How to Write Your Own Blazingly Fast Library of Special Functions for Intel Xeon Phi Coprocessors

Vadim Karpusenko
Colfax International

Andrey Vladimirov
Stanford University

May 3, 2013

Abstract

Statically-linked libraries are used in business and academia for security, encapsulation, and convenience reasons. Static libraries with functions offloadable to Intel Xeon Phi coprocessors must contain executable code for both the host and the coprocessor architecture. Furthermore, for library functions used in data-parallel contexts, vectorized versions of the functions must be produced at the compilation stage.

This white paper shows how to design and build statically-linked libraries with functions offloadable to Intel Xeon Phi coprocessors. In addition, it illustrates how special functions with scalar syntax $\big($*e.g.*, $y = f(x)\big)$ can be implemented in such a way that user applications can use them in thread- and data-parallel contexts. The second part of the paper demonstrates some optimization methods that improve the performance of functions with scalar syntax on the multi-core and the many-core platforms: precision control, strength reduction, and algorithmic optimizations.

Contents

1 Introduction

For highly parallel applications, Intel Xeon Phi coprocessors yield better performance and energy efficiency than the latest generation Intel Xeon CPUs with similar energy consumption. Intel Xeon Phi coprocessors are based on the Many Integrated Core (MIC) architecture featuring over 50 cores with four-way hyper-threading and 512-bit wide vector registers. A theoretical peak performance of 1 TFLOP/s in double precision can be achieved in applications for Intel Xeon Phi coprocessors if both thread parallelism data parallelism in the form of SIMD (Single Instruction for Multiple Data) operations are utilized in the application.

A large subset of applications that can benefit from the MIC architecture are problems in which a significant amount of computing time is spent on the evaluation of the values of special functions of one or several variables:

$$y = f(x, y, z, \dots), \tag{1}$$

where the function must be computed for an array of argument values:

$$y_i = f(x_i, y_i, z_i, \dots), \qquad i \in 0 \dots N. \tag{2}$$

Routines of this type are used in grid-based and Monte Carlo calculations in which the values of quantities on the grid or in the random set of points are expressed via

a) special mathematical functions, such as exponential integrals, gamma functions, Bessel functions, hypergeometric functions, and other,

b) analytical fits to experimental trends: particle reaction cross sections, radiation spectra, statistical distributions, empirical laws etc.,

c) solutions of differential equations (for example, the Black-Scholes formula [1]), or

d) approximations of complex functional dependencies (e.g., asymptotic expansions).

In applications that apply these routines to calculate multiple scalar arguments in an array, the developer can take advantage of a parallel framework such as OpenMP and of the automatic vectorization capability of the Intel compiler to parallelize the calculation across all available resources as outlined in Listing 1.

```
1  // Declaration, implementation and usage of SomeFunction() are all in the same source file
2  float SomeFunction(const float arg) {
3     // ...
4  }
5
6  float* x= ...
7  float* y= ...
8
9  // This loop may be automatically vectorized by the compiler,
10 // because the compiler sees the code of the body of SomeFunction()
11 #pragma omp parallel for
12 #pragma simd
13 for (int i = 0; i < N; i++)
14     y[i] = SomeFunction(x[i]);
```

Listing 1: Layered thread and data parallelism in the calculation of `SomeFunction()` for an array of arguments `x[]`. This code can be automatically vectorized by the Intel compiler.

In Listing 1, the loop that calls `SomeFunction()` on an array of arguments is automatically distributed across multiple threads as requested by `#pragma omp`. Additionally, `#pragma simd` requests automatic vectorization, i.e., that the compiler packs multiple consecutive values of the array elements into a SIMD register, and simultaneously runs all SIMD vector lanes through `SomeFunction()`, thus layering data parallelism with thread parallelism. Automatic vectorization may or may not work,

depending on the operations in the the function. However, the very possibility of automatic vectorization in this case relies on the visibility of the code of this function to the compiler.

In some cases, for convenience or security reasons, or for commercial distribution, the developer may wish to place `SomeFunction()` into a statically-linked library. In this scenario, the application that calls `SomeFunction()` (the user application) is implemented in a separate file. As a consequence, the compiler will refuse to vectorize the loop in the user application, because the source code of the library function is not visible in this case. The failure of vectorization may penalize the application performance by up to a factor of 16x on the MIC architecture and by up to 8x on a multi-core processor. However, special measures can be taken in order to forewarn the compiler of the possibility of automatic vectorization of `SomeFunction()` at the time that the library is compiled. This may allow to vectorize the function call in the user application.

Additionally, if the library is used in applications running on a Intel Xeon Phi coprocessor in the offload mode, the library archive must contain functions compiled for the MIC architecture. Otherwise, when the offload point is reached, the user application will fail with a runtime error.

This white paper demonstrates step-by-step the process of creation of a static library with two requirements:

a) Library functions must be executable from user applications running on an Intel Xeon Phi coprocessor in the offload mode, and

b) Automatic vectorization must succeed in user applications that execute library functions with scalar syntax on an array of arguments.

This process is demonstrated in Section 2. After that, in Section 3, we demonstrate some of the optimization practices for functions with scalar syntax designed for the Intel Xeon family processors. Our optimizations enable the library function that we implemented as an educational example to perform on par with the highly optimized Intel's implementation. This leads to a discussion in Section 4 of programming in a high level language compared to the usage of inline assembly or intrinsics.

2 Building and using a static library with offloadable functions

2.1 Example: the Gauss error function

The objective of this paper is to show how to create static libraries of special functions of one or several scalar variables, which can be used in the offload model for calculations on Intel Xeon Phi co-processors. To illustrate the process, we will create a library implementation of the Gauss error function, which is a function of one variable. We chose this function because a highly optimized implementation of the error function is provided in the Intel Math Library. Therefore, it is convenient to use Intel's implementation `erff()` to estimate the accuracy of our implementation and to compare its performance to a highly optimized implementation.

The definition of the Gauss error function [2], commonly denoted as $\mathrm{erf}(x)$, is defined by Equation (3). It is a special (i.e., non-elementary) function occurring in probability, statistics, and partial differential equations:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, \mathrm{d}t. \tag{3}$$

To calculate the value of the Gauss error function for argument x, our library implementation uses a rational approximation given by Abramowitz and Stegun [4]:

$$\begin{aligned} &\mathrm{erf}(x) \approx 1 - \left(a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5\right) e^{-x^2}, \\ &\text{where} \quad t = \frac{1}{1 + px}, \\ &p = 0.3275911, \quad a_1 = 0.254829592, \quad a_2 = -0.284496736, \\ &a_3 = 1.421413741, \quad a_4 = -1.453152027, \quad a_5 = 1.061405429. \end{aligned} \tag{4}$$

This approximation accurately $\left(\text{with a maximum error } \epsilon \le 1.5 \times 10^{-7}\right)$ represents the non-negative part of the Gauss error function. And since the error function is an *odd function*, the following property is used to calculate $\mathrm{erf}(x)$ for negative values of the argument:

$$\mathrm{erf}(-x) = -\mathrm{erf}(x). \tag{5}$$

The plots of the Gauss error function defined by Equation (3) and the rational approximation without the correction of negative side defined by Equation (4) are shown in Figure 1.

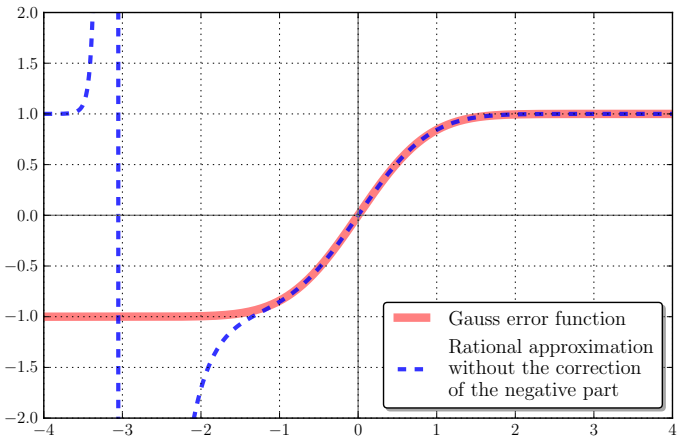

Figure 1: The Gauss error function (solid red line) and the rational approximation (4) (dashed blue line).

The implementation of the Gauss error function via a rational approximation is discussed in detail in Section 2.2. At the moment we assume that a scalar version of this function `MyErfScalar()` is already implemented in a library and can be called from the user application `Driver.cpp`. Listing 2 demonstrates how the values of the Gauss error function can be calculated for arguments from the array `fIn[]` in a parallel OpenMP environment:

```
#pragma omp parallel for
    for ( int i = 0; i<lTotal; i++)
        fOut[i] = MyErfScalar(fIn[i]);
```

Listing 2: Schematic code illustrating the usage of `MyErfScalar()`, a rational approximation of the Gauss error function.

In order to offload the execution of this loop to the Intel Xeon Phi coprocessors we need to specify additional pragmas, as discussed in the next section.

2.2 Offloading functions to a coprocessor

The library that we are constructing consists of two files: `MyLib.h`, which contains the declaration of the only function in the library (the Gauss error function), and `MyLib.cpp`, which contains the implementation of the function.

We intend to use this library in the offload mode, i.e., the application is launched on the host, but at some point it transfers some data and code to the coprocessor and uses the MIC architecture to perform a part of the calculation. Because the instruction sets for the host CPU and for Intel Xeon Phi coprocessors are different, the compiler must compile the code of the offloaded function twice: once for the host, and another time for the coprocessor. In order to instruct the compiler to do so, two lines should be used: `#pragma offload_attribute(push, target(mic))` and `#pragma offload_attribute(pop)`. These two lines should enclose the implementation and the declaration of the function as shown in Figures 3 and 4.

```
#pragma offload_attribute(push, target(mic))

#include <math.h>

float MyErfScalar(const float inx){

    const float x = fabs(inx);

    const float p  = 0.3275911;
    const float t  = 1/(1+p*x);
    const float t2 = pow(t, 2);
    const float t3 = pow(t, 3);
    const float t4 = pow(t, 4);
    const float t5 = pow(t, 5);

    const float res = 1.0 - (0.254829592*t - 0.284496736*t2 + 1.421413741*t3 -
                        1.453152027*t4 + 1.061405429*t5) * exp(-x*x);

    return copysign(res, inx);
}

#pragma offload_attribute(pop)
```

Listing 3: Unoptimized source code file `MyLib.cpp` with the implementation of the function `MyErfScalar()`.

```
#ifndef _MYLIB_H_
#define _MYLIB_H_

#pragma offload_attribute(push, target(mic))
float MyErfScalar(const float inx);
#pragma offload_attribute(pop)

#endif
```

Listing 4: Unoptimized header file `MyLib.h` with the declaration of the Gauss error function `MyErfScalar()`.

The function implementation in Figure 3 has a number of flaws that prevent it from achieving good performance. These flaws are corrected in Section 3. However, at this point, the code in Figure 3 is a straightforward translation of Equation (4).

The function `MyErfScalar()` can be used in a user application running on the host and in an application running on the coprocessor in the offload mode as shown in Listing 5. In order to offload the calculation to a coprocessor, `#pragma offload` should be used.

```
#include "MyLib.h"

void RunOnHost(){
  float* fIn = ... // Allocation and initialization of data
  float* fOut = ...
#pragma omp parallel for
    for ( int i = 0; i<lTotal; i++ )
        fOut[i] = MyErfScalar(fIn[i]);
}

void RunOnCoprocessor(){
#pragma offload target(mic:0)
    {
      float* fIn = ... // Allocation and initialization of data
      float* fOut = ...
#pragma omp parallel for
      for ( int i = 0; i<lTotal; i++ )
          fOut[i] = MyErfScalar(fIn[i]);
    }
}

int main() {
  ...
}
```

Listing 5: User application `Driver.cpp`, which calls the Gauss error function `MyErfScalar()` declared in `MyLib.h`.

The line `#pragma offload target(mic:0)` indicates that at runtime, the code following it should be offloaded to the first Intel Xeon Phi coprocessor (zero-based numbering system). If the coprocessor number is not specified, and several Intel Xeon Phi coprocessors are installed in the system, then the runtime offload library may choose any coprocessor for the offload execution. If no coprocessors are present, the code in the scope of `#pragma offload` is still executed, but the application runs on the host. This is another reason why the compiler must compile the code twice when the offload attribute is applied to a function.

In this example, we allocate and initialize the data used by the function on the coprocessor. In cases when some data must be transferred between the host and the coprocessor, the data transfer pattern must be included in the clauses of this pragma as described in the Intel C++ Compiler XE 13.1 User and Reference Guide [3].

2.3 Building a static library with offloadable code

Libraries with offloadable code are built similarly to libraries with host-only code. First, all source codes should be compiled into object files as shown in Listing 6.

```
vadim@skynet% icpc -c -o MyLib.o MyLib.cpp
vadim@skynet% ls
MyLib.cpp  MyLib.h  MyLibMIC.o  MyLib.o
```

Listing 6: Using the Intel C++ compiler to create object file `MyLib.o` for the host architecture and `MyLibMIC.o` for the MIC architecture. The latter file is created automatically when some of the objects in the source code are marked with the offload attribute.

After the compilation of `MyLib.cpp`, two additional files appear: `MyLib.o`, which is the object file with the host code, and `MyLibMIC.o`, which is the object file with the coprocessor code. The latter file was created because the function in `MyLib.cpp` is marked with the offload attribute (see Section 2.2).

When a static library is built, both the host and the coprocessor code must be packed into the library archive. The tool `xiar`, which comes with Intel C++ Composer XE, should be used to achieve this. The syntax of `xiar`'s command line arguments is similar to the syntax of the standard tool `ar` commonly used to create static libraries in Linux. Namely, common `ar` tool options can be used to `d` (Delete), `r` (Replace), `m` (Move), and `x` (Extract) static library components. However, to add the offloadable code, the argument `-qoffload-build` should be added (this argument is not supported by `ar`). It tells `xiar` to create both versions of static library: `libMyLib.a` for the host CPU, and the corresponding `libMyLibMIC.a` with offloadable code. For all manipulations with the static library, only the original name of the library should be provided. `xiar` automatically manipulates the corresponding coprocessor library and member filenames.

The process of library creation is shown in Listing 7. In our case, the library contains only one object file. For libraries with multiple object files, the names of the files should be listed at the end of `xiar`'s arguments, separated by spaces.

```
vadim@skynet% icpc -c -o MyLib.o MyLib.cpp
              ...
vadim@skynet% xiar cru -qoffload-build libMyLib.a MyLib.o
xiar: executing 'ar'
xiar: executing 'ar'
vadim@skynet% ls lib*
libMyLib.a  libMyLibMIC.a
```

Listing 7: Using the Intel C++ compiler to create the object file and `xiar` to archive it into a static library `libMyLib.a`.

2.4 Using the library functions

Now with the source code file of the library `MyLib.cpp` compiled and archived into a static library `libMyLib.a`, we can call the Gauss error function from `Driver.cpp` as shown in Listing 5. When linking a static archive that contains offloadable code, the standard linker options `-L<path>` and `-l<libname>` should be used to provide the information about the library location and its name. The compiler automatically incorporates the corresponding coprocessor library (`libMyLibMIC.a`) at the linking phase. Listing 8 shows the procedure for the compilation of the application and linking it with the static library `libMyLib.a` that we have just created.

```
vadim@skynet% icpc -c -openmp -vec-report6 -o Driver.o Driver.cpp
Driver.cpp(8): (col. 17) remark: vectorization support: call to function _Z11MyErfScalarf
cannot be vectorized.
Driver.cpp(7): (col. 5) remark: loop was not vectorized: existence of vector dependence.
Driver.cpp(18): (col. 12) remark: vectorization support: call to function _Z11MyErfScalarf
cannot be vectorized.
Driver.cpp(17): (col. 7) remark: loop was not vectorized: existence of vector dependence.
Driver.cpp(18): (col. 12) remark: *MIC* vectorization support: call to function
_Z11MyErfScalarf cannot be vectorized.
Driver.cpp(17): (col. 7) remark: *MIC* loop was not vectorized: existence of vector
dependence.
vadim@skynet% icpc -o runme -openmp Driver.o -lMyLib -L.
```

Listing 8: Compilation of `Driver.cpp` and linking the object file with the library program with the static library `libMyLib.a`. The resulting executable is not vectorized, as seen from the report generated by `-vec-report6` flag.

When the program `Driver.cpp` is compiled and linked with the library `libMyLib.a`, the resulting executable uses the offloadable function `MyErfScalar()` on an Intel Xeon Phi coprocessor in a parallel OpenMP context. However, according to the compilation report requested by the compiler flag `-vec-report6`, this code is not automatically vectorized.

Vectorization is crucial for computationally expensive calculations, especially on Intel Xeon Phi coprocessors. The loops in lines 7 and 17 of Listing 5 are not vectorized, because the library `MyLib.cpp` and the user application `Driver.cpp` are compiled separately. Vectorized form of the function is not produced during the compilation of `MyLib.cpp`, and the code of the function is not visible to the compiler when it processes the loops in `Driver.cpp`. In the next section we demonstrate how to enable automatic vectorization through the use of a language construct known as elemental functions.

2.5 Vectorization and elemental functions

Any C/C++ scalar function of one variable, which describes operations only on a singe data element, or several *independent* scalar elements, which have no interdependencies, can be declared as an *elemental function* using the qualifier `__attribute__((vector))`:

```
1  __attribute__((vector)) float MyElementalFunction(const float x, ...);
```

An elemental function can be called as a regular function to operate on a single scalar element:

```
1    float y = MyElementalFunction(x, ...);
```

or used in a loop that expresses a data-parallel computation:

```
1    float x[n], y[n];
2    for (int i = 0; i < n; i++)
3      y[i] = MyElementalFunction(x[i], ...);
```

or it can be used with array notation supported by the Intel C++ compiler:

```
1    float x[n], y[n];
2    y[0:n] = MyElementalFunction(x[0:n], ...);
```

The benefit of using an elemental function in these cases that the compiler may be able to automatically vectorize the loop or the expression with elemental notation, even though the source code of the function is not visible at the time of compilation. The resulting vectorized code uses SIMD operations. That means that several consecutive values from array `x[]` are loaded into a vector register, and all operations in the function `MyElementalFunction` are performed on each element of that vector register concurrently. The Intel C++ compiler can generate a short vector form of elemental function using AVX

vector operations (256-bit wide) for Intel Xeon processors and IMCI vector operations (512-bit wide) for Intel Xeon Phi coprocessors.

The vector attribute accepts clauses in the format `__attribute__((vector(clauses)))` [5] in order to clarify the pattern of the usage of the function arguments in a parallel context. The clauses of the vector attribute take the following values:

- `processor(cpuid)` requests compilation for a specific processor architecture;
- `vectorlength(n)` specifies the SIMD vector length, which should be equal to a power of 2;
- `vectorlengthfor(datatype)` provides information about the data type used (`datatype` should be one of the built-in types: `int, float, double`, etc.);
- `linear(scalar:inc)` helps the compiler to optimize consecutive increase/decrease of variable `scalar` by value `inc` at every step;
- `uniform(param)` indicates that the same value of `param` is used across the parallel context;
- `[no]mask` generates a [non-]masked vector version of the routine.

Using these clauses, the compiler may be able to optimize the executable code more efficiently.

In order to enable automatic vectorization in the library `libMyLib.a`, the error function must be declared as elemental. This must be done in the function implementation in file `MyLib.cpp`, as well as in the header file `MyLib.h`, as shown in Listings 9 and 10.

```
#ifndef _MYLIB_H_
#define _MYLIB_H_

#pragma offload_attribute(push, target(mic))
__attribute__((vector)) float MyErfElemental(const float inx);
#pragma offload_attribute(pop)

#endif
```

Listing 9: Optimized header file `MyLib.h` with the declaration of the Gauss error function implementation as an elemental function `MyErfElemental()`. Compare to Listing 4.

```
__attribute__((vector)) float MyErfElemental(const float inx){

    const float x = fabs(inx);

    const float p  = 0.3275911;
    const float t  = 1/(1+p*x);
    const float t2 = pow(t, 2);
    const float t3 = pow(t, 3);
    const float t4 = pow(t, 4);
    const float t5 = pow(t, 5);

    const float res = 1.0 - (0.254829592*t - 0.284496736*t2 + 1.421413741*t3 -
                        1.453152027*t4 + 1.061405429*t5) * exp(-x*x);

    return copysign(res, inx);
}
```

Listing 10: Optimized source code file `MyLib.cpp` uses the vector attribute to declare an elemental implementation of the Gauss error function. Compare to Listing 3.

To use the vectorized form within the user application (`Driver.cpp`), special `#pragma simd` should be applied after `#pragma omp parallel for`, indicating that the `for`-loop iterations should

be vectorized within parallel OpenMP context, as shown in Listing 11. The static OpenMP scheduling mode is the only mode currently supported for layered thread and data parallelism. For other scheduling modes, strip-mining must be applied to the loop (see, e.g., [9]).

```
#include "MyLib.h"

void RunOnHost(){
  float* fIn = ... // Allocation and initialization of data
  float* fOut = ...
#pragma omp parallel for schedule(static)
#pragma simd
    for ( int i = 0; i<lTotal; i++)
        fOut[i] = MyErfScalar(fIn[i]);
}

void RunOnCoprocessor(){
#pragma offload target(mic:0)
    {
      float* fIn = ... // Allocation and initialization of data
      float* fOut = ...
#pragma omp parallel for schedule(static)
#pragma simd
      for ( int i = 0; i<lTotal; i++)
          fOut[i] = MyErfScalar(fIn[i]);
    }
}

int main() {
  ...
}
```

Listing 11: Improved user application `Driver.cpp` calls the elemental function `MyErfElemental()` declared in `MyLib.h`. Compare to Listing 5.

When the library and the user application `Driver.cpp` are re-compiled, the vectorization report indicates successful automatic vectorization both in the host code and in the coprocessor code, as shown in Listing 12.

```
vadim@skynet% icpc -c -xAVX -vec-report3 -openmp -O3 -o MyLib.o MyLib.cpp
MyLib.cpp(4): (col. 59) remark: FUNCTION WAS VECTORIZED.
MyLib.cpp(4): (col. 59) remark: *MIC* FUNCTION WAS VECTORIZED.
vadim@skynet% xiar cru -qoffload-build libMyLib.a MyLib.o
xiar: executing 'ar'
xiar: executing 'ar'
vadim@skynet% icpc -c -xAVX -vec-report3 -openmp -O3 -o Driver.o Driver.cpp
Driver.cpp(8): (col. 5) remark: SIMD LOOP WAS VECTORIZED.
Driver.cpp(19): (col. 7) remark: SIMD LOOP WAS VECTORIZED.
Driver.cpp(19): (col. 7) remark: *MIC* SIMD LOOP WAS VECTORIZED.
Driver.cpp(19): (col. 7) remark: *MIC* PEEL LOOP WAS VECTORIZED.
Driver.cpp(19): (col. 7) remark: *MIC* REMAINDER LOOP WAS VECTORIZED.
```

Listing 12: Compiling the offload library `MyLib.cpp` with elemental functions and linking it with the application `Driver.cpp` that requests automatic loop vectorization using `#pragma simd`. Compare to Listing 8.

3 Optimization

At this point in the discussion, we have a working example of a static library `libMyLib.a` with offloadable elemental functions. As mentioned in Section 2.2, the Gauss error function implementation in Listing 3 has a number of flaws. In this section, we optimize the code of the function for better performance. We will use the following steps of optimization:

- *Consistency of the precision* of functions and constants. We ensure that all functions and numerical constants used in arithmetic expressions are in single precision.
- *Strength reduction*. Using an algebraic transformation, we express an expensive transcendental arithmetic operation (the natural base exponential) two less expensive operations (the base 2 exponential function combined with multiplication).
- *Algorithm optimization*. Using a recurrence relation, we re-write the polynomial expression used in the approximation in such a way that the calculation of the powers of the argument t becomes unnecessary. This reduces the amount of calculation that the function performs.

As a preview of our results, we show the performance of the code at several optimization steps in Figure 2. The performance is measured in the number of function evaluations per second. In order to place the results of optimization in perspective, the last set of bars in Figure 2 shows the performance of the function `erff()` from the Intel Math Library.

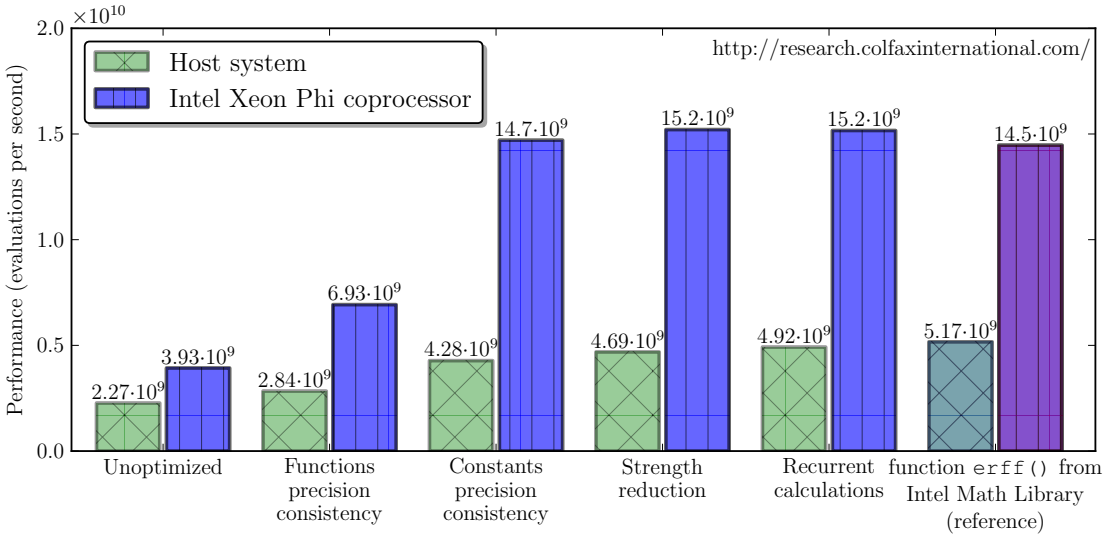

Figure 2: Performance of the library function evaluating the Gauss error function approximation. Green and blue bars show, respectively, the performance measured on the host system with two Intel Xeon E5 series processors, and on an Intel Xeon Phi coprocessor. The performance at each step of the optimization process is compared to the performance of `erff()` from the Intel Math Library. All calculations are parallelized and vectorized, and performed on an array of `n`$= 2^{29}$ single precision floating-point arguments.

3.1 Performance benchmark setup

This section describes the hardware, runtime environment and measurement methodology used for benchmarking the error function implemented in our library.

Computing System

In this paper, all tests are run on a CX2265i-XP5 server [6] with the following specifications:

- Host system: two eight-core Intel Xeon E5-2680 processors with two-way hyper-threading, 64 GB of DDR3 RAM at 1,333 MHz, running the Cent OS 6.4 Linux operating system. Intel C++ Composer XE 13.1.1 is used to compile the code;
- Coprocessor: one 60-core Intel Xeon Phi coprocessor SKU B1QS-5110P with 8 GB of GDDR5 RAM, running MPSS (MIC Platform Software Stack, the driver suite for Intel Xeon Phi coprocessors) version 2.1.5889-16. This is a pre-production sample of the coprocessor, and the results on the production version may be different.

We compare the performance of a single coprocessor to the performance of the two-socket host system, because the nominal power consumption in both platforms is roughly the same.

Environment Variables

OpenMP threads can be bound to physical processing units (processor sockets, physical or logical cores) by setting a thread affinity mask. With the Intel OpenMP library, thread affinity is controlled via the environment variable KMP_AFFINITY [7]. The KMP_AFFINITY environment variable accepts the following general syntax:

```
export KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

For all measurements, we use the compact affinity type, binding the OpenMP threads to logical cores. This type of affinity improves the performance of high arithmetic intensity applications such as our benchmark of the Gauss error function. The importance of thread affinity is demonstrated below:

```
vadim@skynet% KMP_AFFINITY="" ./runme # Affinity mask not set
Average: 0.354 +- 0.030 sec; trials 2-10 included

vadim@skynet% KMP_AFFINITY=compact,granularity=fine ./runme  # Compact affinity
Average: 0.207 +- 0.000 sec; trials 2-10 included
```

Compiler Arguments

OpenMP parallelism in the application is enabled with the compiler argument `-openmp`. In order to achieve the best performance, we use the highest optimization level set by the argument `-O3`. In addition to it, several fine-tuning flags are used as discussed below.

Applications compiled with the Intel C++ compiler can be tuned to the target hardware architecture using the `-x` flag, which tells the compiler to generate code for a specific set of vector instructions. The `-x` flag takes a mandatory option, which can be one of the following: `AVX` (*i.e.* `-xAVX`), `SSE4.2`, `SSE4.1`, `SSE3`, `SSE2`, etc. We use `-xAVX` for the host system with Intel Xeon E5 processors. Code offloaded to Intel Xeon Phi coprocessors uses IMCI vector instructions automatically.

The handling of special domains of floating-point numbers (extremes, NaNs, infinities and denormal numbers) by the functions of the Intel Math Library can be disabled [8]. This improves the performance of the exponential function used in our implementation. We use the compiler argument `-fimf-domain-exclusion=15` for that.

During the development process, in order to diagnose the success of vectorization, we used the compiler flags `-vec-report[0-6]`. This flag controls the verbosity level of the automatic vectorization report. `-vec-report0` disables diagnostics, and `-vec-report6` provides the most detailed output, reporting which loops were optimized, which loops were skipped, and why. This output can be useful to identify possible strategies to get a loop to vectorize.

Performance Measurement

The setup of the workload that we use for performance measurement is shown in Listing 11. The Gauss error function implementation is vectorized and runs in a parallel OpenMP environment on Intel

Xeon processors or on an Intel Xeon Phi coprocessor in the offload mode. For the offloaded code, the input and output arrays are initialized on the coprocessor. We do not include into the benchmark the time of offload initialization, data transfer, or memory allocation. In this sense, our results pertain to the cases when the data is already on the coprocessor.

The Gauss error function is calculated on an array of 2^{29} arguments. This number corresponds to the maximum size of two arrays `fIn[]` and `fOut[]` that can fit into the memory of the Intel Xeon Phi coprocessor that we used. Out of 10 trials for each optimization step, only the last 8 are included in the calculation of the average execution time.

The accuracy of our library function at all optimization steps was controlled by comparing the result of our implementation to the result of Intel's double precision implementation of the error function, `erf()`. Details on the accuracy of our implementation are reported in Section 3.6.

All the optimizations covered in the subsequent sections are made in the library source code file `MyLib.cpp`. The user application `Driver.cpp` is not changed between the optimization steps.

3.2 Consistency of precision: functions

Most of the common mathematical functions in the Intel Math Library (automatically linked when the header file `math.h` is included) have double precision and single precision versions. The names of the single precision versions are obtained from the double precision function names by adding the suffix `-f`. For example, `exp()` is a double precision function, which takes an argument of type `double`, returns an argument of type `double`, and maintains the accuracy required for double precision. However, `expf()` is a single precision function, which takes an argument of type `float` and returns a value of type `float`. Similarly, `sinf()`, `copysignf()`, `fabsf()` are all single precision functions.

In the Intel Math Library, single precision functions are always faster than their double precision counterparts. This aspect of programming is sometimes ignored by users familiar with the GNU family compilers. Indeed, in older versions of the GNU math library, single precision functions do not provide better performance than double precision functions, or sometimes perform even worse than the double precision functions.

Because our Gauss error function implementation relies on an approximation with an accuracy of $1.5 \cdot 10^{-7}$, single precision is sufficient for all calculations. Therefore, our implementation can benefit from using single precision functions. With this optimization, the implementation takes on the form shown in Listing 13 (only the modified part of the code is shown).

```
3     const float x = fabsf(inx); // was fabs(inx) before optimization
4
5     const float p  = 0.3275911;
6     const float t  = 1/(1+p*x);
7     const float t2 = powf(t, 2); // was pow(t, 2) before optimization
8     const float t3 = powf(t, 3); // was pow(t, 3) before optimization
9     const float t4 = powf(t, 4); // was pow(t, 4) before optimization
10    const float t5 = powf(t, 5); // was pow(t, 5) before optimization
11
12    // this expression contained exp(-x*x) before optimization
13    const float res = 1.0 - (0.254829592*t - 0.284496736*t2 + 1.421413741*t3 -
14                        1.453152027*t4 + 1.061405429*t5) * expf(-x*x);
15
16    return copysignf(res, inx); // was copysignf(res, inx)  before optimization
```

Listing 13: Error function implementation in `MyLib.cpp` optimized by using single precision functions. Compare to Listing 10.

As shown in Figure 2, this optimization increases the performance on the host by 25% and on the coprocessor by 75%.

3.3 Consistency of precision: constants

Another convention of arithmetics in the C/C++ languages that is sometimes overlooked is the default type of floating-point literal constants. By default, floating-point literal constants in C and C++ codes are assumed by the compiler to be of type `double`. For example, `0.25` is assumed to be a double precision constant. Constants that do not have a decimal point are assumed to be of type `int`. For example, `1` is assumed to be a 32-bit signed integer. Literal constants of type `float` must be marked with the suffix `-f` and contain a decimal point. Otherwise, implicit type conversion operation are used to convert the values between double precision and single precision or between integers and floating-point values.

With the above considerations, we further improve the implementation of our Gauss error function, by marking floating-point literals as single precision constants. Listing 14 demonstrates the changes.

```
5   const float p  = 0.3275911f; // was 0.3275911 before optimization
6   const float t  = 1.0f/(1.0f+p*x); // was 1/(1+p*x) before optimization
7   const float t2 = powf(t, 2.0f); // was powf(t, 2) before optimization
8   const float t3 = powf(t, 3.0f); // Note: the powf function recognizes
9   const float t4 = powf(t, 4.0f); // that the exponent is an integer
10  const float t5 = powf(t, 5.0f); // and optimizes the calculation accordingly.
11
12  // this expression did not have the suffix -f in constants before optimization
13  const float res = 1.0f - (0.254829592f*t - 0.284496736f*t2 + 1.421413741f*t3 -
14                    1.453152027f*t4 + 1.061405429f*t5) * expf(-x*x);
```

Listing 14: Error function implementation in `MyLib.cpp` further optimized by using single precision constants.

The performance gain of this optimization (see Figure 2) is additional 50% on Intel Xeon processors and 110% on Intel Xeon Phi coprocessors.

3.4 Strength reduction

In the Intel Math Library, the exponential and logarithmic functions with base 2 (`exp2f()` and `log2f()`) are faster than their natural base counterparts (`expf()` and `logf()`). The base 2 functions are optimized using the architecture specifics of floating-point number representation. If possible, the Intel Math Library may use hardware instructions for the calculation of the base 2 exponential and logarithm.

We can modify our implementation to use the base 2 exponential function instead of the natural base exponential function. This can be done by applying the following transformation:

$$e^x = 2^{x \log_2 e}, \tag{6}$$

where $\log_2 e \approx 1.442695040$ is a constant. We need a single precision value of this constant, and so the suffix `-f` is used in the code (Listing 15).

```
12  const float l2e = 1.442695040f;
13  const float res = 1.0f - ( 0.254829592f*t1 - 0.284496736f*t2 + 1.421413741f*t3 -
14                             1.453152027f*t4 + 1.061405429f*t5 ) * exp2f(-x*x*l2e);
```

Listing 15: Error function implementation in `MyLib.cpp` optimized with strength reduction: using a base 2 exponential function instead of the natural base exponential function. Compare the usage of the exponential function to Listing 14.

This optimization is known as strength reduction. Strength reduction is the replacement of a computationally expensive operation with one or more less expensive operations. In this case, we replace the natural base exponential with the base 2 exponential combined with a floating-point multiplication.

Another commonly used strength reduction operation is the replacement of division (`a = b/c`) with multiplication by the reciprocal value (`r = 1.0/c; a = b*r`). This optimization relies on the fact that floating-point multiplication in most architectures is significantly faster than floating-point division. If the reciprocal value is used multiple times, this optimization improves performance. In some cases, the compiler may implement this optimization automatically — it depends on the optimization level and the floating-point model.

Compared to the previous optimization step, strength reduction improves the performance of our Gauss error function implementation by additional 10% on the host processor and by 3% on the coprocessor. This is reflected in Figure 2.

3.5 Algorithm optimization: recurrence relation

For the final touch, we will use an optimization of the algorithm that reduces the number of arithmetic operations by using a recurrence relation.

Recurrence relations express a value in a mathematical sequence via previous values in that sequence. In our case, we have a polynomial expression in Equation (4). In order to evaluate this expression, we compute the powers of the argument `t` and then use them in the explicit evaluation of the polynomial. We can optimize this procedure by using the recurrence relation

$$a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \equiv t\left(a_1 + t\left(a_2 + t\left(a_3 + t\left(a_4 + t a_5\right)\right)\right)\right). \tag{7}$$

Comparing the expressions in the left-hand side and the right-hand side of Equation (7), it is easy to see that both of them have 9 arithmetic operations: 4 additions and 5 multiplications. However, the expression in the right-hand side only uses the values of t, and therefore, it is not necessary to compute the powers t^2, t^3, t^4 and t^5 in order to use that expression. Taking advantage of this transformation, we arrive at the last implementation of the Gauss error function approximation shown in Listing 16.

```
__attribute__((vector)) float MyErfElemental(const float inx){
    const float x = fabsf(inx);

    const float p = 0.3275911f;
    const float t = 1.0f/(1.0f+p*x);
    const float l2e = 1.442695040f; // log2f(expf(1.0f))
    const float e = exp2f(-x*x*l2e);
    float res = -1.453152027f + 1.061405429f*t;
    res = 1.421413741f + t*res;
    res =-0.284496736f + t*res;
    res = 0.254829592f + t*res;
    res *= e;
    res = 1.0f - t*res;

    return copysignf(res, inx);
}
```

Listing 16: Error function implementation in `MyLib.cpp` optimized by performing a recurrent evaluation of the polynomial expression. This is the final optimized version of the function. Compare to the unoptimized function in Listing 10.

With this optimization, the performance further increases by 5% on the host and remains unchanged on the coprocessor (see Figure 2).

The final result of our optimization produces performance very close to that of the Intel Math Library implementation `erff()`. Our implementation is some 5% slower than `erff()` on the host, and 5% faster than `erff()` on the coprocessor. The two implementations, however, produce different accuracy (see Section 3.6). The coprocessor performs our implementation 3.1x faster than two host processors.

3.6 Accuracy

We estimated the accuracy of our implementation `MyErfElemental()` and of the Intel Math Library implementation `erff()` by comparing their results to the values calculated by the double precision Intel Math Library implementation `erf()` in the range of values $-3 < x < 3$. The results are summarized in the table below.

Two accuracy metrics are reported: the maximum absolute error and the root mean square error.

Implementation	`MyErfElemental()`	`erff()`
Max. error (host)	$9.2 \cdot 10^{-7}$	$7.5 \cdot 10^{-8}$
Max. error (coprocessor)	$5.7 \cdot 10^{-7}$	$7.5 \cdot 10^{-8}$
Root mean square error (host)	$1.1 \cdot 10^{-7}$	$2.3 \cdot 10^{-8}$
Root mean square error (coprocessor)	$1.1 \cdot 10^{-7}$	$2.3 \cdot 10^{-7}$

Table 1: Accuracy of our implementation `MyErfElemental()` and of the Intel Math Library implementation `erff()` in the argument range $-3 < x < 3$.

The maximum error of our calculation is greater than the accuracy of the approximation (4) because of the accumulated numerical error and the finite precision of the exponential function `exp2f()`.

4 Summary

To fully exploit computational resources of Intel Xeon processors and Intel Xeon Phi coprocessors, applications must utilize both thread and data parallelism. For parallel applications that apply scalar functions to an array of values, these special functions can be implemented as elemental functions and packaged in the form of a statically-linked library. This approach works for applications running on common CPUs as well as for applications that offload computation to Intel Xeon Phi coprocessors. We provided a step-by-step demonstration of how such libraries can be created, and how elemental functions from a static library can be offloaded to a coprocessor and used in a thread-parallel OpenMP context layered with data-parallel SIMD processing.

4.1 Recap: producing a static offload library

The roadmap to the development of a statically-linked special function library for Intel Xeon Phi coprocessors is:

1) Write the header files and the source code files with library function and implementations;
2) Mark the declarations and implementations of functions with the offload attribute using `#pragma offload_attribute` in order to request the compilation of offload versions of the functions;
3) For functions with scalar syntax that may be used in SIMD contexts, mark them as elemental using `__attribute__((vector))` in order to compile vectorized forms of these functions;
4) Compile the source codes into object files using the Intel C++ compiler;
5) Archive the object files into a static library using the Intel tool `xiar` with the command line argument `-qoffload-build` in order to create the MIC architecture version of the library;
6) In user applications:
   a) the library header files must be included to access the functions;
   b) the library must be linked as usual with the linker argument `-l`;
   c) in data-parallel loops, `#pragma simd` should be used to employ vector versions of the elemental library functions, and
   d) `#pragma offload` can be used to offload calculations to an Intel Xeon Phi coprocessor;
7) Object files with suffixes `MIC.o` and library files with suffixes `MIC.a` contain the code offloadable to the coprocessor. These files do not have to be specified during the compilation and linking phases; however, the `lib*MIC.a` file but must be available to the users linking their applications with the library.

The optimization of functions with scalar syntax for Intel Xeon Phi coprocessors is based on the same principles as the optimization process for Intel Xeon processors. Consistency of precision must be maintained in the calls to other functions and in the specifications of constants; type conversions must be avoided. Strength reduction may be used whenever it is profitable (replacing division with multiplication by the reciprocal value and using highly optimized or hardware-supported transcendental functions). Algorithm optimizations that reduce the amount of arithmetics in a function must be considered.

When an elemental function is developed with the intent to use it in a data-parallel contexts, the best way to think of vectorization is that the operations in the function are applied independently to each SIMD vector lane. That said, an elemental function with scalar syntax must be expressed as a scalar function, and the compiler will take care of vectorization when it is requested by the user application.

4.2 High-level language or "ninja programming"?

Libraries expressed in a high-level programming language can be built for multi-core and many-core systems with a single compilation command. That said, it is not necessary to develop separately the host and the coprocessor versions of the library. Furthermore, few or no code modifications are necessary in order to port the code of the library to computer architectures of the past or of the future, because the compiler takes care of using the respective processor instructions.

On the other hand, some developers prefer to fine-tune their performance-critical functions using inline assembly or compiler intrinsics. That approach gives the developer more control in tuning the code. However, portability is out of the question for programs expressed with assembly instructions or intrinsic functions. In this approach, separate versions of the code must be developed and maintained for the host and for the coprocessor, and instruction-coded programs may not run efficiently (or at all) on future generations of processors and coprocessors.

However, when the function performance is more important than the amount of effort required for porting, the choice of the programming method boils down to the question: how good is the compiler at optimization and automatic vectorization?

The example demonstrated in this paper uses a high-level language and relies on the automatic vectorization capability of the Intel C++ compiler. Nevertheless, our library function that calculates a rational approximation for the Gauss error function performs on par with the highly optimized Intel Math Library implementation of this calculation `erff()` in terms of speed and accuracy.

Evidently, the efficiency of the Intel C++ compiler in the automatic implementation of SIMD parallelism for Intel Xeon processors and Intel Xeon Phi coprocessors is very high. With that in mind, the high-level language approach is an appealing programming methodology for the Intel Xeon family products.

References

[1] http://en.wikipedia.org/wiki/Black-Scholes

[2] http://en.wikipedia.org/wiki/Error_function

[3] Intel C++ Compiler XE 13.1 User and Reference Guide, Offload Using a Pragma. http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/hh_goto.htm#GUID-44F5B8E2-8EFD-4C51-ACF8-357900798834.htm

[4] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions*, New York: Dover, 1972.

[5] Intel C++ Compiler XE 13.1 User and Reference Guide, Elemental Functions. http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/GUID-90A7F490-941F-4C07-A88E-07BBA14AE6AF.htm

[6] Colfax CX2265i-XP5 2U Rackmount Server. http://www.colfax-intl.com/xeonphi/CX2265i-XP5.html

[7] Intel C++ Compiler XE 13.1 User and Reference Guide, Thread Affinity Interface. http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/hh_goto.htm#GUID-8BA55F4A-D5AE-4E27-8C25-058B68D280A4.htm

[8] Wendy Doerner. "Advanced Optimizations for Intel MIC Architecture, Low Precision Optimizations". http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture-low-precision-optimizations

[9] "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors". Colfax International, 2013, ISBN: 978-0-9885234-1-8. http://www.colfax-intl.com/xeonphi/book.html