Hindawi
Scientific Programming
Volume 2017, Article ID 5482468, 18 pages
https://doi.org/10.1155/2017/5482468

Research Article

Inastemp: A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization

Berenger Bramas

Max Planck Computing and Data Facility (MPCDF), Gieenbachstrae 2, 85748 Garching, Germany

Correspondence should be addressed to Berenger Bramas; berenger.bramas@mpcdf.mpg.de

Received 10 February 2017; Revised 19 June 2017; Accepted 14 August 2017; Published 20 September 2017

Academic Editor: Davide Ancona

The development of scientific applications requires highly optimized computational kernels to benefit from modern hardware. In recent years, vectorization has gained key importance in exploiting the processing capabilities of modern CPUs, whose evolution is characterized by increasing register-widths and core numbers, but stagnating clock frequencies. In particular, vectorization allows floating point operations to be performed at a higher rate than the processor's frequency. However, compilers often fail to vectorize complex codes and pure assembly/intrinsic implementations often suffer from software engineering issues, such as readability and maintainability. Moreover, it is difficult for domain scientists to write optimized code without technical support. To address these issues, we propose Inastemp, a lightweight open-source *C*++ library. Inastemp offers a solution to develop hardware-independent computational kernels for the CPU. These kernels are portable across compilers and floating point precision and vectorized targeting SSE(3,4.1,4.2), AVX(2), AVX512, or ALTIVEC/VMX instructions. Inastemp provides advanced features, such as an *if-else* statement that vectorizes branches that cannot be removed. Our performance study shows that Inastemp has the same efficiency as pure intrinsic approaches on modern architectures. As side-results, this study provides micro benchmarks on the latest HPC architectures for three different computational kernels, emphasizing comparisons between scalar and intrinsic-based codes.

1. Introduction

The development of efficient computational kernels for CPUs is useful for various scientific applications that have hot-spots that cannot be accelerated using vendor-provided libraries, such as LAPACK/BLAS. Such kernels must have an appropriate memory access pattern and use the CPUs' full capacities for achieving high performance. On the one hand, memory access tuning is usually algorithm dependent [1] and tied to the data structures. On the other hand, the CPU provides different features for computing faster, but these must be explicitly enabled. Among these features, vectorization—that is, the capacity of modern CPUs to apply a single instruction on multiple data (SIMD)—is becoming indispensable for the development of efficient kernels. While the difference between a scalar code and its vectorized equivalent was *only* of a factor of two in the year 2000 (SSE technology and double precision), the difference is now up to a factor of four on most CPUs (AVX) and up to eight (AVX512) on the latest CPU generation.

However, vectorization is troublesome to achieve because it must be done with specific assembly instructions or their intrinsic aliases. Moreover, the explicit use of vectorization ties code to the hardware and leads to issues in code factorization, maintainability, and readability. These realizations motivated the HPC community to create dedicated software and libraries to vectorize computational codes. Some of them, called autovectorization tools, help developers by converting high-level codes into vectorized source codes or executable binaries. Many compilers support this feature, but they fail to be vectorized in many cases, despite their promotion by vendors, who argue that compilers do complete optimization work. Meanwhile, more advanced autovectorization software has been designed, as in [2], where the authors use a high-level language to describe a formula that is converted into an optimized source code targeting CPUs or GPUs. However, conversion and optimization of code is an ongoing research topic. Furthermore, some algorithms can be vectorized only if they are expressed at a higher level than what these tools get as input. This is why the community has proposed another family of tools, called vectorization libraries. These libraries provide an abstraction of the hardware vector that the programmer manipulates to convert a scalar code to its vectorized equivalent. This allows the developer to focus on the algorithm, its transformation, and the data access. All the hardware specificities are managed by the library, with low-level instructions.

The challenges for the development of a vectorization library can be summarized in three points: first, there must be zero cost in performance compared to a native low-level implementation. Second, the degree of abstraction must be appropriate to the expertise of the targeted users, and the tool should provide the features needed by the targeted field. Finally, a vectorization library must be easy to use and to incorporate into an existing project. Additionally, it should support a wide range of hardware and stay up to date with new hardware, or it will simply become useless.

A possible abstraction mechanism is to use the template feature from the *C++* language to manage vectorization. The idea of using templates to provide a generic layer above intrinsic functions, as we propose in this paper, is not new and was first proposed a decade ago [3]. It has more recently appeared in dedicated tools [4–11] and in the scope of HPC projects [12, 13]. However, this approach has not become widespread because it is incompatible with the two most widely used languages in HPC, which are *C* and *Fortran*. Meanwhile, in recent years, *C++* and software engineering have grown in popularity in the HPC community [14, 15], making solutions that rely on this language more feasible. Most of the existing vectorization libraries have performance similar to pure intrinsic-based code, and their differences mainly concern design, features, and code expression. As an example, OpenVec [11] is an open-source library that provides its own data type but chooses at compile time which instruction set to use. It is a *C/C++* code that remains at low level and allows abstraction of the hardware in a simple way. Like most other tools, it is appropriate for experts, it does not target scientific programming only, and it has not been updated to the latest available hardware so far.

These considerations were the starting point of our Inastemp library, initiated while implementing Complexes-pp, a *C++* biophysics application with coarse-grained interactions (publication in preparation). We wanted to propose a system that lets scientists integrate new kernels with *automatic* vectorization. Furthermore, our objective was to use the same approach for several other projects and to factorize the knowledge and the investment using intrinsic operators. We rather propose a tool for nonexpert developers to implement computationally intensive numerical kernels and at the same time aim for a clean and easy-to-maintain software design. The prominent features of our library are a branch mechanism, CPU detection during the configuration, and various numerical operations.

The current paper is organized as follows. We describe the principles of vectorization and the main techniques to apply this optimization in Section 2. Then, we present the *intrinsics-as-template method*, Inastemp, and some related patterns in Section 3. We dedicate Section 4 to Inastemp's advanced branch features and Section 5 to a brief comparison between Inastemp and two existing libraries. Finally, we provide some examples and describe their performance on modern architectures.

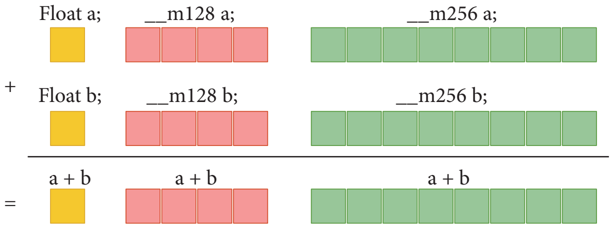


Figure 1: Summation example of single precision floating points values using (yellow square) scalar standard *C++* code, (red square) SSE vector of four values, and (green square) AVX vector of eight values.

2. Background

2.1. Common Vectorization Strategies

2.1.1. Vectorization Overview. The term vectorization refers to a CPU's feature to apply a single operation/instruction to a vector of values instead of only a single value [16]. It is common to refer to this concept by Flynn's taxonomy term, SIMD, for single instruction on multiple data. By adding SIMD instructions/registers to CPUs, it has been possible to increase the peak performance of single cores, despite the stagnation of the clock frequency. The same strategy is used on new hardware, where the length of the SIMD registers has continued to increase. In the rest of the paper, we use the term *vector* for the data type managed by the CPU in this sense. It has no relation to an expandable vector data structure, such as *std::vector*. The size of the vectors is variable and depends on both the instruction set and the type of vector element and corresponds to the size of the registers in the chip. Vector extensions to the *x86* instruction set, for example, are SSE [17], AVX [18], and AVX512 [19], which support vectors of sizes 128, 256, and 512 bits, respectively. This means that an SSE vector is able to store four single precision floating point numbers or two double precision values. Figure 1 illustrates the difference between a scalar summation and a vector summation for SSE or AVX, respectively.

Some operations are performed with one instruction with some sets, but with a combination of instructions with others. When a desired operation is performed with several instructions, this leads to different possible implementations, each having different performance. For example, there is usually no instruction to perform a reduction or horizontal/vector summation, that is, the sum of all the values contained in a vector to get a scalar. This operation is widely used, and many developers write their own instruction combination, leading to different implementations and duplicated efforts.

2.1.2. Compiler-Based Vectorization. Relying on the compilers' ability to perform optimizations offers many advantages, since it is common to start the development of a new kernel in standard *C/C++/Fortran*, first without any low-level optimizations. At this level, optimizations are limited but they leave the code readable and easy to maintain. However, it is not easy to improve a code such that it will be adequately optimized by a compiler. The programmer writes the code from his comprehension of the compiler's interpretation and optimization stages, which may not be the best way to express the original idea and may even lead to negative optimizations. Moreover, among the algorithms that solve the same problem, some are more appropriate to be implemented using vectorization such that the conversion is easier for a human [20]. To help programmers and to use their knowledge about the computation, some compilers come with their own dedicated pragmas to mark variables or loop optimizations. But some of these pragmas are not portable and still require advanced technical knowledge to be used correctly. While the latter point may be attenuated in the near future thanks to the OpenMP *SIMD* pragma [21], there is no guarantee of the degree to which a code is vectorized by a given compiler, and whether it will be optimized similarly by all compilers. Finally, in the long term, this approach is tied to the evolution of compilers and hardware, hoping that the upcoming releases of a compiler will not eventually require modifications of the code.

2.1.3. Explicit Vectorization in Assembly. Any kernel can be directly written in assembly instructions to bypass the compiler and to interact directly with the registers and the hardware. It is then possible to increase register use, to reduce the cache access accordingly, and also to control the number of instructions. However, the cost of development and maintenance of assembly code is high. It is difficult to know how such code will finally be executed by a CPU because of the complexity of modern hardware composed of units that perform prediction/speculation, out-of-order execution, and register renaming. While this method may be useful for small kernels, when they represent the costly part of an application in extreme HPC, it is difficult to apply to large functions, or when complex data structures are involved. Finally, this approach is clearly not appropriate when the objective is to have a suitable solution for numerical scientists, not experts in low-level *C*++ programming and performance optimization.

2.1.4. Explicit Vectorization with Intrinsics. Intrinsics are small low-level functions that are intended to be replaced with a single assembly instruction by the compiler. Using intrinsics allows code to be developed at the level of the programming language—*C*++ in our case—while being able to tell the compiler to use particular instructions. Therefore, the compiler still has room for optimizations and transformations to convert an intrinsic-based code into a binary. The compiler decides how to store an intrinsic vector variable, which may be in a register or in memory. Additionally, intrinsic-based codes are much easier to write, to maintain, and to debug than assembly code and can still achieve comparable performance, as will be shown below. However, each instruction set has its own intrinsics, which means that, for example, an SSE code has to be rewritten completely to be transformed into AVX. A conversion from one set to the other is usually easy to perform with a word-by-word translation but leads to code duplication and maintenance issues, in addition to closely tying the algorithm to a specific hardware. Moreover, developers who are not experts in low-level optimization often struggle to use intrinsics.

2.1.5. Branch Vectorization. A wide range of scientific computational kernels are naturally written with conditional branches, such as *if-else* statements. In our context, a branch describes the need to apply distinct operations to different values in a single vector. Current CPU hardware does not provide dedicated instructions to manage branches, mainly for two reasons. First, the cost of the instructions cannot be reduced by explicitly skipping some values, that is, asking the CPU to work only on a subpart of a vector. Second, exchanging values between vectors and building a vector from noncontiguous values in memory are both expensive operations.

A possible implementation of a *vectorized branch* is to compute all of the branches separately and to merge the different results. The merge can be done by multiplying the results by *0* or *1*, but it is also possible to perform binary operations with a mask to select the correct values. The binary-based approach is generally faster than the arithmetic technique because it has a lower latency. The resulting behavior is similar to the one on GPUs, where different execution paths of the threads from a warp are computed separately. Inastemp provides several mechanisms to help the developer in converting scalar code with branches into a fully vectorized binary.

2.2. C++ Template Programming. Template code is parameterized with types. It is generic: the same code can be instantiated with many different data types, as long as those types conform to some required API. When the compiler encounters an instantiation of a template with a particular type, it replaces the generic parameterized code with code specific to that type. A template argument type can be a native type, a class/struct, or even a SIMD-intrinsic vector. Therefore, writing a template function offers important benefits in terms of factorization and maintainability without degrading performance. While this method increases the compilation workload, this is usually not an issue when dealing with scientific HPC applications where the execution time is what really matters. However, it is sometimes difficult to understand or to find the origin of compilation errors produced by template codes.

3. Introducing Inastemp, the Intrinsics-as-Template Library

3.1. Object-Oriented (OO) Design. Inastemp is based on a pure object-oriented design, with one C++ class per instruction set (SSE3, SSSE3, SSE4, etc.). Table 1 provides the existing classes and the underlying intrinsic data type for each of them. An alternative design would be to have only one class per intrinsic data type/vector size. However, this would limit the number of classes and require macros inside the code to select the appropriate versions/implementations for

Table 1: Inastemp vector classes.

Class name	Accuracy	Intrinsic type	Possible hardware
InaVecSCALAR	*float,double*	—	All
InaVecSSE3	*float,double*	__m128(d)	Intel Pentium Dual-Core, AMD Athlon 64
InaVecSSSE3	*float,double*	__m128(d)	Intel Atom, AMD Bobcat
InaVecSSE4	*float,double*	__m128(d)	Intel Silvermont, AMD Barcelona
InaVecAVX	*float,double*	__m256(d)	Intel Sandy Bridge, AMD Bulldozer
InaVecAVX2	*float,double*	__m256(d)	Intel Haswell, AMD Carrizo
InaVecAVX512COMMON	*float,double*	__m512(d)	Intel Skylake
InaVecAVX512KNL	*float,double*	__m512(d)	Intel Knights Landing
InaVecALTIVEC	*float,double*	__vector (float | double)	IBM Power-8
InaVecBestType	*float,double*	*Compilation dependent*	*Compilation dependent*

```
(1) InaVecSSE4<double> a_vec, another_vec;      // Build without initialization
(2) InaVecSSE4<double> a_third_vec(a_vec);  // Copy constructor
(3) a_vec = another_vec;  // Copy operator
```

Code 1

each method, depending on the available instruction sets. In contrast, our approach is highly flexible because it allows an unlimited number of classes. Moreover, we have full control over the loads/stores/conversions, methods, and operators connected to each class, independently of the native intrinsic data type. Inside the library, we factorize some codes and functionalities between the classes, mainly using class inheritance. For instance, the class *InaVecAVX2* inherits from *InaVecAVX* since they use the same intrinsic type and AVX2 is an extension of AVX. By doing this, we can specialize some of the *InaVecAVX2* methods that can be improved with AVX2 instructions or otherwise use the methods from *InaVecAVX*.

Constructors and Load/Store Operators. The way scalars or pointers are converted into vector classes forms the layer between the user code and the Inastemp kernels. Therefore, the conversions are important to achieve compact and natural expressions within code. All the Inastemp vector classes have the following constructors:

(i) Inastemp classes have an empty constructor (line (1)), a copy constructor (line (2)), and a copy operator (line (3)) (see Code 1).

(ii) A scalar can be converted into a vector having all the values in the vector equal to it. We deliberately enable implicit constructor (line (1)) and assignment operator (line (2)) because it makes it possible to put scalars and numbers directly inside an expression (see Code 2).

(iii) Values from a pointer can be loaded with an explicit constructor (line (2)) or with the *setFromArray* method. If the array's address is correctly aligned, performance can be improved by loading using the *setFromAlignedArray* method (line (4)) (see Code 3).

(iv) Constructor from *initializer_list* is supported (see Code 4).

(v) The vector can be stored in memory using the *storeInArray* and *storeInAlignedArray* methods. See lines (3) and (5) (see Code 5).

These methods make it possible to write a code as shown in Code 6, where we have two temporary objects at line (3), one from the conversion of 4.5 and the other from the loading of the input array.

Returning a New Object instead of Updating the Method's Target. Intrinsic operators do not modify the vectors given as input parameters but return the result as a new vector (all operations can be seen as passed/returned by value). In reality, some operations do modify input registers, but this is invisible at the intrinsic's level. We followed this convention and made the choice to return a new object containing the result of the desired operation instead of modifying the object itself. In Code 7, a new object is returned when we call the *sqrt* method at line (3), and a scalar is returned when we call the reduction/horizontal sum at line (5).

3.2. Template Functions for Inastemp Vector Classes.

All the Inastemp classes have the same interface; they provide the same methods and operators. Therefore, it is possible to write template functions where the template type parameter can be replaced by any Inastemp class. These functions are then hardware-independent algorithms. Code 8 shows a scalar product and two possible compilations of this function, for AVX2 at line (7) and for AVX512KNL at line (8). The fact that the intrinsic vectors, chosen at compile time, have different sizes could lead to different algorithms. Therefore, one should carefully manage the transfer from scalars to vectors so that the functions remain correct for vectors of any size.

Selecting the Best Possible Vector Class for a Specific Hardware. During the configuration stage, Inastemp checks for the compiler and the hardware capabilities. By default, all the

```
InaVecSSE4<double> real_to_vec = 9.0; // Equivalent to: InaVecSSE4<double> ... real_to_vec(9.0);
real_to_vec = 11.0; // Equivalent to: real_to_vec.setFromScalar(11.0);
```

Code 2

```
double* ptr = new double[InaVecSSE4<double>::VecLength];
InaVecSSE4<double> ptr_to_vec(ptr);       // same as .setFromArray() method
alignas(InaVecSSE4<double>::Alignment) align_array[InaVecSSE4<double>::VecLength];
ptr_to_vec.setFromAlignedArray(align_array);    // load from aligned array
```

Code 3

```
InaVecSSE4<double> ilist_to_vec {{1, 2}};
```

Code 4

instruction sets supported by both the compiler and the CPU are enabled. Users can also activate any vector/class supported by the compiler, even if it is not supported by the hardware. This stage creates different macros in the main Inastemp header and adds shortcuts to the best available class. In Code 9, we show how a function can be templatized to the best available set at line (11).

3.3. Abstraction Overhead. Inastemp, like other vectorization libraries, suffers from a potential issue from its abstraction mechanism: it may generate codes that are not fully optimized. This happens when a sequence of operations must be written differently to be fully optimized, compared to what the library generates from successive high-level calls. This is especially true when there are several calls to functions that work on vectors smaller than the hardware registers, which happens when using a SSE vector on a CPU that supports AVX, for example. In such cases, an optimized version of multiple calls would merge the small vectors together to fully exploit the hardware. These optimized implementations must be done by experts at the library level. Consequently, even if we attempt to propose the more common functions for the implementation of scientific kernels, some users will need to ask for new features.

4. Advanced Features of Inastemp

4.1. Branch Management. In this section, we describe how to compare/test Inastemp vectors using methods or operators. Then, we explain how we use these comparison results in our branch mechanisms.

4.1.1. Testing Numerical Vectors. In scalar programming, the result from a test is Boolean, whereas, in vector programming, it is a vector of Booleans with as many values as there are in the tested numerical vectors. Inastemp provides an abstraction over this pseudo-Boolean vector, which is either a vector of integers, composed of $0xF\dots F/0$ for true/false, or a single integer for some instruction sets, where a single bit corresponds to one test result. The tests and comparisons on numerical vectors can be done using static member functions or operators, as shown in Code 10, where we perform three different tests. First, we test if the values included in a vector are positive at lines (6) and (7). Then, we compare two vectors at lines (10) and (11). Finally, we test if two vectors are different at lines (15) and (16).

Inastemp also provides static methods to obtain floating point vectors, composed of *1./0.* (true/false), instead of pseudo-Boolean vectors. While they prove to be useful in specific cases, they are left out of the current study, because they are less efficient than the pseudo-Boolean vectors to manage branches.

4.1.2. Conditional Static Functions. We provide three static methods to merge floating point vectors, based on pseudo-Boolean vectors: *IfTrue(b, v)*, *IfFalse(b, v)*, and *IfElse(b, v_true, v_false)*, where *b* is a Boolean vector and *v/v_true/v_false* are numerical vectors. *IfTrue(b, v)* returns a numerical vector where the value at position *i* is equal to *v[i]* if *b[i]* is true, or 0 otherwise. *IfFalse(b, v)* is equivalent to *IfTrue(b.not(), v)*, and *IfElse(b, v_true, v_false)* allows selecting values from *v_true* or *v_false* if the Booleans in *b* are true or false, respectively. Using these methods, we can rewrite any branch-based scalar code following the conversion in Table 2. This approach computes all of the branches even if the Boolean vector is filled only with true or false.

Code 11 presents examples for the three methods. At line (5), we fill vector *res* with 10, where the condition is true, or with 20 otherwise. We increment *res* by twice the current values, at positions that are not equal to *a_value* at line (9). At lines (14), we set to zero the positions in *res* that are different from 60, we explicitly use a precomputed Boolean vector at line (19), and we increment some of the positions of *a_value* at line (22).

4.1.3. Advanced Branch System with C++ Lambda/Anonymous Functions. Using the method described above allows for the vectorization of arbitrary expressions. However, the conversion of scalar code is not straightforward, especially when it contains a lot of branches. Inastemp provides templates

```
InaVecSSE4<double> a_value = 10;
double* ptr = new double[InaVecSSE4<double>::VecLength];
a_value.storeInArray(ptr);       //
alignas(InaVecSSE4<double>::Alignment) align_array[InaVecSSE4<double>::VecLength];
a_value.storeInAlignedArray(align_array);      //
```

Code 5

```
void aFunction(const double* input, double* output){ // ptr a pointer not aligned
      InaVecSSE4<double> sse4_9 = 9.0;
      const InaVecSSE4<double> result = 4.5 + sse4_9 * InaVecSSE4<double>(input);
      result.storeInArray(output);
}
```

Code 6: Inastemp example using the SSE4 Inastemp class (not generic).

```
InaVecAVX2<double> a_avx2_value = 100.0;
// The call does not modify a_avx2_value
InaVecAVX2<double> another_avx2_value = a_avx2_value.sqrt();
// The call does not modify a_avx2_another_value
double sum = another_avx2_value.horizontalSum();
```

Code 7: Inastemp calling method example using the AVX2 Inastemp class.

```
template <class VecType>
double ScalarProduct(const VecType& v1, const VecType& v2){
      return (v1 * v2).horizontalSum();
}

// Possible usage
double res_1 = ScalarProduct<InaVecAVX2<double>>(....);
double res_1 = ScalarProduct<InaVecAVX512KNL<double>>(....);
```

Code 8: Templatized function example.

```
template <class VecType>
void user_kernel(/* parameters */){
      /* code */
}

// This header contains macros about instruction sets available
#include <InastempConfig.h>

void user_function(/* parameters */){
      // InaVecBestType<double> is equivalent to InaVecBestTypeDouble
      user_kernel<InaVecBestTypeDouble>(/* ... */ );
}
```

Code 9: Automatic selection of the best available instruction set.

```
// Considering InaVecBestType can contain 4 doubles
InaVecBestType<double> a_value = {{100.0, 50.3, −80.8, 0.3}};
InaVecBestType<double> another_value = {{−1, 2, −3, 4}};

// [true, true, false, true]
InaVecBestType<double>::InaVecMask a_is_positive_mask = a_value.isPositiveMask();
InaVecBestType<double>::MaskType a_is_positive_mask_op = a_value < 0.;

// [false, false, true, true]
InaVecBestType<double>::InaVecMask a_lower_than_another_mask = . . .
        InaVecBestType<double>::IsLowerMask(a_value, another_value);
InaVecBestType<double>::MaskType a_lower_than_another_mask_op = a_value < another_value;


// [true, true, true, true]
InaVecBestType<double>::MaskType a_lower_than_another_mask = . . .
        InaVecBestType<double>::IsNoEqualMask(a_value, another_value)
InaVecBestType<double>::MaskType a_lower_than_another_mask_op = a_value != another_value;
```

Code 10: Inastemp comparison-vector.

```
// Considering InaVecBestType can contain 4 doubles
InaVecBestType<double> a_value = {{11.0, 8.0, 7.0, 20.0}};

// Return 10 where condition is true (a_value < 10) else return false
InaVecBestType<double> res = InaVecBestType<double>::IfElse(a_value < 10, 10., 20.);
// res => {{20.0, 10.0, 10.0, 20.0}}

// Return res     2 where condition is false (a_value == res) else return 0
res += InaVecBestType<double>::IfFalse(a_value == res, res ∗ 2);
// res => {{20.0, 10.0, 10.0, 20.0}} + {{2     20.0, 2     10.0, 2     10.0, 0}}
// res => {{60.0, 30.0, 30.0, 20.0}}

// Return res where condition is true (res != 10) else return 0
res = InaVecBestType<double>::IfTrue(res != 60, res);
// res => {{0, 30.0, 30.0, 20.0}}

// Equivalent to: if(0 < a_value) a_value += 10;
InaVecBestType<double>::MaskType a_is_positive_mask = 0 < a_value;
a_value += InaVecBestType<double>::IfTrue(a_is_positive_mask, 10.0);

// Equivalent to: if(a_value < another_value) a_value += another_value − 1
a_value += InaVecBestType<double>::IfTrue(a_value < another_value, (another_value − 1.));
```

Code 11: Inastemp static condition statements.

for chaining branching commands, so that code can be structured similarly to the standard if-then-else statements. The mechanism allows for complex branches as long as they respect the following conditional structure:

(i) *If(b)* must be followed by *Then(v)*.

(ii) *Then(v)* can only be followed by an *Else(v)* or *ElseIf(b)*.

(iii) *ElseIf(b)* is similar to *If(b)*.

(iv) *Else(v)* ends the test and cannot be followed by another statement.

v can be a vector or an expression that returns a vector, including a lambda/anonymous function. The compiler is encouraged to inline lambda functions (the C++ standard [22] requires that lambda functions are considered similarly to functions that the programmer has marked with the "inline" keyword). Code 12 shows an example where we pass vectors, instructions, or lambda functions to the test system.

4.1.4. Testing Boolean Vectors. The pseudo-Boolean vectors are real *C*++ classes that provide methods and operators, like binary operations, but also comparison methods that return a standard *C*++ Boolean. One of these methods tests if all the

Table 2: Pseudocode to convert a branch-based scalar algorithm to a vectorized code using simple conditional methods. Here A and B can be functions or a complete scope of instructions computed before the merge.

Scalar	Using simple conditional methods

if(test){
res = A();
}
else {
res = B();
}
	res_A = A;
	res_B = B;
	res = IfElse(test, res_A, res_B);
	
	// or if A and B are functions
	res = IfElse(test, A(), B());

if(test){
res_0 = A();
}
else {
res_1 = B();
}
// then use res_0 and res_1
	res_0 = IfTrue(test, A());
	res_1 = IfFalse(test, B());

values inside the Boolean vector are either all true or all false. The result is potentially useful to improve the performance by avoiding the computation of branches that are not needed. On the other hand, this technique also adds some penalty, by disturbing the instruction pipelining and prediction.

In Code 13 there are two methods that compute exactly the same things: the natural exponentiation of the lowest values between *v1* and *v2*, minus the greatest value between *v1* and *v2*. Function *LowestExp* always computes both *(v1–v2).exp()* and *(v2–v1).exp()* and merges the result using a test. In *LowestExpWithTest*, we first look at the test result to see if it is true for all values. In the case it is, we only compute *(v1–v2).exp()*—dividing the total cost by a factor of two—and otherwise we do as in *LowestExp*.

4.1.5. Branch Performance Model. In this section, we describe a simple model to estimate the potential benefit of vectorization. Without branches, a computational kernel of C_{scalar} instructions in scalar can be transformed into $C_{\text{vec}} = C_{\text{scalar}}/V$ vectorized instructions, where V is the SIMD-width (the number of values inside a vector). Using scalar, the cost of a kernel with branches can be decomposed as $C_{\text{scalar}} = C_{\text{core}} + C^{s}_{\text{branch}}$, where C_{core} is the cost of the branch-independent section and C^{s}_{branch} the cost of the selected branch. Vectorization allows the replacement of V calls to a scalar kernel by one call to its vectorized equivalent. Therefore, we can rewrite the formula to $C'_{\text{scalar}} = C_{\text{core}} + C'_{\text{branch}}$, where *prime* means in average: C'_{branch} is the average cost of all V selected branches. When the same branch is always selected for all calls, or if all branches have a similar cost, then we have $C_{\text{scalar}} = C'_{\text{scalar}}$.

We can bound the cost of the vectorized kernel with branches by

$$C_{\text{vec}} = \frac{\left(C_{\text{core}} + C_{\text{all-branches}}\right)}{V}, \tag{1}$$

where

$$C_{\text{all-branches}} = \sum_{i=1}^{B} C^{i}_{\text{branch}}. \tag{2}$$

In (2), B is the number of branches in the kernel, and $C_{\text{all-branches}}$ is the cumulated cost of all branches.

To expect an improvement, the extra cost from the computation of all branches must be less than the speedup from the vectorization:

$$\begin{gathered} C_{\text{vec}} < C'_{\text{scalar}} \Longrightarrow \\ C_{\text{all-branches}} < (V-1)\,C_{\text{core}} + VC'_{\text{branch}}. \end{gathered} \tag{3}$$

If the core part is dominant, and the branch section is insignificant, subsequently the criterion is true, and we have $C'_{\text{vec}} \approx C_{\text{core}}/V$. If the branches are the dominant part, then we must have $C_{\text{branches}} < VC'_{\text{branch}}$ to expect any benefit. This criterion will not be true when $V < B$, that is if there are more branches than values inside the vector, but also if the average cost of the selected branch is lower than the average cost of all branch $C'_{\text{branch}} < C_{\text{branches}}/B$. Moreover, as the size of the vectors continues to increase with the new hardware, this beneficial limit also increases. Nevertheless, one can estimate the maximum potential gain for using vectorization in its own kernel with the formula of the theoretical speedup given by

$$S = \frac{C'_{\text{scalar}}}{C'_{\text{vec}}}. \tag{4}$$

4.2. Miscellaneous Features. Inastemp provides a set of operations to satisfy the needs of common computational kernels.

Horizontal Summation/Multiplication. The need for summing or multiplying all of the elements of a vector into a resulting scalar is widespread. We provide an efficient approach for the different instruction sets or redirection to a dedicated hardware instruction if it exists.

Natural Exponential Function. We provide an implementation of the natural exponential function from [23]. Inastemp uses the dedicated exponential operators from the instruction sets or the compilers when they exist. Computing with a vectorized exponential might not be faster than with a scalar one, but it avoids unpacking each value and calling the standard exponential on individual values before packing the vector again.

Fast Power Operation. We provide an implementation of a fast power with an integer coefficient. It performs as many multiplications as the number of bits set in the coefficient, which is only one more than the exponentiation-by-squaring algorithm.

Flop Counters. It is possible to count the number of floating point operations (Flop) done by an Inastemp class as shown in Code 14. The Flop counter class catches every arithmetic operation and offers several methods for retrieving the different counters. The overhead is very low, since we just

```
// Considering InaVecBestType can contain 4 doubles
InaVecBestType<double> a_value = {{11.0, 8.0, 7.0, 20.0}};
InaVecBestType<double> another_value = {{1.0, 0.0, 2.0, 3.0}};

// Example with multiple conditions
a_value *= InaVecBestType<double>::If(a_value * 10. < another_value)
                    .Then(another_value)
                    .ElseIf(a_value < 10.).Then(a_value*10)
                    .ElseIf(a_value == 0).Then([&](){
                        return another_value.sqrt();
                    })
                    .Else([&](){
                        InaVecBestType<double> velse = {{1.0, 2.0, 3.0, 4.0}};
                        return velse * 100 + 40;
                    });
```

Code 12: Inastemp advanced condition manager.

```
InaVecBestType<double> aKernel(const InaVecBestType<double> v1, const InaVecBestType<double> v2){
    // Same as:
    // return InaVecBestType<double>::IfElse(v1 < v2, (v1 − v2).exp(), (v2 − v1).exp());
    // Both (v1−v2).exp() and (v2−v1).exp() are computed
    return InaVecBestType<double>::If(v1 < v2)
                                .Then((v1−v2).exp())
                                .Else((v2−v1).exp());
}

InaVecBestType<double> aKernelWithTest(const InaVecBestType<double> v1, const . . .
      InaVecBestType<double> v2){
      const InaVecBestType<double>::MaskType v1lowerv2 = (v1 < v2);
      // Similar to v1lowerv2 == InaVecBestType<double>::MaskType(true)
      if( v1lowerv2.isAllTrue() ){
          return (v1−v2).exp();
      }
      return InaVecBestType<double>::If(v1lowerv2)
                                .Then((v1−v2).exp())
                                .Else((v2−v1).exp());
}
```

Code 13: Inastemp testing the Boolean vector to avoid computing all branches.

increment integer values while performing floating point operations. This functionality allows us to have a portable performance metric to profile a code or to compare hardware. This tool is currently not thread-safe, and it does not count Flops when calling *sqrt* and *rsqrt*. For these two operations, we return the number of values that have been processed, without giving an estimation of the number of Flops, since it is a single hardware instruction in most cases.

5. A Brief Comparison of Inastemp and Some Related Work

In Table 3, we compare different aspects of Inastemp and two well-known libraries, OpenVec [11] and VCL [10]. All three packages are written on top of intrinsics and intend to separate the algorithms from their vectorization. However, they also have important differences in terms of design and potential users.

The OpenVec library is extremely lightweight because its interface is a redirection to the real intrinsics using macro preprocessing. This approach is appropriate for small projects, but it suffers from a static design and lack of control due to global definitions. For example, the instruction sets SSE3, SSSE3, SSE41, and SSE42 use the same vector data type *(__m128(d))*, and thus functions for these different sets have the same prototype. Moreover, it is not easy to maintain or to use because the complexity grows as we use more and more instruction sets. Therefore, OpenVec is intended for programmers who are used to intrinsics and who seek a minimal abstraction mechanism. Finally, this library does not support ALTIVEC/VMX, nor does it provide any help in the compilation process.

```
template <VecType>
void user_function( /* ... */ ){

}

// To compute in release mode
user_function<InaVecBestType<double>>(/* ... */);

// To record the number of Flops
user_function<InaVecFLOPS<InaVecBestType<double>>>(/* ... */);

// Flops counters
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getMulOp();
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getDivOp();
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getAddOp();
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getSubOp();
// For Sqrt and Rsqrt it will be the number of calls times the length of the vector
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getRsqrt();
InaVecFLOPS<InaVecBestType<double>>::GetFlopsStats().getSqrt();
```

Code 14: Inastemp Flops counters.

Table 3: Brief comparison of OpenVec, VCL, and Inastemp.

	OpenVec [11]	VCL [10]	Inastemp
Language	*C*	*C++*	*C++*
Use a modern repository	Yes (GitHub)	No	Yes (GitLab)
Use CI	Unit test	Not public	Yes
Latest update	2015	May 2017	June 2017
Support KNL	Not optimized	Yes	Yes
Support ALTIVEC/VMX	No	No	Yes
Design	Macro interface	OO with Macro	Pure OO (C++11)
Target users	Programmers used to intrinsics	Experienced C++ programmers with intermediate knowledge in vectorization	Experienced C++ programmers
Conditional system	Yes (basic)	Yes (select function)	Yes (advanced)
Used by real applications	Unknown (no citations)	Yes	Yes (Internally)
Incorporation	Header inclusion	Header inclusion	CMake subproject or installation
CPU detections	None	None	Yes (CPUID)

VCL is a popular library that uses more abstraction than OpenVec and is intended to be used in all kinds of programs. It offers classes for many different types of vectors (including signed/unsigned integer/short vectors) and is well furnished in terms of high-level functions. The library mixes both OO and procedural programming and uses one class per intrinsic data type, such that macros are needed inside the classes to apply specific instruction sets' optimizations. This not only makes the code more complicated, but also prohibits production of a generic binary where different instruction sets of the same family are embedded (similarly to feature-specific autodispatch code paths from the Intel compiler).

Inastemp is intended to be more specific and easier to use for numerical scientists, not experts in low-level programming. Its high level of abstraction limits the possible misuse of vectorization. Since Inastemp has been developed using scientists' feedback in a real application, it does not cover all needs, and future users might require new features. However, this approach is also a way to keep the library compact.

6. Performance Study

6.1. Hardware/Software Configurations. To study the performance, flexibility, and robustness of our library, we target different systems and compilers from a personal computer to various HPC platforms. The version of Inastemp is *0.2.2* (the package is freely available at https://gitlab.mpcdf.mpg.de/bbramas/inastemp under the MIT Lisence).

I3-PC. It is equipped with an Intel Core i3-4160 CPU at *3.60 GHz*. We select and test the SSE41 and AVX instruction sets. We use the compilers GCC version 6.2.0 and Clang 3.5 and thus will refer to these combinations as GCC-I3-PC and Clang-I3-PC in the following.

IX-HPC. It is equipped with an Intel Xeon CPU E5-2698 v3 at *2.30 GHz*. We select and test the SSE41 and AVX instruction sets but passed the flag to turn on the AVX2 instructions set in order to enable fused multiply-add (FMA). We use the compilers GCC 6.2.0 and Intel 17.0.1 and thus will refer to these combinations as GCC-IX-HPC and Intel-IX-HPC in the following.

P8-OP. It is equipped with an IBM Power-8-NVL (OpenPower) at *4.023 GHz*. We select and test the ALTIVEC/VMX [24] instruction set, which is the IBM SIMD technology. We use the compilers GCC 6.2.0 and IBM XL C++ V13.1.5 (Community Edition) and thus will refer to these combinations as GCC-P8-OP and Xl-P8-OP in the following.

Intel-KNL. It is equipped with an Intel Knights Landing CPU Xeon Phi 7210 at *1.30 GHz*. We select and test the SSE41, AVX, and AVX512(KNL) instruction sets. We use the compilers GCC 6.2.0 and Intel 17.0.0 and thus will refer to these combinations as GCC-KNL and Intel-KNL in the following. The CPU is configured in flat mode, and all tests use the on-chip high bandwidth memory exclusively.

6.2. Objectives. The primary objective of the performance study is to evaluate whether using Inastemp achieves the same performance as an explicit intrinsic-based code. Therefore, we test different instruction sets on each hardware, not only the best available one. However, we do not intend to analyze the efficiency of the tested algorithms, the benefit of one instruction set over another, or the different CPU capabilities which are out of the scope of the current paper. We test a scalar algorithm written in simple *C/C++* that can be, in principle, fully vectorized by the compiler. In addition, we test a single template function using various Inastemp classes for two accuracies, such that the results labeled *Inastemp SSE* and *Inastemp AVX* rely on the same function but with different classes as templates. The intrinsic-based kernels are written using native intrinsic code, which is nothing more than a manual inlining of the template function. All executions are bound to a single core using *numactl* with the options *–physcpubind=X –localalloc* (except for the KNL where we use *membind=1*). Aggressive compilation flags are used for all tests: *-O3 -march=native*.

6.3. Particle-Particle Interactions (with Branch) Test Case. In Figure 2, we provide the performance results for a custom particle-interaction kernel (this source code is available inside the Inastemp package, in the file: *Examples/ParticlesWithBranch/main.cpp*). The algorithm computes the interaction between N particles with a double loop and uses a conditional statement to apply a coefficient depending on the distance between particles. The potential to be computed between two particles i and j is given by

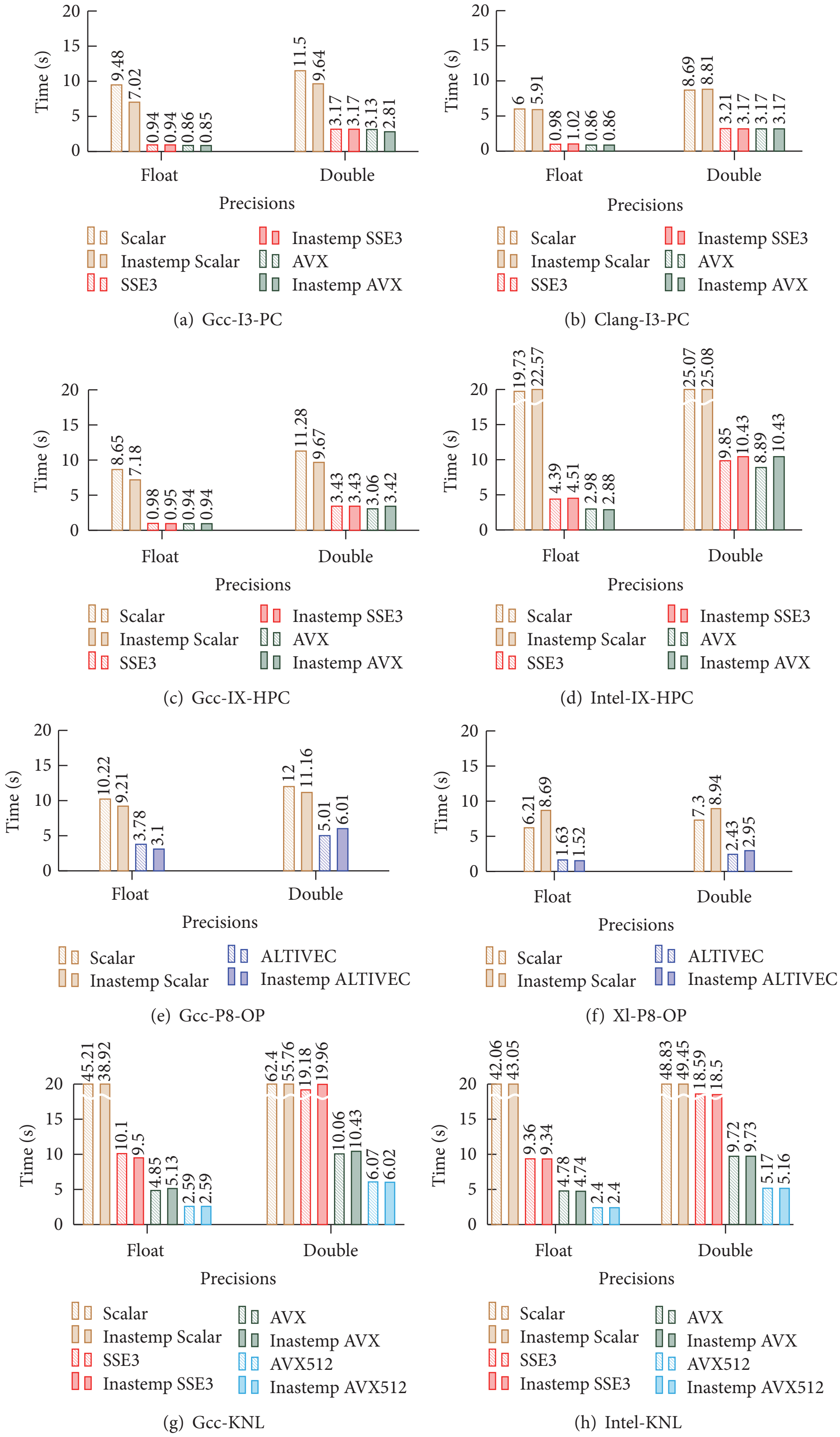
$$P_{i,j} = \frac{V_j + C}{r_{i,j}}, \qquad \text{with } i \neq j, \text{ where } C = 0 \text{ for } r < R,\ C \neq 0 \text{ for } r \geq R, \tag{5}$$

where V_j is the physical value of the particle j, $r_{i,j}$ is the distance between particles i and j, and R can be seen as a cutoff distance as it frequently occurs in molecular dynamics codes. This statement has been vectorized using our branch mechanism, and thus the kernel does not contain any scalar *if/else* and is fully vectorized as illustrated by the code snippet in Code 15.

Figure 2 shows that Inastemp delivers the same performance as pure intrinsic-based kernels in all cases and with all compilers. We use only a single template function for all the Inastemp-based execution, whereas we need one kernel for each configuration to use native intrinsics (instruction set × accuracy different implementations). Therefore, Inastemp allows replacing more than 600 lines of code by less than a hundred.

Comparing the results for different architectures shows that the I3-PC provides the best execution time; see Figures 2(a) and 2(b). The GCC compiler delivers better performance than Intel compiler on the IX-HPC for both accuracies and all instruction sets (including scalar); see Figures 2(c) and 2(d). On the KNL both compilers show similar performance; see Figures 2(g) and 2(h). The performance on the KNL and the Power-8 (Figure 2(e)) are not as good as the Intel classic CPUs for the scalar or Inastemp-based codes. In fact, the VMX instruction set on the Power-8 is similar to SSE with a vector size of 128 bits, but the Power-8 CPU has a high frequency such that we expected better results. For the KNL, the frequency is lower than a classic Intel CPU and the AVX512 vector contains twice the number of values than an AVX vector. The Inastemp interface and extra layer do not add significant overhead because we observe that the pure intrinsic-based kernel is performing similarly. Finally, our performance model, introduced in Section 4.1.5, shows its limits because the speedup of using AVX instead of SSE is far from two in most cases except KNL. In fact, while AVX instructions process twice the number of values compared to SSE, they can have a higher cost in terms of CPU-cycles.

6.4. Natural Exponential Test Case. Figure 3 shows the time to compute a natural exponential (this source code is available inside the Inastemp package, in the file: *Examples/Exp/main.cpp*). We have only a single Inastemp template kernel that calls the *exp* function provided by our library, whereas we implemented several intrinsic-based kernels for different instruction sets. The results show that both approaches have similar performance. However, we experienced important differences between compilers, especially on the KNL; see Figures 3(g) and 3(h). We can also notice that the Intel compiler has been able to catch our call to the scalar *exp* and to transform it into an optimized one. On conventional IX-HPC hardware (see Figures 3(c) and 3(d)), the usage of AVX does not provide any significant speedup compared to SSE implementations. On IBM Power-8 the performance makes this architecture competitive with the others (Figure 3(e)). Finally, Inastemp appears to be as efficient as native intrinsics except for the combination of SSE and GCC. This is not a real limitation since the Intel hardware should be used with AVX here, but it is a good illustration of the important role of the compilers in the optimization stage.



(a) Gcc-I3-PC

(b) Clang-I3-PC

(c) Gcc-IX-HPC

(d) Intel-IX-HPC

(e) Gcc-P8-OP

(f) Xl-P8-OP

(g) Gcc-KNL

(h) Intel-KNL

Figure 2: Execution time in seconds, for particle-particle interactions between $N = 40000$ particles (N^2 total interactions).

```
// Scalar interaction
const RealType dx = positionsX[idxTarget] − positionsX[idxSource];
const RealType dy = positionsY[idxTarget] − positionsY[idxSource];
const RealType dz = positionsZ[idxTarget] − positionsZ[idxSource];

const RealType distance = std::sqrt(dx∗dx + dy∗dy + dz∗dz);
const RealType inv_distance = 1/distance;

if(distance < cutDistance){
    potentials[idxTarget] += ( inv_distance ∗ physicalValues[idxSource] );
    potentials[idxSource] += ( inv_distance ∗ physicalValues[idxTarget] );
}
else{
    potentials[idxTarget] += ( inv_distance ∗ (physicalValues[idxSource]−constantIfCut) );
    potentials[idxSource] += ( inv_distance ∗ (physicalValues[idxTarget]−constantIfCut) );
}

// Vectorized using advanced branch manager
const VecType dx = targetX − VecType(&positionsX[idxSource]);
const VecType dy = targetY − VecType(&positionsY[idxSource]);
const VecType dz = targetZ − VecType(&positionsZ[idxSource]);

const VecType distance = VecType(dx∗dx + dy∗dy + dz∗dz).sqrt();
const VecType inv_distance = VecOne/distance;

const typename VecType::MaskType testRes = (distance < VecCutDistance);

const VecType sourcesPhysicalValue = VecType(&physicalValues[idxSource]);

targetPotential += inv_distance ∗ VecType::If(testRes).Then([&](){
      return sourcesPhysicalValue;
}).Else([&](){
    return sourcesPhysicalValue−VecConstantIfCut;
});
const VecType resSource = inv_distance ∗ VecType::If(testRes).Then([&](){
      return targetPhysicalValue;
}).Else([&](){
    return targetPhysicalValue−VecConstantIfCut;
});

const VecType currentSource = VecType(&potentials[idxSource]);
(resSource+currentSource).storeInArray(&potentials[idxSource]);
```

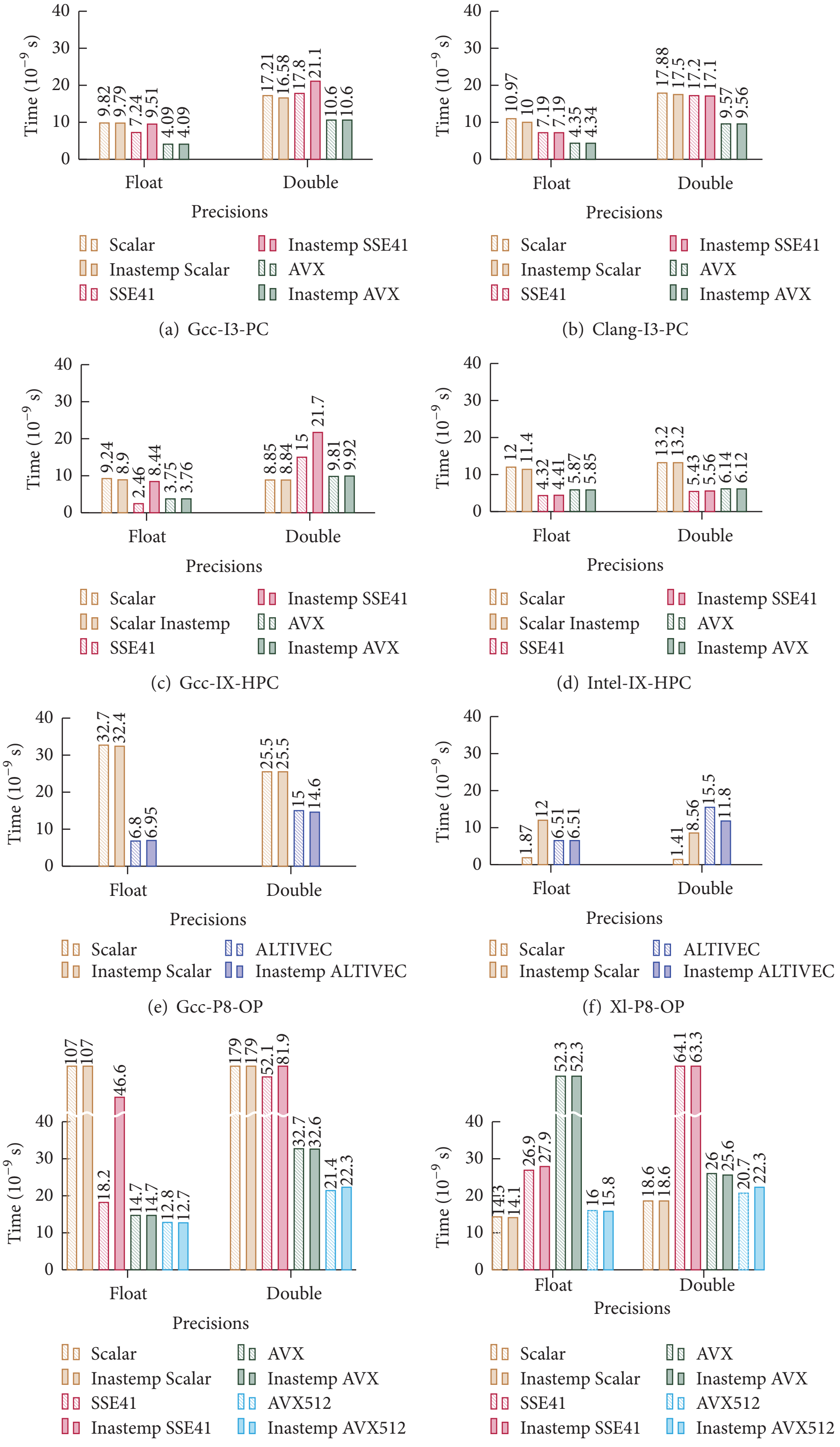
Code 15: Code extract of the particle interactions example taken from Inastemp and introduced in Section 6.3.

6.5. General Matrix-Matrix Product (Gemm) Test Case. We have implemented a Gemm kernel (this source code is available inside the Inastemp package, in the file: *Examples/Gemm/main.cpp*) that relies on an advanced blocking strategy as shown in Code 16. However, as stated, our objective is not to optimize our Gemm implementation but only to assess the Inastemp library. Therefore, we did not tune the block size for the different hardware/accuracies. On the IBM Power-8 we used a more naive Gemm implementation that appears to have better performance (comparison is not shown here).

Figure 4 shows the Flop-rate (the number of floating point operations per second) for the different configurations. On the conventional Intel CPUs, the behavior follows the

```
template <class RealType, size_t PanelSizeK, size_t PanelSizeiA,
            size_t PanelSizejB, class VecType>
void ScalarGemmIna(const RealType* __restrict__ A, const RealType* __restrict__ B,
                    RealType* __restrict__ C, const size_t size){

    const int BlockSize = VecType::VecLength;

    static_assert(PanelSizeK >= BlockSize, ”PanelSizeK must be greater than block”);
    static_assert(PanelSizeiA >= BlockSize, ”PanelSizeiA must be greater than block”);
    static_assert(PanelSizejB >= BlockSize, ”PanelSizejB must be greater than block”);
    static_assert((PanelSizeK/BlockSize)*BlockSize == PanelSizeK, ”PanelSizeK must be a . . . multiple of block”);
    static_assert((PanelSizeiA/BlockSize)*BlockSize == PanelSizeiA, ”PanelSizeiA must be a . . . multiple of block”);
    static_assert((PanelSizejB/BlockSize)*BlockSize == PanelSizejB, ”PanelSizejB must be a . . . multiple of block”);
    // Restrict to a multiple of panelsize for simplcity
    assert((size/PanelSizeK)*PanelSizeK == size);
    assert((size/PanelSizeiA)*PanelSizeiA == size);
    assert((size/PanelSizejB)*PanelSizejB == size);

    for(size_t ip = 0 ; ip < size ; ip += PanelSizeiA){
        for(size_t jp = 0 ; jp < size ; jp += PanelSizejB){

            for(size_t kp = 0 ; kp < size ; kp += PanelSizeK){

                alignas(64) RealType panelA[PanelSizeiA*PanelSizeK];
                alignas(64) RealType panelB[PanelSizeK*BlockSize];

                for(size_t jb = 0 ; jb < PanelSizejB ; jb += BlockSize){

                    CopyMat<RealType, BlockSize>(panelB, PanelSizeK, &B[jp*size + kp], size);

                    for(size_t ib = 0 ; ib < PanelSizeiA ; ib += BlockSize){

                        if(jb == 0){
                            CopyMat<RealType, BlockSize>(&panelA[PanelSizeK*ib], PanelSizeK, . . .
    &A[(ib+ip)*size + kp], size);
                        }

                        for(size_t idxRow = 0 ; idxRow < BlockSize ; ++idxRow){
                            for(size_t idxCol = 0 ; idxCol < BlockSize ; ++idxCol){
                                VecType sum = 0.;
                                for(size_t idxK = 0 ; idxK < PanelSizeK ; idxK += BlockSize){
                                    sum += VecType(&panelA[(idxRow+ib)*PanelSizeK+ idxK])
                                            * VecType(&panelB[idxCol*PanelSizeK+ idxK]);
                                }
                                C[(jp+jb+idxCol)*size + ip + ib + idxRow] += sum.horizontalSum();
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Code 16: Gemm function taken from the Inastemp examples and studied in Section 6.5.



(a) Gcc-I3-PC

(b) Clang-I3-PC

(c) Gcc-IX-HPC

(d) Intel-IX-HPC

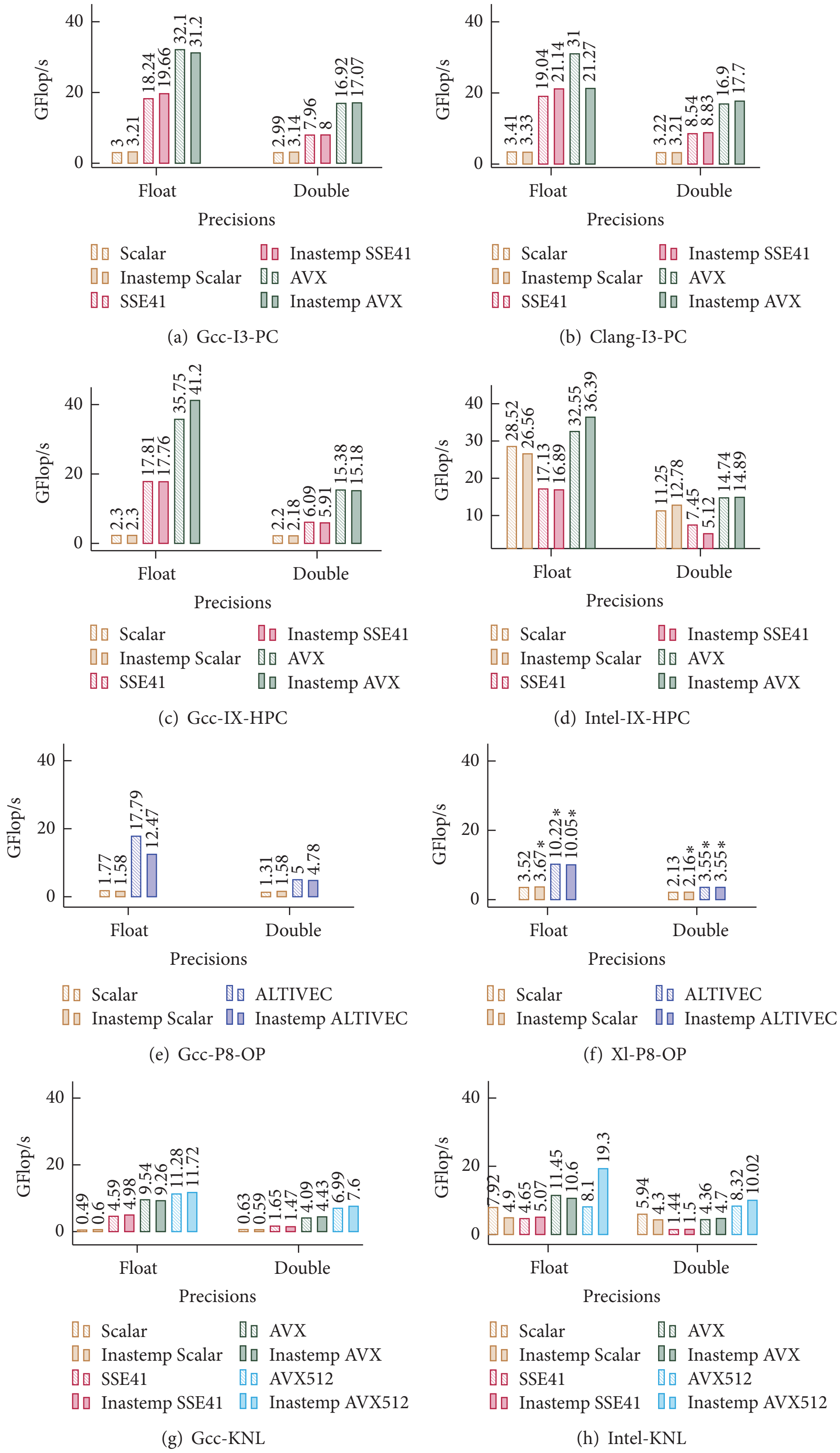
(e) Gcc-P8-OP

(f) Xl-P8-OP

(g) Gcc-KNL

(h) Intel-KNL

Figure 3: Average time in nanoseconds (10^{-9} s) to compute a natural exponential, averaged over 51200000 computations.



(a) Gcc-I3-PC

(b) Clang-I3-PC

(c) Gcc-IX-HPC

(d) Intel-IX-HPC

(e) Gcc-P8-OP

(f) Xl-P8-OP

(g) Gcc-KNL

(h) Intel-KNL

FIGURE 4: Gigaflop per second to compute a general square matrix-matrix product, where the average was taken from three executions. Matrix dimension is $N = 2058$ in Double and $N = 2560$ in Float. (∗) These executions use a simpler blocking scheme that shows better performance for the respective configurations (Xl-P8-OP Figure 4(f)).

expected pattern and using SSE or AVX provides a significant speedup; see Figures 4(a), 4(b), 4(c), and 4(d). However, the Clang compiler has not been able to apply the same degree of optimization for the AVX in Float, making the Inastemp-based kernel slower. On the IX-HPC, the Inastemp Float kernel has been optimized better by Intel and GCC. Additionally, one can note that the Intel compiler had successfully autovectorized the scalar kernels, with and without Inastemp (Figure 4(d)). Despite the high regularity of the data access in addition to the loop range known at compile time, GCC has not been able to do so, which illustrates the need for vectorization libraries. The performance of the IBM Power 8, Figure 4(e), is improved by the vectorization but remains low compared to Intel hardware. On the KNL, the Intel compiler applied a higher degree of optimization than GCC for the scalar kernels, Figures 4(g) and 4(h). However, the Intel compiler fails to optimize the pure intrinsic-based AVX512 kernel, even though it succeeds with the template Inastemp-based kernel.

7. Conclusions

We present Inastemp, a lightweight, open-source library dedicated to the development of computationally intensive kernels and their SIMD optimization. Inastemp runs on all major CPU architectures, including the recently released IBM Power-8 or Intel-KNL, and provides similar performance to pure intrinsic-based implementations in most configurations. Furthermore, Inastemp fosters the creation of elegant and maintainable code design that is crucial for long-living HPC applications. Inastemp facilitates the abstraction of the vectorization mechanism from the hardware and allows the replacement of several low-level implementations by a single template function. Its simplicity makes it usable for computational scientists, but the library can also be extended by HPC experts. Moreover, codes which utilize Inastemp are future-proof for upcoming SIMD instruction sets and will benefit from any Inastemp update without any modification, the optimizations being delegated to Inastemp developers.

In addition, our results show the limits of autovectorization by the compiler and illustrate the benefit of using intrinsic-based kernels. As a perspective, we intend to provide more mathematical functions, such as the logarithm, and to add the compilation tools to build generic binaries. Finally, the Inastemp library is already used in the Max-Planck Society in various projects supported by the MPCDF. The MPCDF will provide long-term support for Inastemp, including the incorporation of future architectures, and will address any issues which arise within the code.

Conflicts of Interest

The author declares that there are no conflicts of interest regarding the publication of this paper.

References

[1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Inc., Boca Raton, FLa, USA, 1st edition, 2010.

[2] B. Videau, K. Pouget, L. Genovese et al., “D5. 5–boast: a metaprogramming framework to produce portable and efficient computing kernels for hpc applications version 1.0,” Mont-Blanc Consortium Partners, 2017, http://www.montblanc-project.eu/sites/default/files/D5.5_0.pdf.

[3] J. Falcou and J. Serot, “Application of template-based metaprogramming compilation techniques to the efficient implementation of image processing algorithms on SIMD-capable processors,” in *Proceedings of the Advanced Concepts for Intelligent Vision Systems*, Brussels, Belgium, 2004.

[4] R. Möller, “Design of a low-level C++ template SIMD library,” Tech. Rep., Universität Bielefeld, 2016, https://www.ti.uni-bielefeld.de/downloads/publications/templateSIMD.pdf.

[5] M. Kretz, *Extending C++ for explicit data-parallel programming via SIMD vector types [Ph.D. thesis]*, Johann Wolfgang Goethe-Universität, 2015.

[6] M. Kellogg, “QuickVec C++ Library,” 2016, https://www.andrew.cmu.edu/user/mkellogg/15-418/proposal.html.

[7] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira, “Simple, portable and fast SIMD intrinsic programming: Generic SIMD Library,” in *Proceedings of the 2014 1st ACM SIGPLAN Workshop on Programming Models for SIMD/Vector Processing (WPMVP ’14)*, pp. 9–16, New York, NY, USA, February 2014.

[8] M. Gross, “Neat SIMD: Elegant vectorization in C++ by using specialized templates,” in *Proceedings of the 14th International Conference on High Performance Computing and Simulation (HPCS ’16)*, pp. 848–857, Innsbruck, Austria, July 2016.

[9] R. Leißa, I. Haffner, and S. Hack, “Sierra: A SIMD extension for C++,” in *Proceedings of the 2014 1st ACM SIGPLAN Workshop on Programming Models for SIMD/Vector Processing (WPMVP ’14)*, pp. 17–24, New York, NY, USA, February 2014.

[10] Agner Fog, “VCL C++ vector class library,” 2017, http://www.agner.org/optimize/vectorclass.pdf.

[11] P. Souza, L. Borges, C. Andreolli, and P. Thierry, “OpenVec portable SIMD intrinsics,” in *Proceedings of the 2nd EAGE Workshop on High Performance Computing for Upstream*, pp. 82–86, September 2015.

[12] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, “GRGMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation,” *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[13] E. Cieren, *Molecular dynamics for exascale supercomputers [Ph.D. thesis]*, Université de Bordeaux, 2015, http://www.theses.fr/2015BORD0174.

[14] M. Heroux, “Opportunities and challenges in developing and using scientific libraries on emerging architecture,” in *Proceedings of the Conference on Computational Science and Engineering (CSE ’15)*, SIAM, Salt Lake City, Utah, USA, 2015.

[15] B. Adelstein, “Modern C++ For HPC,” Lawrence Berkeley National Laboratory, 2016, Berkeley C++ Summit, https://sites.google.com/a/lbl.gov/berkeleycppsummit2016/.

[16] P. M. Kogge, *The Architecture of Pipelined Computers*, CRC Press, 1981.

[17] Intel, “Intel 64 and IA-32 architectures software developer’s manual: Instruction set reference (2A, 2B, 2C, and 2D),” Tech. Rep., 2016, https://software.intel.com/en-us/articles/intel-sdm.

[18] Intel, “Introduction to Intel Advanced Vector Extensions,” Tech. Rep., 2016, https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[19] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," Tech. Rep., 2016, https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

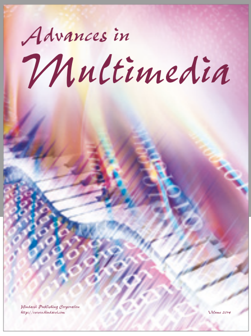
[20] B. Bramas, O. Coulaud, and G. Sylvand, "Time-Domain BEM for the Wave Equation: Optimization and Hybrid Parallelization," in *Euro-Par 2014 Parallel Processing*, vol. 8632 of *Lecture Notes in Computer Science*, pp. 511–523, Springer International Publishing, 2014.

[21] OpenMP Architecture Review Board, "OpenMP application program interface 4.0," 2013, http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

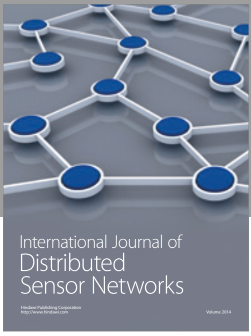
[22] INCITS/ISO/IEC, "ISO International Standard ISO/IEC 14882: 2014 - Programming languages - C++, page 95, Section 5.1.2-5," Tech. Rep., International Organization for Standardization (ISO), Geneva, Switzerland, 2016, http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS/ISO/IEC+14882:2014+.

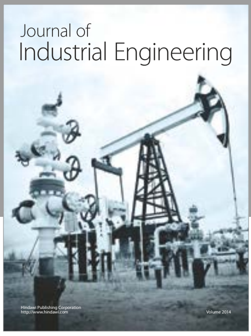
[23] A. C. I. Malossi, Y. Ineichen, C. Bekas, and A. Curioni, "Fast exponential computation on simd architectures," in *Proceedings of the HIPEAC-WAPCO*, Amsterdam, The Netherlands, 2015.

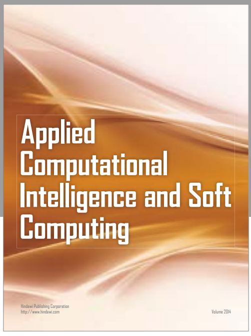
[24] IBM, "Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM Power8," 2016, https://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf.

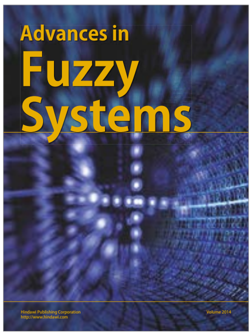

Advances in
Multimedia

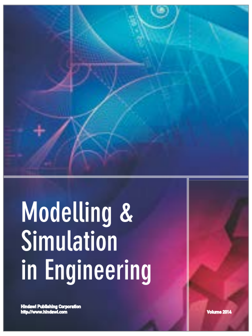
The Scientific
World Journal

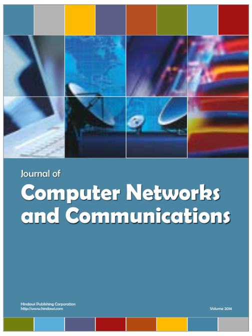

International Journal of
Distributed
Sensor Networks


Journal of
Industrial Engineering

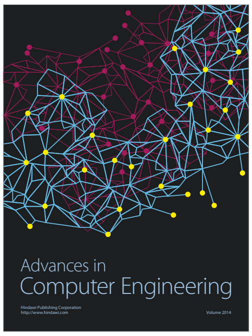

Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

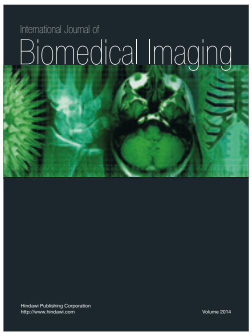

Modelling &
Simulation
in Engineering

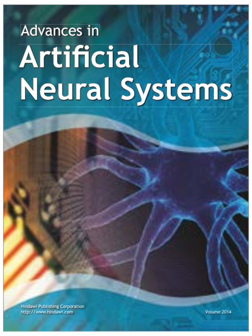

Journal of
Computer Networks
and Communications

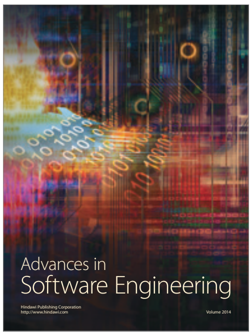
Advances in
Artificial
Intelligence


Advances in
Computer Engineering

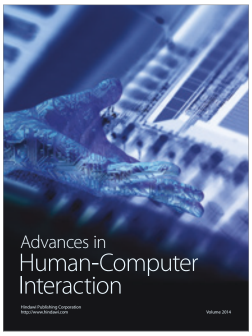
International Journal of
Computer Games
Technology

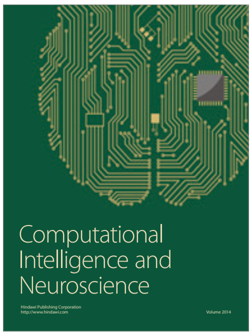

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering