Implementing Mathematical Data Types on Top of .NET

Jürg Gutknecht[1], Roman Mitin[1], Nikolai Zolotykh[2], Nina Gonova[2]

[1]Institute for Computer Systems – ETH Zurich
Zurich, Switzerland

[2]Department of Computational Mathematics and Cybernetics – State University of Nizhny Novgorod
Nizhny Novgorod, Russia

{gutknecht, mitin}@inf.ethz.ch,
{nina.gonova, nikolai.zolotykh}@gmail.com

Abstract. *We present our experience with the design and implementation of a mathematical extension for an object-oriented programming language. Primarily this includes support for multidimensional matrices, indexing by ranges and vectors, sparse data structures. As first class citizens in the language, they lead to a more natural code in scientific programs and enable powerful compiler optimizations. We discuss our concept of smoothly integrating mathematical constructs into the language and compiler on top of .NET. Finally we show that this approach results in a reasonably good performance and thereby we prove the suitability of the (fully managed) .NET platform for high performance computing applications.*

1. Introduction

Support for mathematical programming exists not only in highly specialized packages like MATLAB, Mathematica and R, but also in some general-purpose languages, for example, Fortran 90, ZPL, APL and Python (NumPy). Several attempts of adding mathematical extensions to members of the Pascal language family have also been made such as, for example, Pascal-XSC [1] and Math Oberon [2].

Zonnon [3] is another descendant of Pascal. It grew out of our participation in Project 7/ 7+ as implementers of academic languages. Zonnon embodies an advanced actors-like object model without, however, sacrificing the benefit of full interoperability [4]. As a Pascal descendant, Zonnon has a clear, non-cryptical and expressive syntax that makes it an attractive tool for the development of scientific applications also by non-computer scientists like, for example, applied mathematicians.

Our project aimed at two different and not (obviously) compatible goals: 1) providing powerful constructs with a clean semantics for intuitive and productive mathematical programming and 2) offering opportunities for the compiler to generate optimized (MSIL) code, especially in the cases of mathematical operations on multidimensional arrays. Generally speaking, the optimization potential depends directly on the constraints (or the lack of constraints) imposed by the programming model (for example in connection with pointers, reference aliasing, statement independence, data locality, etc.) and by the target architecture (for example whether the architecture provides vector instructions or not, see [5]). As compiler developers we can fully use the potential of the .NET framework.

The rest of this paper is organized as follows. In Section 2 we present the mathematical extensions to the Zonnon language. Section 3 briefly describes the overall framework of the Zonnon compiler for .NET, while the specific mappings of the mathematical constructs to .NET are presented in detail in section 4. Sections 5 and 6 are devoted to the discussion of specific compiler optimizations and to the performance evaluation.

2. Mathematical Language Extensions

The naturality and suitability of the language constructs designed for some specific application domain have a substantial impact on both the productivity of development and the reliability of the final program. When we were developing our mathematical extensions we strived for a clear, coherent and unified concept that on the one hand directly reflects the corresponding mathematical concepts on a sufficiently high level of abstraction and, on the other hand, is generic and minimal in terms of new language constructs. Consistent operational semantics together with strong typing and the corresponding option of static compiler checks are intended to support the programmer implicitly and safeguard the resulting software against traps as they are typically present when using mathematical libraries.

In general, three groups of mathematical constructs can be distinguished in Zonnon: *indices*, *operators* and *functions*. All of these constructs can be applied on objects of the newly introduced types of *mathematical array* and *sparse mathematical array*. We intentionally adopted a MATLAB-like nomenclature.

2.1. Mathematical Arrays

Mathematical arrays are used in Zonnon for expressing composed mathematical objects such as tensors, matrices and vectors. The main semantic difference between simple array types and mathematical array types is that the former types have a reference semantics while the latter are value-types, which seems to be more natural and helps to avoid aliasing problems. Consequently, assignment in the case of simple arrays denotes assignment of references, whereas in the case of mathematical arrays assignment means deep copy.

In Zonnon we use a special modifier `{math}` to distinguish mathematical arrays from simple arrays. For example, a type for a vector of three elements can be defined as following:

```
type Vector = array { math } 3 of integer;.
```

The table below demonstrates the difference of the operational semantics between simple arrays and reference-objects on the one hand, and mathematical arrays and value-objects on the other hand.

Table 1. Semantic differences between simple arrays and reference-objects, mathematical arrays, and value-objects

		simple array, reference-object	mathematical array	value-object, elementary type
assignment, comparison		reference	value	value
passing to a subprogram	with `var` modifier	reference on reference	reference on reference	reference
	without `var` modifier	reference	reference on a constant array	value

2.2. The Indexing Mechanism

In mathematical algorithms it is not unusual that a certain operation must be performed not on a complete array but on some sub-structure such as (a part of) a column or row of a matrix or on a diagonal instead. With the goals of both a unified programming pattern for operations on sub-structures and an increased potential for optimizations in mind, we included in our extension a powerful indexing/ selection mechanism that makes programming of operations on sub-structures equally convenient as operations on full arrays.

If `a` is a n-dimensional array, then a *slice* can be designated as `a[index_1,..., index_n]`, where `index_i` (selector of a subset of elements in `i`-th dimension) can be either an ordinary index as described in the language report [6], or a complex index: a range, a numerical vector, or a Boolean vector:

a) The specification of a *range* looks like `[a]..[b] [by c]`, where `a`, `b` and `c` must be integer constants or variables, and the expression `a..b by c` stands for an ordered set $\{a + i \cdot c : i \in \mathbb{N},\ 0 \le i \cdot c \le b - a\}$.

b) A *numerical vector index* is a one-dimensional array of any positive length with elements of integer type. Corresponding elements of the indexed array are selected and concatenated in the same order as they appear in the index vector.

c) A *Boolean vector index* is a vector consisting of Boolean elements standing in one-to-one correspondence with the elements of the indexed array in the appropriate dimension. Each value of `true` in the Boolean vector then selects the corresponding element in the indexed array, and each value of `false` suppresses the corresponding element. Therefore, the length of the resulting sub-structure is equal to the number of `true` values in the Boolean vector.

The following example demonstrates the use of the indexing features offered in Zonnon:

```
module Main;
  var
      A : array {math} 5 of real;
      B : array {math} * of real;
      b : array {math} 5 of boolean;
      i : array {math} 3 of integer;
begin
      i := [1, 4, 0];
      b := [false, false, true, false, true];
      (*...*)
      B := A[3..];    (*B = [A[3], A[4]]*)
      B := A[i];      (*B = [A[1], A[4], A[0]]*)
      B := A[b]       (*B = [A[2], A[4]]*)
end Main.
```

2.3. Operators and Additional Functions

While our focus was on providing only a minimum set of generic constructs for mathematical programming (in contrast to offering a large set of useful features), we hardwired a set of basic operators and functions on mathematical arrays into the language. Some operations such as arithmetic and type conversion are understood to act elementwise, while others such as comparison and maximum search are reduction operators. Again with the aims of preserving the level of mathematical abstraction and of compact notation in mind, we also added some specific matrix operators.

As a general guideline, the types of results of array operations correspond to the types of analogous operations with scalars. Further details can be found in the Zonnon language report [5].

2.4. An Example

Let us now look at an example. The following Zonnon program computes the phase trajectory of a dynamical system described in terms of a system of differential equations

$$\begin{cases} \dot{x} = p(y - x) \\ \dot{y} = x(r - z) - y \\ \dot{z} = xy - bz \end{cases}$$

This is the famous Lorenz attractor, see, for example, [7].

As can be seen directly in the code, a forth-order Runge-Kutta method with a fixed time step is used for the integration.

```
type Vector3D = array {math} 3 of integer;

procedure { public } Process (x0:Vector3D; tMax, dt:real);
var x, x1, k1, k2, k3, k4 : Vector3D;
    t : real;
begin
   t := 0.0; x := x0;
   while t <= tMax do
      (* computing coefficients k1, k2, k3, k4 according to *)
      k1 := f(t, x); (*              the Runge-Kutta method *)
      k2 := f(t + dt/2, x + dt/2 * k1);  (*    we work with vectors *)
      k3 := f(t + dt/2, x + dt/2 * k2);  (* using natural notations *)
      k4 := f(t + dt, x + dt * k3);      (*          for operations *)
      x1 := x + dt/3 * (1/2 * k1 + k2 + k3 + 1/2 * k4);
      Connect(x, x1);
      x := x1;
      t := t + dt;
   end
end Process;

procedure { public } f (t: real, x: Vector3D) : Vector3D;
var Res : Vector3D;
begin
   Res :=[-x[0]*p + p*x[1],    (* computing all 3 components of the *)
          -x[0]*x[2] + r*x[0] - x[1], (* vector Res component-wise *)
           x[0]*x[1] - b*x[2]       ];
   return Res;
end f;
```

Note that type `Vector3D` is defined in terms of a mathematical array and used for denoting three-dimensional (real) vectors. Thanks to a natural combination and iteration of the basic math features offered by Zonnon, the actual algorithm within the Process structure looks amazingly compact.

Procedures in module `Graph3D` and type `Camera` enable the graphics pipeline in the matrix form. Module `Graph3D` contains matrix transformations, type `Camera` makes use of them when mapping points of the attractor:

```
module Graph3D;
      import System.Math as Math;
      const arc = 0.01745;
      type {public, ref} Vector = array {math} 4 of real;
      type {public, ref} Matrix = array {math} 4,4 of real;

      procedure {public} Hom (x: Vector3D): Vector;
      var VRes : Vector;
      begin
            VRes[0..2] := x;
            VRes[3] := 1.0;
            return VRes;
      end Hom;

      procedure {public} Unhom (x: Vector) : Vector3D;
      var VRes : Vector3D;
      begin
            VRes := x[0..2] / x[3];     (* dividing the first three  *)
               (* components of the vector x into the last component *)
            return VRes;
      end Unhom;

      procedure {public} Trans (v1, v2, v3: real) :  Matrix;
      var MRes : Matrix; i : integer;
      begin
            MRes :=[[1., 0., 0., v1], (* computing components of the *)
                    [0., 1., 0., v2], (*   matrix MRes line by line *)
                    [0., 0., 1., v3],
                    [0., 0., 0., 1.]];
            return MRes;   (* shifting matrix *)
      end Trans;

      procedure {public} RotX (phi: real) : Matrix;
      var c, s : real;  MRes : Matrix;
      begin
            c := Math.Cos(arc*phi);
            s := Math.Sin(arc*phi);
            MRes := [[1., 0., 0., 0.],
                     [0., c , -s, 0.],
                     [0., s ,  c, 0.],
                     [0., 0., 0., 1.]];
            return MRes;   (* matrix of rotation about x-axis *)
      end RotX;

      procedure {public} RotY (phi: real) : Matrix;
      var c, s : real; MRes : Matrix;
      begin
            c := Math.Cos(arc*phi);
            s := Math.Sin(arc*phi);
            MRes := [[c , 0., s , 0.],
                     [0., 1., 0., 0.],
                     [-s, 0., c , 0.],
                     [0., 0., 0., 1.]];
            return MRes;   (* matrix of rotation about y-axis *)
      end RotY;
      procedure {public} Proj (left, right, bottom, top,
                         near, far: real): Matrix;
      var MRes : Matrix;
      begin
        MRes := [
           [2*near / (right-left), 0., (right+left)/(right-left), 0.],
           [0., 2*near/(top-bottom), (top+bottom)/(top-bottom),  0.],
           [0., 0., -(far+near)/(far-near), -2*far*near/(far-near)],
           [0., 0., -1., 0.]];
        return MRes;
      end Proj;
end Graph3D.

type {public, ref} Camera = object (zm0, d0, azim0, elev0,
            left0, right0, bottom0, top0, near0, far0: real)
var   M : Graph3D.Matrix;
      left, right, bottom, top, near, far, zm, d, azim, elev : real;

      procedure {public} Move (dazim, delev, dd: real);
      begin
            azim := azim + dazim;
            elev := elev + delev;
            d := d + dd;
            M := Graph3D.Proj(left, right, bottom, top, near, far)
               * Graph3D.Trans(0.0, 0.0, -d)
               * Graph3D.RotX(elev)      (* matrix multiplication *)
               * Graph3D.RotY(-azim)
               * Graph3D.Trans(0.0, 0.0,- zm)
      end Move;

      procedure {public} Map (x: Vector3D) : Vector3D;
      begin
            return Graph3D.Unhom(M * Graph3D.Hom(x))
      end Map;
begin (* initialization *)
end Camera;
```

2.5. Sparse Arrays

In many scientific applications, notably in those based on solving partial differential equations, sparse structures play a dominating role. As it can be easily confirmed by inspecting any arbitrary snippet of code, programming with sparse structures immediately leads to lengthy, obscure and unreadable code. Consequently, we decided in favor of adding a suitable notion of sparse structure to the language. However, carefully weighing benefits against added complexity, we refrained from adding specific sparse structures such as triangular, banded etc. and added one generic sparse type instead. The implementation was based on the work done in [8].

The notational difference between dense and sparse structures is minimal: simply use the modifier `{sparse}` instead of `{math}` and pass a set of suitable parameters to the constructor. The main attraction of this approach is now the fact that nearly all the mathematical operations in Zonnon can be applied equally to full or sparse matrices, without any explicit indication by the application program. Behind the scenes, the corresponding sparse data structure is mapped to a structure consuming space in size proportional to the number of nonzero entries.

The following example shows how convenient it is to define, create and use sparse matrices.

```
module Main;
type sparse_matrix = array {sparse} *,* of integer;
var
      a, b : sparse_matrix;
      i, j, s : array {math} 3 of integer;
begin
      i := [1, 5, 0];
      j := [2, 4, 1];
      s := [35, 72, 11];
      a := new sparse_matrix(i, j, s);
      b := -a; (* element-wise negation *)
end Main.
```

2.6. Extensibility

Hand-in-hand with our approach of adding a minimal set of generic constructs to the language goes extensibility and adaptability. This is important because no set of built-in features can satisfy up to the details all possible needs. The constructs must allow certain arbitrary functionality on the implementation level in order to be able to adjust to current situations [9]. One source of flexibility is mathematical arrays where the base element type is user-defined. If there are overloaded operators for this type, they will be used when performing operations on mathematical arrays. The following example demonstrates the power of this feature:

```
module Main;
  type vc = array {math} 10 of complex;
  (* define complex numbers type *)
  type {public, ref} complex = object
  var {public}
      re, im : real;
      (* ... *)
  end complex;

  (* overload operator '+' for complex numbers *)
  operator {public} '+' (z1, z2 : complex): complex;
    var res : complex;
  begin
      res := new complex;
      res.re := z1.re + z2.re;
      res.im := z1.im + z2.im;
      return res;
  end '+';

var ac, bc, cc : vc;
begin
(* because operator '+' is overloaded for the complex type,
   it is possible to sum two mathematical arrays of complex numbers *)
      cc := ac + bc
end Main.
```

User-defined mathematical arrays add extra opportunities for optimizations. For instance, the compiler can eliminate unnecessary temporary copies when applying operators. In the example above, the `operator '+'` creates a new instance of type `complex` for each element of array `cc`. In fact, the result can directly be written to `cc[i]` by means of inlining of the modified body of the '`+`' operator.

3. The Zonnon Compiler Framework

The Zonnon compiler is written in C#, and it generates ordinary .NET assemblies containing intermediate MSIL code and metadata. For both code generation and integration with Microsoft Visual Studio the Common Compiler Infrastructure library (CCI) [10] as provided by Microsoft is used. The CCI library provides support in terms of abstract classes for high-level infrastructure (in particular, structures for building attributed program trees and methods for performing semantic checks on the trees), code generation, and integration with the Visual Studio IDE [11].

Basically, the compiler is organized in a quite traditional way: the scanner transforms the source text into a sequence of tokens to be consumed by the parser. After that, the compiler builds an Abstract Syntax Tree (AST) with Zonnon-oriented nodes which, in turn, is then transformed into a tree with CCI-oriented nodes. The final result of the entire compilation process is a .NET assembly. The data flow in the compilation process is shown in Figure 1.

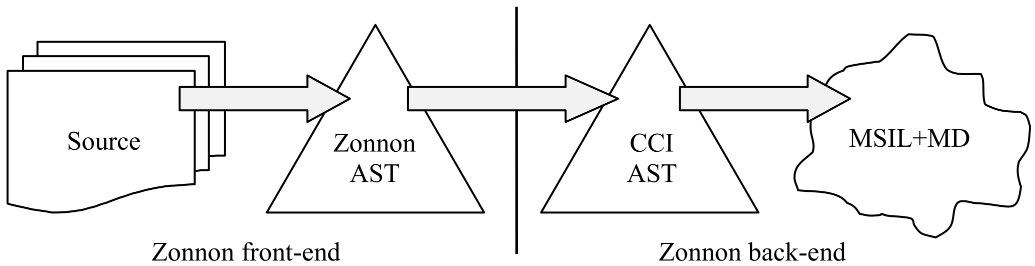


Figure 1. Compilation process

The main reason for the construction of two trees (instead of just one) is our desire for the separation of language-oriented concerns and platform-oriented concerns. The mapping of the mathematical extensions to the .NET framework, which is of particular interest here, takes place at the conversion of the Zonnon AST into the CCI AST.

4. Mapping to .NET

A language can be implemented on the .NET interoperability platform if and only if there is a mapping of its constructs to the Common Language Runtime (CLR) [12]. Depending on the paradigm and model represented by the language, this may be quite challenging [13].

There are two options for the mapping of mathematical constructs to the corresponding CLR structures:

1. The mapping is done at the stage when the Zonnon AST has already been built but the CCI AST has not been created yet, by means of the Zonnon tree concretization process. In this case the mapping is independent of the .NET framework, because we are only using Zonnon-related structures. The main disadvantage of such an approach is the limitation to platform capabilities that can be expressed in terms of the Zonnon programming language.

2. The mapping occurs during the conversion of the Zonnon tree into the CCI tree. This approach makes the mappings depending on the target platform, and it lets us use its full potential.

Weighing up all the pros and cons, we have chosen the second approach.

In order to implement a mathematical operation in the most general case, the compiler uses CCI structures to construct a proper function, which implements this operation, and then inserts its call at the right place in the program tree.

For example, if `a` is a vector, `b` is a matrix, `k` is a scalar, the following Zonnon construction

```
a := b[m, n1..n2 by step] * k;
```

will be mapped by the compiler into a CCI structure corresponding to the following statement in C#:

```
a = ElementWiseSRArrayScalarMult2d<type of b><type of k>
      (b, k, m, n1, n2, step);
```

The compiler then generates a function with the following implementation expressed in C#:

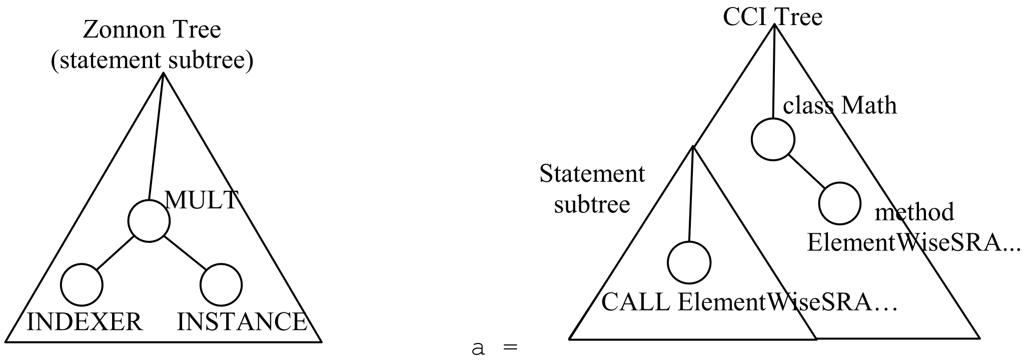
```
public static <type of result>[]
ElementWiseSRArrayScalarMult2d<type of b><type of k>
(<type of b> [,] left, <type of k> right,
         Int32 n_s0, Int32 n_r0_from, Int32 n_r0_to, Byte n_r0_by)
{
        Int32 i0 = new Int32(), n0 = new Int32();

       n0 = ((n_r0_to-n_r0_from) / n_r0_by) + 1;
       <type of result>[] res = new Int32[n0];

       for ( i0 = 0; i0 < n0; i0 += 1)
             res[i0] = left[n_s0, n_r0_from + (i0*n_r0_by)] * right;

       return res;
}
```

The mapping of the Zonnon construct described above to the CCI structures is shown schematically in Fig.2.



```
a := b[m, n1..n2 by step]*k
```

```
a =
ElementWiseSRArrayScalaeMult2sIntInt
(b, k, m, n1, n2, step);
```

Figure 2. A Zonnon tree and a CCI tree when mapping mathematical extensions

Functions generated by the compiler have unique names synthetically constructed from the types of base elements and their number of dimensions, the implemented operation, and additional indices. When converting another mathematical construct, the compiler first searches for a matching function; if such a routine has not yet been created, it will be generated automatically by the compiler.

5. Optimizations

Besides of compact code and optimal readability, the second important benefit to be drawn from the implementation of powerful mathematical extensions on the language level is the built-in potential for runtime optimization by the compiler. We can distinguish the following groups of optimizations:

1. Typical optimizations for arithmetic expressions applied to mathematical arrays: constant folding, common sub-expression elimination, loop invariant code motion.

Example 1. Let us consider the following piece of code:

```
x0 := A*b + c;
x1 := A*b + d;
```

The process of common sub-expression elimination, `A*b` in this case, is a usual optimization step in any compiler:

```
t0 := A*b;
x0 := t0 + c;
x1 := t0 + d;
```

If `A` is a matrix and `x0`, `x1`, `b`, `c`, `d` are vectors, the operators of addition and multiplication are overloaded, and the compiler is not allowed to apply such an optimization in the general case. Thus, we need to provide complementary support for performing similar optimizations on mathematical arrays.

2. Memory using optimizations (copy propagation).

The semantics of mathematical arrays in Zonnon requests that their value be copied whenever the array is assigned to some variable. However, performing a (deep) copy each time can be ineffective in practice. A possible way around unnecessary copying of arrays is keeping control of their ownership in combination with a copy-on-write strategy. This can be done statically.

Example 2. Let us now consider matrices A, B, and C of size 100 x 100, defined as local variables and let us take a look at this example:

```
B := A;
D := A;
D[7,9] := B[5,7];
```

A and D are used later in the code only for reading, B is not used anymore. If we apply the most straightforward implementation, then copies of `A` for `B` and `D` will be created. However, a simple analysis of the control flow shows that array `B`, once initialized, is used for reading only, and that `A` is not changed at all. So, there is no need to create a deep copy during `B := A`, because copy references are OK in this case.

3. Effective code generation for compound expressions: loop merging, operator ordering.

Example 3. The assignment `x := A*b + c`, where `A` is a matrix, `x`, `b`, `c` are vectors, is mapped by means of a simple conversion into a sequence of function calls, similar to `Math.Assign(x, Math.Add(Math.Multiply(A, b), c))`. During the execution of this statement two additional arrays will be created, and the last one will be assigned to vector x.

However, the compiler could alternatively generate a much more specific function that makes no use of additional variables and avoids excessive loops:

```
for(int i=0; i<x.Length; i++)
{
      int tmp = c[i];
      for(int j=0; j<b.Length; j++)
      {
            tmp += A[i,j]*b[j];
      }
      x[i] = temp;
}
```

Example 4. For resolving the expression `A := B1 * !B2`, where all variables are matrices, the compiler has to transpose `B2`, store the result in main memory, and then multiply it with `B1`. Taking a more global view it is possible to implement matrix/ transposed matrix multiplication as a “dwarf” directly by reading both matrices row-by-row.

4. Optimal choice of method for matrix multiplication.

It is well known that matrix multiplication should be implemented differently according to the sizes of the involved matrices. For small matrices (for example, of size 2x2, 3x3, 4x4) the costs of even a single function call are too high. In this case it makes much sense to eliminate the loop by unrolling the corresponding computations into a sequence of assignment statements[1]. In contrast, minimizing cache misses is of paramount importance for bigger matrices.

If the compiler is informed about the size of matrices in computations, it can easily choose the optimal method. In the cases of unknown size, the compiler is still able to make an assumption or to dynamically inject a size check as a basis for choosing the optimal method.

There are two main approaches for implementing matrix operations, in particular matrix multiplication. The first one uses information about the architectural characteristics for taking advantage of them (see, for example, [14]). When applying this approach, it is in particular possible to choose an optimal size for matrix blocks with the aim of avoiding CPU stalls. The second approach suggests the use of cache-oblivious algorithms [15], which looks quite attractive at first sight for being implemented in a compiler. However, experimental results in [16] have shown that the optimal performance can only be achieved with respect to some theoretical models of a memory hierarchy; in practice cache-oblivious algorithms are heavily defeated by cache-conscious algorithms in the field of dense linear algebra [16]. Our

[1] SSE instructions are not available directly on .NET, but we still can hope that if the JIT compiler encounters a simple sequence of instructions in a base block which can be vectorized, it will do it, if not now, then in later versions of .NET framework.

implementation of an optimized matrix multiplication for Zonnon has also been guided by [17].

6. Experimental Results

In this section we finally give some facts and figures of the performance of our implementation in real applications. As a sample benchmark we chose some machine learning algorithms, for the reason that they require a large amount of expensive computations on vectors and matrices that can conveniently be represented by means of mathematical arrays.

The underlying computer is based on an Intel Core 2 Duo T7250 CPU running at 2,00GHz and connected to a 2038 MB RAM.

Example 1. We have implemented in Zonnon a simple classification algorithm based on kernelized clusterization and compared its performance with a corresponding implementation in Oberon [18] for the classification of the USPS handwritten digits (0 and 1) problem. The Oberon compiler used for comparison is generating native code, and it is highly optimized towards exploiting SIMD capabilities.

Table 2. Experimental results for digits classification problem

Kernel type	Time spent on (in ms)	Zonnon/C# unoptimized	Zonnon optimized	Oberon optimized
Linear	learning	1219	339	32
	train sample evaluating	2660	667	312
	test sample evaluating	18264	4050	780
Polynomial	learning	1274	334	31
	train samples evaluating	2898	653	343
	test samples evaluating	18455	4098	936
Gaussian	learning	4268	2593	8331
	train samples evaluating	8377	5523	16738
	test samples evaluating	51600	34892	101822

As the results show, the optimized Zonnon implementation is up to 4.5 times faster than its unoptimized counterpart. It is worth noting in this connection that the performance of unoptimized Zonnon is roughly equal to the performance of similar algorithms written in C#. Consequently, developing mathematical applications in C# is less effective in comparison to Zonnon both in terms of development time and runtime efficiency.

Example 2. We have used a kernel ridge regression method with a Gaussian kernel for image reconstruction from irregularly sampled data. Such problems occur frequently in the context of biomedical imaging [19].

Table 3. Experimental results for image reconstruction problem

	Zonnon/C# unoptimized	Zonnon optimized	Matlab
Application in general	719784	104867	69910

SLE solving	571265	65541	32921

Image reconstruction is a particularly computing-intensive application. It is therefore interesting to compare the performance of our math extensions to a general purpose language with the performance of a highly specialized math language such as Matlab. As the table shows, the resulting application in optimized Zonnon is approximately 1.5 times slower than the corresponding application in Matlab. The crucial operation in this application is solving large system of linear equations with the conjugate gradients method, where Matlab is about 2 times faster than optimized Zonnon. Considering that Matlab takes advantages of BLAS and Intel MKL libraries which are highly optimized with concrete hardware in mind, this is still a respectful result for an application on top of .NET.

7. Conclusion

The main purpose of the Zonnon language project is to open up a field for experiments with evolutionary language concepts [20] and to investigate the potential of the .NET framework and the Microsoft CCI tools for compilers development. The Zonnon compiler is the first compiler for .NET that is fully integrated into Visual Studio and developed outside of Microsoft. Moreover, to the best of our knowledge, the mathematical extension of the Zonnon programming language is the first extension of this kind for the .NET framework.

We have analyzed the concepts of matrix-vector style of programming as they are provided by current languages and packages and, based on the results of our study, we have extended the arsenal of the Zonnon language with a generic set of powerful math constructs. This endeavor was successful. We have also extended the existing Zonnon language test suite so that it covers the mathematical extensions. We showed that 1) the approach of amalgamating Matlab-like convenience and a general purpose object-oriented programming style is feasible and clearly superior to both a) using math libraries and b) programming math applications natively in C# and that 2) the .NET just-in-time runtime does a very good job in optimizing vector/ matrix code. In cases of intensive use of highly optimized “dwarfs” such as the conjugate gradient method, special purpose math languages like Matlab are still better in performance but not at all out of reach.

Acknowledgments

We are very grateful to Sergey Lyalin, Felix Friedrich, Alexey Morozov for a number of comments and suggestions and Prof. Viktor Gergel for his support.

References

[1] Klatte, R., Kulisch, U.W., Neaga, M., Ratz, D. and Ullrich, C.P., *Pascal-XSC: Language Reference with Examples*, Springer-Verlag New York, Inc., 1992.

[2] Friedrich, F., Gutknecht, J., Morozov, O. and Hunziker, P., *A Mathematical Programming Language Extension for Multilinear Algebra*, Proceedings of Kolloqium über Programmiersprachen und Grundlagen der Programmierung, Timmendorfer Strand, 2007.

[3] Zonnon Compiler http://zonnon.ethz.ch/compiler/download.html.

[4] Lowy, J., *Programming .NET components*, 2nd Edition, O'Reilly Media, Inc., 2005.

[5] Allen, R. and Kennedy, K., *Automatic Translation of FORTRAN Programs to Vector Form*, Rice University, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, October 1987.

[6] Gutknecht, J., *Zonnon Language Report*, Draft, Zurich, July 2009.

[7] Neymark, Yu.I. and Landa, P., *Stochastic and chaotic oscillations*, Nauka, Moscow, 1987.

[8] Gilbert, J., Moler, C. and Schreiber, R., *Sparse Matrices in MATLAB: Design and Implementation*, SIAM. J. Matrix Anal. & Appl. Volume 13, Issue 1, January 1992.

[9] Friedrich, F. and Gutknecht, J., *Array-Structured Object Types for Mathematical Programming*, Proceedings of Joint Modular Language Conference, Oxford 2006.

[10] Common Compiler Infrastructure (CCI) http://ccimetadata.codeplex.com/.

[11] Zueff, E., *Common Compiler Infrastructure from a Compiler Writer's Perspective*, Microsoft Corporation Video, USA, Redmond, 2003.

[12] Box, D. and Sells, C., *Essential .NET, Volume I: The Common Language Runtime*, Addison-Wesley Professional, 2002.

[13] Gutknecht, J. and Zueff, E., *Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET*, Extended Abstract, OOPSLA'2002 Conference, Seattle, November 2002.

[14] Goto, K., *Anatomy of High-Performance Matrix Multiplication*, ACM Transactions on Mathematical Software (TOMS) archive, Volume 34, Issue 3, May 2008.

[15] Prokop, H., *Cache-Oblivious Algorithms*, Master Thesis at the Massachusetts Institute of Technology, June 1999.

[16] Yotov, K., Roeder, T., Pingali, K., Gunnels, J. and Gustavson, F., *An experimental comparison of cache-oblivious and cache-conscious programs*, Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures, San Diego, California, USA, 2007.

[17] Parello, D., Temam, O. and Verdun, J.-M., *On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance: matrix-multiply revisited*, Proceedings of the 2002 ACM/IEEE conference on Supercomputing table of contents, Baltimore, Maryland, 2002.

[18] Theodoridis, S. and Koutroumbas, K., *Pattern Recognition*, 4th Edition, Academic Press, 2008.

[19] Morozov, O. and Hunziker, P., *Tensor B-Spline Reconstruction of Multidimensional Signals From Large Irregularly Sampled Data*, The Seventh International Kharkov Symposium on Physics and Engineering of Microwaves, Millimeter and Submillimeter Waves, Kharkov, Ukraine, 2010.

[20] Gutknecht, J., Mitin, R. and Zueff, E., *Project Zonnon: A .NET Language Platform Challenge*, Proceedings of the Conference on Innovative Views of .NET Technologies (IVNET'06), Florianopolis, Brazil, October 2006.