged vector elements in a scalar manner. In the worst case the vectorized algorithm performs more iterations than the sequential one. To implement this idea, we rename `pe_from_pg` to `pe_from_pg_iter`, and re-write `pe_from_pg` as follows:

```
VECTORIZED_MEMBER(linear(fe_out))
template <class T> T eos::witt::pe_from_pg(T temp, T Pgas,
                                           T *fe_out)
{
  T Pe = 1.2 * init_pe_from_T_Pg<T>(temp, Pgas), oPe = 0, dif;

  // static computations with no convergence check
#pragma noinline
  oPe = Pe; Pe = 0.5*(oPe + pe_pg<T>(temp, oPe, Pgas));
  [...] // repeat the above line a few times

  // refine with masked vectorized calls (ICC),
  // or scalar calls (GCC)
#pragma noinline
  return pe_from_pg_iter<T>(temp, Pgas, Pe, oPe, fe_out);
}
```

In the above implementation everything except for `pe_from_pg_iter` is efficiently vectorized. Since the initial iterations cannot be done in a loop (GCC bug [6]), the number of repetitions must be defined at compile-time and should be chosen experimentally such that it gives best results for real-world data. The current implementation has been tuned for the test data sets and gives very good results.

4.2. RT solver

The *Radiative Transfer* solver is a C++ adaptation of a well-known FORTRAN 66 code *Atlas* by Robert Kurucz [7]. The algorithms and their implementation performed well in 1970s but require adjustments to work efficiently on modern architectures. Some difficulties are the numerous `if`-statements, double to integer conversions, and non-contiguous (*gather*) table lookups. The following optimization techniques have been applied to various RT solver functions:

- Elimination of reference arguments (Section 4.2.1).
- Elimination of static `if`-statements through explicit loop unrolling and function inlining (Section 4.2.2).
- Vectorization of data-dependent `if`-statements through enforced computations on all SIMD lanes and filtering of the result (Section 4.2.3).
- Improving vectorization through intrinsics-based fine-tuning of certain individual vectorized functions (Section 4.2.4).

4.2.1. Elimination of reference arguments

Originally most RT functions returned `void`, and the results were returned through a reference variable instead, e.g.:

```
void HEMIOP(double &AHEMIN, double XHE1, double FREQ,
                            double T, double XNE)
{
  double A =  3.397E-26+(-5.216E-11+7.039E05/FREQ)/FREQ;
  double B = -4.116E-22+( 1.067E-06+8.135E09/FREQ)/FREQ;
  double C =  5.081E-17+(-8.724E-03-5.659E12/FREQ)/FREQ;
  AHEMIN = (A*T+B+C/T)*XNE*XHE1*1.E-20;
  return;
}
```

As discussed in *Section 4.1.3 Reference and pointer return arguments*, automatic vectorization of the above function is prevented by the reference argument. In the optimized implementation the functions are declared as vectorized, `inline`, and return a value in a standard way:

```
VECTORIZED_FUNCTION()
inline double HEMIOP(double XHE1, double FREQ,
                     double T, double XNE)
{
  double A =  3.397E-26+(-5.216E-11+7.039E05/FREQ)/FREQ;
  double B = -4.116E-22+( 1.067E-06+8.135E09/FREQ)/FREQ;
  double C =  5.081E-17+(-8.724E-03-5.659E12/FREQ)/FREQ;
  return (A*T+B+C/T)*XNE*XHE1*1.E-20;
}
```

4.2.2. Elimination of static `if`-statements

Static `if`-statements can be resolved by the compiler at compile time. One example is the `COULX` function:

```
double COULX(int N, double freq, double Z)
{
  static const double A[6]={...},
                      B[6]={...},
                      C[6]={...};
  double CLX, FREQ1;
  int n;

  n=(N+1)*(N+1);
  if(freq>=Z*Z*3.28805e15/n) {
     FREQ1=freq*1.e-10;
     CLX=0.2815/FREQ1/FREQ1/FREQ1/n/n/(N+1)*Z*Z*Z*Z;
     if (N>=6) return CLX;
     CLX*=(A[N]+(B[N]+C[N]*(Z*Z/FREQ1))*(Z*Z/FREQ1));
     return CLX;
  }
  return 0.;
}
```

The emphasized `if`-statement prevents automatic vectorization of this function, because different code path could be chosen for each vector element. Note however that `COULX` is called in a loop with static bounds:

```
void HOP(...)
{
  [...]
  for(int N = 0; N<8; N++) CONT[N] = COULX(N,FREQ,1.0);
  [...]
}
```

The loop performs 8 iterations calling `COULX` with the iterator as argument. Due to the emphasized `if`-statement `COULX` cannot be vectorized, but even if it could - as discussed in *Section 4.1.4 Convergence criterion in loops* -

a bug prevents GCC from vectorizing this loop directly due to the function call. Both problems are overcome by manually unrolling the constant loop and inlining the call to `COULX`:

```
VECTORIZED_FUNCTION()
inline double COULX(int N, double freq, double Z);

VECTORIZED_FUNCTION()
void HOP(...)
{
  [...]
  CONT[0] = COULX(0,FREQ,1.0);
  CONT[1] = COULX(1,FREQ,1.0);
  [...]
  CONT[6] = COULX(6,FREQ,1.0);
  CONT[7] = COULX(7,FREQ,1.0);
  [...]
}
```

The above results in a very efficient vectorized code for both tested compilers.

4.2.3. Vectorization of data-dependent `if`-statements

In contrast to static `if`-statements, the control flow of data-dependent `if`-statements is dynamic and depends on the input data. Vectorization of such constructs is often possible but can be less effective. Consider the `HMINOP` function:

```
VECTORIZED_FUNCTION()
inline double HMINOP(double FREQ, ...)
{
  [...]
  if(FREQ <= 1.8259E14)     HMINBF = 0.;
  else if(FREQ >= 2.111E14) HMINBF = ...; // formula 1
  else                      HMINBF = ...; // formula 2
  [...]
}
```

Depending on the value of `FREQ` a different code path may be chosen for each vector element. Compilers may handle this in one of the following ways:

- Extract scalar values from vector arguments, perform scalar computations for each scalar, and assemble the individual results into a vector. In this case the “vectorized” clones perform more work than the scalar implementation and is likely slower.
- Compute all required code paths for all vector elements in a vectorized way and based on a mask re-assemble the result. This method can only be used if - except for computing the result - the computations do not have any side effects, e.g., they do not change any object properties, or variables.

Depending on the vector length, the number of `if`-statements (distinct execution paths) and the input data one approach may perform better than the other. Unfortunately, compilers do not always make the best choice, and there are cases in ART, which proved to be problematic. Consider the `LUKEOP` function:

```
VECTORIZED_FUNCTION()
double LUKEOP(...)
{
  double ALUKE = ( N1OP(FREQ,TKEV          )*XN1 +
                   O1OP(FREQ               )*XO1 +
                   Mg2OP(FREQ,TKEV         )*XMg2+
                   Si2OP(FREQ,FREQLG,T,TLOG)*XSi2+
                   Ca2OP(FREQ,TKEV         )*XCa2)*STIM;
  return ALUKE;
}
```

Several other functions are called to compute the result. All of them are vectorized correctly, and so `LUKEOP` is also vectorized. However, it is itself called from inside an `if`-statement:

```
VECTORIZED_FUNCTION()
void cop(...)
{
  [...]
  ALUKE = 0;
  if(T < 30000.) ALUKE = LUKEOP(...);
  [...]
}
```

Since the compilers cannot know if `LUKEOP` will be executed for all vector elements, they choose to execute a scalar version instead. This, in addition, prevents vectorization of the entire `cop` function.
One solution is to enforce that `LUKEOP` always performs all calculations for all vector elements, and in the end to zero out the elements of the result, for which it shouldn't have been called:

```
VECTORIZED_FUNCTION()
double LUKEOP(...)
{
  if(T >= 30000.) STIM=0;

  return  ( N1OP(FREQ,TKEV          )*XN1 +
            O1OP(FREQ               )*XO1 +
            Mg2OP(FREQ,TKEV         )*XMg2+
            Si2OP(FREQ,FREQLG,T,TLOG)*XSi2+
            Ca2OP(FREQ,TKEV         )*XCa2)*STIM;
}

VECTORIZED_FUNCTION()
void cop(...)
{
  [...]
  ALUKE = LUKEOP(...);
  [...]
}
```

While the above approach does result in a vectorized code, in some cases it takes longer than necessary. In particular, if `T >= 30000` for all vector elements then `LUKEOP` doesn't need to be called at all. There is no simple way to implement this check within the OpenMP framework, but it can be done explicitly using intrinsics, as described in *Section 4.2.4 Hand-tuning of automatically vectorized functions*.

4.2.4. Hand-tuning of automatically vectorized functions

Although modern compilers usually produce better code than any programmer, in some cases automatic vectorization of a function might not yield optimal results. One example is the `COULFF` function in `cop.cc`. It returns a value of a two-dimensional function by converting the `double` input arguments to integer indices and performing a lookup in a static pre-computed table. In a vectorized function each vector element will most likely result in a different lookup index. Compilers implement this on modern CPUs using a `vgather` instruction, which reads data from non-contiguous memory addresses into a vector register.
Despite compiling for AVX2, for the considered function GCC can only emit SSE2 instructions (2 `double`s in a vector instead of 4). In AVX clones, when the input arguments are 4-wide vectors, GCC first splits them in half, performs computations on each half separately, and reassembles the result. The problem is triggered by how GCC combines vectorization of `int` and `double` computations in the same function [8]. The Intel compiler correctly vectorizes this function using 4-wide vectors.
To improve performance of GCC compiled code `COULF` has been re-implemented manually using compiler intrinsics. The optimized code is shorter than what ICC produces (50 vs. 67 machine instructions). Performance difference relative to the Intel code was small, but this approach solves the inefficient GCC implementation problem. Since vectorization is based on the OpenMP pragmas and automatically generated vectorized clones,

some effort is needed to force GCC to use the intrinsics based implementation instead of its own clone (see Appendix C.).

5. Performance

For performance testing all codes were compiled with Intel 18.0.1 and GCC 8.1.0. Using GCC version at least 8.1.0 is required, because due to a bug found in GCC the compiler does not produce optimal RT code with the earlier versions [9]. With GCC, a custom-built Glibc 2.27 with its vectorized math library `libmvec` was used. For MPI tests we used OpenMPI 3.1.0 (GCC) and Intel MPI (ICC).
The code was executed on a Broadwell system (E5-2683 v4). In sequential tests the computing process was bound to first available core using `numactl --physcpubind=1`. Execution time was measured by enclosing the relevant portions of the code inside `tic()/toc()` timer calls calls implemented with `gettimeofday`.

5.1. Math Library Benchmarks

`libm` functions relevant for ART were benchmarked by measuring execution time of the following type of vectorized and scalar loops:

```
    tic();
    for(j=0; j<ntests; j++)
  #pragma omp for simd
      for(i=0; i<nel; i++){
        out[i] = exp(in[i]); // pow, log
      }
    toc();
```

The array size was kept small and similar to the data sizes operated on by ART (`nel=10e3`). Test loops were executed `ntests=10e3` times and the total execution time was measured. The following table presents the execution time (in seconds) and speedup of vectorized vs. scalar implementation (in parentheses):

	GCC		**ICC**		
function	scalar	vectorized	scalar	high accuracy vectorized	low accuracy vectorized
`exp`	1.35	0.18 (7.5x)	0.78	0.25 (3.1x)	0.18 (4.3x)
`pow`	5.49	1.04 (5.3x)	1.95	1.13 (1.7x)	0.80 (2.4x)
`log`	1.96	0.25 (7.3x)	0.96	0.37 (2.6x)	0.26 (3.7x)

Table 1. Execution time (in seconds) and speedup of vectorized vs. scalar implementations (in parentheses) of basic libm functions.

The scalar implementation in ICC is around 2 times faster than what is provided in Glibc. When using ICC with low accuracy settings, the vectorized `pow` is 20% better than in GCC, while performance of `exp` and `log` are essentially the same for both compilers. When ICC uses high accuracy vector ops, GCC is faster in all cases.
On the Broadwell architecture the vector width is 4 `double`s with AVX2, so an ideal speedup of a vectorized vs. best scalar implementation is maximum 4. With GCC speedups reach up to 7.5 times. This is due to the stricter FP rules for the scalar implementation. With ICC the speedups are from 3.7 times (`exp`) down to 2.4 times (`pow`). Overall, in terms of performance of the vectorized math library both compilers achieve essentially the same, very good results, and provide a substantial improvement over the scalar implementation.

5.2. EOS solver

Computation time required by the EOS solver depends on the input data. The role of EOS is to pre-compute certain physical properties, which are not provided as input, but are later used by the RT solver. Performance analysis has been performed for three basic configurations of input:

- `pgas`: input includes gas pressure, density, and temperature
- `rho`: input includes density and temperature
- `nne`: input includes electron density and temperature

Tests were performed for an example synthetic dataset produced by the Bifrost software (`en024048_hion`). Spatial resolution of the test data was 16 X 16 X 496. Time was measured by enclosing the computational part of

the code inside `tic()/toc()` calls, i.e., code startup was not included in the measurements. The following table shows execution time in seconds of the original (`scalar`) and the optimized (`vectorized`) code:

	GCC		**ICC**		
function	scalar	vectorized	scalar	high accuracy vectorized	low accuracy vectorized
`pgas`	5.7	0.44 (13x)	4.4	0.46 (9.5x)	0.40 (11x)
`rho`	9.7	1.11 (8.7x)	7.2	1.25 (5.8x)	1.06 (6.8x)
`nne`	0.6	0.05 (12x)	0.6	0.06 (10x)	0.05 (12x)

Table 2. Execution time (in seconds) and speedup of vectorized vs. scalar implementations (in parentheses) of the EOS solver for different input data.

For this part of the code, vectorization together with other optimizations produced very good results, with speedups ranging from 7 to 13 times. Interestingly, GCC performance is essentially on a par with ICC performance for the optimized code. The speedups are better than what would be expected from only switching to a vectorized math library because a number of other code optimizations are in place (see *Section 4.1 EOS solver*).

5.3. RT solver

The RT solver takes the output of the EOS as its input data. It computes on the original 3D grid and as the result produces a 2D image from the 3D stacks, i.e., the Z-column in the model is reduced to a single pixel. This is done once for each wavelength of interest.
Performance tests were carried out as for the EOS solver. The table below presents time in seconds required to process 100 wavelengths.

GCC		**ICC**		
scalar	vectorized	scalar	high accuracy vectorized	low accuracy vectorized
9.3	3.8 (2.5x)	9.4	5.9 (1.6x)	4.5 (2.1x)

Table 3. Execution time (in seconds) and speedup of vectorized vs. scalar implementations (in parentheses) of the RT solver.

Performance improvements for the RT solver are more modest than for the EOS. In this case GCC shows both best speedup (2.5 times), and best overall performance. Since the code is limited by the performance of the vectorized math library, the maximum theoretical speedup is 4 times. However, as described in *Chapter 4.2 RT solver*, because of iterative algorithms, and due to a large number of `if`-statements, this portion of the code doesn't vectorize with 100% efficiency.

5.4. Vtune Analysis of Optimized Code

FPU Utilization: 6.9%

SP FLOPs per Cycle: 2.216 Out of 32
Vector Capacity Usage: 90.1%
FP Instruction Mix:
% of Packed FP Instr.: 94.0%
% of 128-bit: 10.7%
% of 256-bit: 83.3%
% of Scalar FP Instr.: 6.0%
FP Arith/Mem Rd Instr. Ratio: 0.966
FP Arith/Mem Wr Instr. Ratio: 5.332

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	FPU Utilization	Vector Instruction Set	Loop Type
__svml_exp4_l9	2.414s	8.8%	AVX(128); AVX(256); AVX2(256); FMA(256)	
cop	1.474s	4.3%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(256); FMA(256)	
_ZNK3eos4witt11rho_from_peldEET_S2_S2_PS2	0.436s	18.5%	AVX(128); AVX(256); AVX2(256); FMA(256)	
COOLOP	0.359s	1.5%	AVX(128); AVX(256); AVX2(128); AVX2(256); AVX2GATHER(256); FMA(256)	
LUKEOP	0.270s	1.9%	AVX(128); AVX(256); AVX2(128); AVX2(256); AVX2GATHER(256); FMA(256)	
[Others]	1.107s	N/A*		

*N/A is applied to non-summable metrics.

Figure 4. Vtune performance summary of the optimized code. Single core execution, Intel compiler.

FPU Utilization: 3.9%

SP FLOPs per Cycle: 1.249 Out of 32
Vector Capacity Usage: 84.5%
FP Instruction Mix:
% of Packed FP Instr.: 87.3%
% of 128-bit: 12.0%
% of 256-bit: 75.4%
% of Scalar FP Instr.: 12.7%
FP Arith/Mem Rd Instr. Ratio: 0.316
FP Arith/Mem Wr Instr. Ratio: 1.461

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	FPU Utilization	Vector Instruction Set	Loop Type
exp_avx2	1.478s	6.9%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(256); FMA(256)	
cop	0.743s	3.0%	AVX(256); AVX2(256); FMA(256)	
_ZNK3eos4witt4gascldEET_S2_S2_PS2	0.421s	7.1%	AVX(128); AVX(256); AVX2(256); FMA(256)	
Si1OP	0.317s	0.6%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(128); FMA(128)	
Si2OP	0.299s	0.3%	AVX(128); AVX(256); AVX2(256); AVX2GATHER(128); FMA(128)	
[Others]	1.440s	N/A*		

*N/A is applied to non-summable metrics.

Figure 5. Vtune performance summary of the optimized code. Single core execution, GCC compiler.

Figure 4 and Figure 5 show VTune FPU Utilization reports for the optimized code compiled with both tested compilers. Compared to the original report, in which over 90% of FP instructions were scalar (Figure 2), in the optimized code 94% and 87% of FP instructions are vectorized for ICC and GCC, respectively. Note that although GCC has a lower fraction of vectorized instructions, its overall performance is better.

Most compute time is spent inside the vectorized math library (for the considered input - the `exp`), which can hardly be improved. Some potential improvements include a better implementation of the `COOLOP` and `LUKEOP` functions. Since those functions now take considerable time, one can skip their computations for data that doesn't require it, as explained in *Section 4.2.3Vectorization of data-dependent `if-statements`*. This would demand an intrinsics-based implementation of those functions.

For GCC, further improvements can be obtained by intrinsics-based implementation of other `COULFF`-like functions with `double` to `int` conversions and table lookups (`Si1OP`, `Si2OP`, `Mg1OP`, `Fe1OP`), as described in *Section 4.2.4 Hand-tuning of automatically vectorized functions*. Because of a bug / design decision, GCC cannot utilize full 4-wide vectors in those functions, and instead uses 2-wide vectors. Overall, VTune analysis confirms that vectorization of the code has been achieved.

5.5. Parallel Performance and Scalability

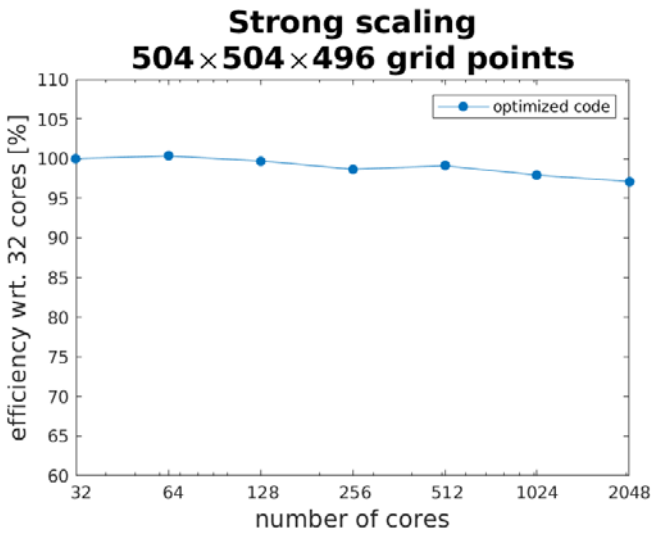

Figure 6. Strong scaling and parallel efficiency of the optimized code on a Broadwell-based HPC installation.

Since the computations do not involve any communication between the participating MPI ranks one can expect a very good parallel efficiency for both the original, and the optimized code. Figure 6 confirms this expectation. Strong scaling of the optimized code is presented for a relatively large problem on up to 2048CPU cores. The parallel efficiency with respect to 32 cores exceeds 95%, and the total EOS+RT runtime on 2k cores is ~2.5s.

6. Conclusions

Within this *Preparatory Access* project, we concentrated mostly on per-core optimization of the ART software package. Amongst others, all critical path functions of the code have been optimized and vectorized using the OpenMP `declare simd` directives. To achieve this goal several techniques have been used:

- Rearrangement of the input data and the internal data structures to facilitate usage of CPU's vector units.
- Vectorization of calls to the math library (`exp, log, pow`).
- Vectorization of functions that return multiple values through pointer and reference variables (difficult because of OpenMP limitations).
- Vectorization of functions that receive pointer arguments, which can be `NULL` (difficult because of OpenMP limitations).
- Explicit loop unrolling to allow vectorization of iterative loops with a convergence criterion.
- Vectorization of data-dependent if-statements through enforced computations on all SIMD lanes and filtering of the result.

A number of technical challenges needed to be overcome to achieve best performance results:

- The OpenMPI stack needed to be compiled with a custom (non-native) Glibc library.
- Individual vectorized clones generated automatically by the compilers needed to be substituted with custom functions implemented using intrinsics.

Throughout the project a number of GCC bugs related to automatic OpenMP vectorization have been submitted [6, 9, 8, 10]. This shows that the support for the relatively new OpenMP vectorization features is still not mature. For some of those bugs effective workarounds have been developed.
Working with ART also pointed to some shortcomings in the OpenMP `simd` vectorization framework. It is for example not possible to return multiple values from functions, e.g., through pointer or reference arguments, unless those arguments point to `linear` arrays [5]. This makes it essentially impossible to pass local temporary variables to vectorized functions. A way around it is to inline such functions, which is of course not always possible.

The performance tests have shown that on a Broadwell-based architecture the optimized code works from 2.5 times faster (RT solver) to 13 times faster (EOS solver) on a single core. The parallel MPI implementation scales with more than 95% efficiency on 2048 cores for relatively small problem sizes. The small loss of efficiency is partly related to the fact that the amount of work performed to some extent depends on the input data. For example, in a vectorized iterative solver all vector elements participate in the same (maximum) number of calculations, while in a scalar code the code may bail out earlier for some corresponding data entries.
Apart from improving the ART efficiency, this project also resulted in several new optimization techniques, which improve effectiveness of automatic code vectorization. Finally, a few generally useful tutorials have been delivered (Appendix A. ABI (symbol names) of Vectorized Functions, Appendix B. Building OpenMPI with a custom Glibc, Appendix C. Combining `omp declare simd` with intrinsics.).

7. Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.

8. References

[1] J. McCaplin, "answer to question: AVX512 auto-vectorization on i9-7900X," [Online]. Available: https://software.intel.com/en-us/forums/intel-isa-extensions/topic/747994.

[2] Intel Corporation, "ICC Documentation, PROCESSOR Clause," [Online]. Available: https://software.intel.com/en-us/node/679659.

[3] OpenMP Architecture Review Board, "OpenMP} Application Program Interface Version 4.0," July 2013. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[4] Intel Corporation, "Intrinsics for Short Vector Math Library (SVML) Functions," [Online]. Available: https://software.intel.com/en-us/node/524289.

[5] M. Krotkiewski, "Multiple return values from an omp declare simd function," Feb 2018. [Online]. Available: http://forum.openmp.org/forum/viewtopic.php?f=3&t=2030.

[6] M. Krotkiewski, "Bug 84261 - gcc fails to vectorize a function call," Feb 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84261.

[7] R. Kurucz, "Atlas: A Computer Program for Calculating Model Stellar Atmospheres," Smithsonian Institution Astrophysical Observatory, 1970.

[8] M. Krotkiewski, "Bug 85050 - Vectorized function - suboptimal gather," Mar 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=85050.

[9] M. Krotkiewski, "Bug 84903 - internal compiler error: in convert_move, at expr.c:229," Mar 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84903.

[10] M. Krotkiewski, "Bug 85232 - gcc fails to vectorize a nested simd function call," Apr 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=85232.

[11] The glibc team, "Vector Function Application Binary Interface Specification for OpenMP," 2015. [Online]. Available: https://sourceware.org/glibc/wiki/libmvec?action=AttachFile&do=view&target=VectorABI.txt.

[12] X. T. a. H. S. a. S. K. a. K. B. S. a. R. G. a. M. G. a. S. V. Preis, *Vector (SIMD) Function ABI*, Intel Corporation, 2015.

Appendix A. ABI (symbol names) of Vectorized Functions

Consider the following example vectorized C function:

```
#include <math.h>

#pragma omp declare simd
double vfun(double v1, double v2)
{
  return exp(v1-v2)/2;
}
```

The code is compiled using GCC:

```
$ gcc -O3 -fopenmp vfun.c -c
```

Looking at the contents of the generated object file one can see several `vfun` symbols:

```
$ nm vfun.o | grep vfun
0000000000000000 T vfun
00000000000001c0 T _ZGVbM2vv_vfun
00000000000000e0 T _ZGVbN2vv_vfun
00000000000002e0 T _ZGVcM4vv_vfun
0000000000000270 T _ZGVcN4vv_vfun
0000000000000540 T _ZGVdM4vv_vfun
00000000000004d0 T _ZGVdN4vv_vfun
0000000000000700 T _ZGVeM8vv_vfun
0000000000000660 T _ZGVeN8vv_vfun
```

The naming convention used by GCC for the vectorized symbols is described in detail in [11]. A slightly different ABI is used by the Intel compiler [12]. In the above listing the first line describes the scalar symbol, while the vectorized symbols start with `_ZGV`. The `[b,c,d,e]` letters denote different CPU architectures, `[2,4,8]` denote vector width, and `[M,N]` denote masked (i.e., not all vector elements participate in computations) and non-masked version of the function.
Having a basic understanding of the symbol naming convention is useful when looking at the assembly code and making sure that the compiler indeed generated instructions we would like. Consider the following code, which calls `vfun`:

```
double test(int n, double *v1, double *v2, double *out)
{
  #pragma omp simd
  for(int i=0; i<n; i++){
    out[i] = vfun(v1[i], v2[i]);
  }
}
```

The code is compiled using GCC, which reports that the loop has been vectorized:

```
$ gcc -fopenmp -O3 test.c vfun.o -fopt-info-vec
test.c:16:28: note: loop vectorized
```

Looking at the loop assembly we can verify that a call to a vectorized variant of `vfun` is executed

```
.L7:
        movupd  0(%rbp,%rbx), %xmm1
        movupd  (%r12,%rbx), %xmm0
        call    _ZGVbN2vv_vfun
        movups  %xmm0, 0(%r13,%rbx)
        addq    $16, %rbx
        cmpq    %r15, %rbx
        jne     .L7
```

However, inspection of the `_ZGVbN2vv_vfun` assembly reveals that the compiler generated a loop with scalar calls to `exp` instead of calling a vectorized version:

```
.L16:
        movsd   16(%rsp,%rbx,8), %xmm0
        subsd   32(%rsp,%rbx,8), %xmm0
        call    exp@PLT
        [...]
        jne     .L16
```

There are two problems here. With GCC we need to add the `-ffast-math` flag, since by default - due to accuracy considerations - calls to `libm` are not vectorized. Also, since we did not specify the architecture, the compiler chose a conservative approach (`_ZGV`**`b`** means *SSE*). With the following compilation flags:

```
$ gcc -march=broadwell -ffast-math -fopenmp -O3 -c vfun.c -fopt-info-vec
vfun.c:9:10: note: loop vectorized
vfun.c:9:10: note: loop vectorized
vfun.c:9:10: note: loop vectorized
vfun.c:9:10: note: loop vectorized
vfun.c:9:10: note: loop vectorized
vfun.c:9:10: note: loop vectorized
```

`vfun` itself is also reported as vectorized (6 times - once for each generated vectorized variant}. Compiled as above, the `test` loop calls the AVX2 variant of `vfun`, which itself calls a vectorized `exp` implementation:

```
_ZGVdN4vv_vfun:
        [...]
        vsubpd  %ymm1, %ymm0, %ymm0
        call    _ZGVdN4v___exp_finite@PLT
        vmulpd  .LC1(%rip), %ymm0, %ymm0
        [...]
```

Appendix B. Building OpenMPI with a custom Glibc

The following has been tested on a Centos 7.4 system with GCC 8.1.0. First, Glibc 2.27 is compiled and installed in a custom directory `$GLIBC_DIR` as follows:

```
$ tar xaf glibc-2.27.tar.bz2
$ mkdir glibc-2.27/build
$ cd glibc-2.27/build
$ ../configure --prefix=$GLIBC_DIR
$ make -j 32
$ make install
```

By default the custom dynamic loader will look for libraries in `$GLIBC_DIR/lib`, and - if present - it will use `$GLIBC_DIR/etc/ld.so.cache`. Since we want the custom Glibc to have access to the usual system libraries, we need to update the cache:

```
$ echo /lib64 > $GLIBC_DIR/etc/ld.so.conf
$ echo /usr/lib64 >> $GLIBC_DIR/etc/ld.so.conf
$ $GLIBC_DIR/sbin/ldconfig
```

If needed, one should also copy the entire contents of the system `/etc/ld.so.conf` and `/etc/ld.so.conf.d` to `$GLIBC_DIR/etc` and re-run `ldconfig`.
Compiling programs against a custom Glibc requires using some non-standard GCC compilation flags. One needs to tell GCC to ignore the native Glibc, which is done with the `-nostdinc` switch. Also, the custom Glibc include path needs to be specified, together with internal GCC paths:

```
$ gcc -O3 -g -nostdinc \
 -I$GCC_DIR/include \
 -I$GCC_DIR/lib/gcc/x86_64-pc-linux-gnu/$GCC_VER/include \
 -I$GCC_DIR/lib/gcc/x86_64-pc-linux-gnu/$GCC_VER/include-fixed \
 -I$GLIBC_DIR/include \
 -c test.c
```

The internal GCC paths can be obtained on the command line, e.g.:

```
$ gcc -xc -v -
[...]
#include <...> search starts here:
 <GCC_DIR>/lib/gcc/x86_64-pc-linux-gnu/<GCC_VER>/include
 /usr/local/include
 <GCC_DIR>/include
 <GCC_DIR>/lib/gcc/x86_64-pc-linux-gnu/<GCC_VER>/include-fixed
 /usr/include
End of search list.
```

Note that the order in which we add the include paths on the command line matters. If we want to use the new Glibc, but still have access to other header files installed on the system, we should add `-I/usr/include` after `-I$GLIBC_DIR/include`. This way the custom Glibc will be picked up first.
The internal C++ include paths are obtained in a similar fashion:

```
$ g++ -xc++ -v -
[...]
#include <...> search starts here:
 <GCC_DIR>/include/c++/<GCC_VER>
 <GCC_DIR>/include/c++/<GCC_VER>/x86_64-pc-linux-gnu
 <GCC_DIR>/include/c++/<GCC_VER>/backward
 <GCC_DIR>/lib/gcc/x86_64-pc-linux-gnu/<GCC_VER>/include
 /usr/local/include
 <GCC_DIR>/include
 <GCC_DIR>/lib/gcc/x86_64-pc-linux-gnu/<GCC_VER>/include-fixed
 /usr/include
End of search list.
```

When linking programs against a custom Glibc GCC needs to be told to ignore the standard libraries, use correct Glibc CRT startup / finish functions, and to use the custom dynamic loader. Also, specifying `RUNPATH` that points to the custom Glibc is useful. For example, when linking a dynamic ELF executable:

```
BEGFILES="$GLIBC_DIR/lib/crt1.o $GLIBC_DIR/lib/crti.o \
$(gcc --print-file-name=crtbegin.o)"
ENDFILES="$GLIBC_DIR/lib/crtn.o $(gcc --print-file-name=crtend.o)"
LDFLAGS="-nostdlib -L$GLIBC_DIR/lib $GLIBC_DIR/lib/libc.so \
-Wl,--start-group \
-lgfortran -lgcc_s -lgcc -lgcc_eh -lstdc++ \
-Wl,--end-group \
-Wl,--rpath=$GLIBC_DIR/lib \
-Wl,--rpath=$GCC_DIR/lib64 \
-Wl,--enable-new-dtags \
-Wl,--dynamic-linker=$GLIBC_DIR/lib/ld-linux-x86-64.so.2"

$ gcc $BEGFILES $ENDFILES $LDFAGS test.o -o test
```

Note that different start / end files are used when linking shared objects (the `-shared` flag):

```
BEGFILES="$GLIBC_DIR/lib/crti.o $(gcc --print-file-name=crtbeginS.o)"
ENDFILES="$GLIBC_DIR/lib/crtn.o $(gcc --print-file-name=crtendS.o)"
```

This is a complication, which makes it impossible to implement a general-purpose compilation environment only through environment variables pre-set with custom compile and link flags. To obtain a general-purpose compilation environment one needs to implement suitable wrappers for GCC, G++, and GFORTRAN. Here we call those wrappers `gccwrap, g++wrap, gfortranwrap`, respectively. A draft of the wrapper, with details left out for brevity, is as follows:

```
#!/bin/bash

ARGS="$@"

# gcc information
GCC_DIR=$(dirname $(dirname `which gcc`))
GCC_VER=$(gcc --version | grep ^gcc | sed 's/^.* //g')

# what are we wrapping?
prog=$(echo $0 | sed -e 's/wrap//')
prog=$(basename $prog)

# analyze the arguments, check if we are compiling, linking, or other
[...]
if test $compile == 0; then
   # not compiling - do not wrap
   $prog $ARGS
   exit $?
fi

# update CFLAGS and CXXFLAGS with -nostdinc and a new Glibc path
CFLAGS=[...]
CXXFLAGS=[...]

# linking - different setup for shared and runnable objects
if test $link != 0; then
   LDFLAGS=[...]
   if test $shared != 0; then
      # shared library start / end files
      BEGFILES=[...]
      ENDFILES=[...]
   else
      # runnable start / end files
      BEGFILES=[...]
      ENDFILES=[...]
   fi
fi

# fix double quotation marks around arguments, e.g., -DVAR="value"
ARGS=$(echo $ARGS | sed -e 's/"\([^"]*\)"/\\\""\1"\\\"/'g)

# execute the wrapped program with modified environment
eval $prog $ARGS $BEGFILES $ENDFILES $LDFLAGS $CXXFLAGS $CFLAGS
```

Using the wrapper, we can now compile and install OpenMPI, which uses the custom Glibc. On a Mellanox Infiniband system with HPCX libraries:

```
$ tar xaf openmpi-3.1.0.tar.bz2
$ cd openmpi-3.1.0
$ ./configure CC=gccwrap CXX=g++wrap FC=gfortranwrap \
--prefix=$OMPI_DIR \
--with-knem=${HPCX_HOME}/knem \
--with-mxm=${HPCX_MXM_DIR} \
--with-hcoll=${HPCX_HCOLL_DIR} \
--with-ucx=${HPCX_UCX_DIR} \
--with-platform=contrib/platform/mellanox/optimized \
$ make -j 32
$ make install
```

For `configure` to succeed several external development packages are needed. Since the wrappers can access the local libraries, those can be simply installed from native OS package manager.

At the time of writing, on newer ConnectX-4 systems the custom OpenMPI runtime cannot find one library:

```
$ mpicc test.c -o test
$ mpirun -np 2 ./test
libibverbs: Warning: couldn't load driver 'mlx5':
libmlx5-rdmav2.so: cannot open shared object file:
No such file or directory
[...]
```

The file is installed as part of the `libmlx5-41mlnx1-OFED` package in `/usr/lib64`, and is a link to `libmlx5.so.1.0.0`. It is not picked up by the custom dynamic loader (non-matching `SONAME`) and has to be manually copied into `$GLIBC_DIR/lib`. The installation is then complete and fully functional.
Note that OpenMPI compiled in the above way will use the GCC wrappers inside `mpicc` to compile user code, hence nothing needs to be changed on the user-side to be able to use the custom Glibc.

Appendix C. Combining `omp declare simd` with intrinsics.

Automatic vectorization of OpenMP `omp declare simd` functions results in several SIMD clones being generated by the compilers. These have well-defined symbol names that depend on the architecture and vector length ([11, 12]). It is of course possible to implement such vectorized functions manually, and make sure the compiler uses them instead of the automatically generated ones. In the following example we define `vfun`, which computes and returns `exp` of the argument. We also define one vectorized clone for AVX2 architectures with vector width of 4 (`_ZGVdN4v_vfun`):

```
manual.c:

    #include <stdio.h>
    #include <math.h>
    #include <immintrin.h>

    double vfun(double i) {
      return exp(i);
    }

    // declare the needed vectorized math functions from GCC / libmvec
    extern __m256d exp_4v(__m256d) asm("_ZGVdN4v___exp_finite");

    // declare and define the vectorized vfun symbol
    __m256d vfun_4v (__m256d) asm ("_ZGVdN4v_vfun");
    __m256d vfun_4v(__m256d i) {
      return exp_4v(i);
    }
```

The appropriate vectorized `exp` symbol (`_ZGVdN4v___exp_finite`) from `libmvec` is declared for the vectorized clone to be able to call it. The declaration needs to be done using the `asm` directive to reference by name the correct binary symbol. Then, the vectorized `vfun` can be called just like any other vectorized function, e.g., from inside an `omp simd` loop:

```
main.c:

    #include <stdio.h>
    #include <stdlib.h>

    #pragma omp declare simd simdlen(4)
    double vfun(double i);

    int main(int argc, char*argv[]) {
      int n = atoi(argv[1]);
      double *vals = (double*)malloc(sizeof(double)*n);

      #pragma omp for simd
      for(int i=0; i<n; i++)

        // this call will be vectorized,
        // but only on AVX2 architectures
        vals[i] = vfun((double)i);
    }
```

While the above usage of the vector ABI provides a simple way to manually implement vectorized functions, in general it requires that all SIMD clones are implemented manually. Otherwise the scalar implementation will be used. In some cases, it may be desirable to improve only one particular clone, and rely on automatic vectorization with `omp declare simd` for all other architectures. This is possible with the use of *weak symbols*. First, produce automatically vectorized clones of `vfun`:

```
auto.c:

    #include <math.h>

    #pragma omp declare simd notinbranch
    double vfun(double i) {
      return exp(i);
    }

$ gcc -fopenmp -march=broadwell -O3 -ffast-math -c auto.c
  \end{Verbatim}
}
```

Verify that all clones have been generated:

```
$ nm auto.o
[...]
0000000000000000 T vfun
0000000000000010 T _ZGVbN2v_vfun
0000000000000080 T _ZGVcN4v_vfun
0000000000000090 T _ZGVdN4v_vfun
00000000000000a0 T _ZGVeN8v_vfun
```

Since we want to use our own implementation of `_ZGVdN4v_vfun`, we use `objcopy` to change this symbol in the automatically generated object file into a weak symbol:

```
$ objcopy --weaken-symbol _ZGVdN4v_vfun auto.o
$ nm auto.o
[...]
0000000000000000 T vfun
0000000000000010 T _ZGVbN2v_vfun
0000000000000080 T _ZGVcN4v_vfun
0000000000000090 W _ZGVdN4v_vfun
00000000000000a0 T _ZGVeN8v_vfun
```

Since weak symbols are overridden by strong symbols at link time, the final executable with the intrinsics-based implementation of `_ZGVdN4v_vfun` can be linked by simply providing the necessary manually optimized object file:

```
$ gcc -o main main.o auto.o manual.o
```