On the Vectorization of Engineering Codes Using Multimedia Instructions

Manoel Cunha[1] and Alvaro Coutinho and J.C.F. Telles[2]

[1] Federal University of Paraná, Curitiba PR, Brazil
[2] Federal University of Rio de Janeiro, Rio de Janeiro RJ, Brazil

Abstract. After years dominating high performance computing, expensive vector computers were gradually replaced by more affordable solutions and the use of vectorization techniques once applied to many scientific codes also faded. This paper addresses the vectorization of engineering codes using Streaming SIMD Extensions (SSE) also known as multimedia instructions. This particular kind of vectorization differs from the old vectorization techniques in the sense that it relies on hardware features and instruction sets only present on modern processors. Evolving from Intel MMX, SSE is the fifth generation of an established technology highly suited to handle computing intensive tasks like encryption/decryption, audio and video compression, also including digital signal processing but not so well explored for scientific computing, specially among engineering programmers. To demonstrate the benefits of vector/SIMD computing and its implementation on existing scalar algorithms, the authors present this technique applied to an engineering program for the solution of two dimensional elastostatic problems with the boundary element method. Taking an application from a well-know reference on the area, the paper focus on the programming techniques and addresses common tasks used in many other codes, like Gauss integration and equations system assembling. Thus, the vectorization guidelines provided here may also be extended to solve many other types of problems, using other numerical methods as well as other multimedia instruction set extensions. Numerical experiments show the effectiveness of the proposed approach.

1 Introduction

Over decades, since the invention of the first computers, hardware and software resources have been created or modified to follow the increasing complexity of engineering and scientific problems. During many years, vector computers dominated high performance computing but its technologies have been recently superseded by more affordable architectures. In the current world of off-the-shelf-built clusters and multi-core blades, only two major manufacturers, NEC and Cray, still offer expensive solutions. Nowadays, vector computers share less than 1% of the high performance computer market.[3]

[3] www.top500.org

Despite the fading use of such computers the benefits of vector computing were not forgotten by hardware engineers who brought the old technology into the new processors present on today's clusters, servers, desktops and notebooks. The first step in this direction came in the late 90's with instructions sets such as Intel MMX, AMD 3D-Now and Apple Altivec. Originally developed for multimedia applications, these Single-Instruction-Multiple-Data (SIMD) instructions quickly evolved into powerful extensions highly suited to applications processing data streams such as encryption/decryption, audio and video compression, digital image and signal processing, among others. Today, Streaming SIMD Extensions (SSE) represent an established technology with significant impact in processor performance.

Unfortunately, developers of engineering codes still do not take full advantage of the resources offered by modern processors and compilers, such as auto-vectorization or explicit language instructions. This fact has called the attention of the present authors that address this work to engineering programmers willing to speed their applications with the use of multimedia instructions. Here, this vectorization technique is applied to a well known boundary element application to solve two-dimensional elastostatic problems, although the concepts can also be implemented to other kind of problems and alternative numerical methods.

The present text is organized as follows: the next section presents an outline of the boundary element theory and the following section describes the selected application. Section 4 introduces the Streaming SIMD Extensions while Section 5 details the SSE implementation of the code. In Section 6 a performance analysis is presented. The paper ends with a summary of the main conclusions.

2 Outline of the Boundary Element Method

The boundary element method (BEM) [1] is a technique for the numerical solution of partial differential equations with initial and boundary conditions.

Using a weighted residual formulation, Green's third identity, Betty's reciprocal theorem or some other procedure, an equivalent integral equation can be obtained and converted to a form that involves only surface integrals performed over the boundary. The bounding surface is then divided into elements and the original integrals over the boundary are simply the sum of the integrations over each element, resulting in a reduced dense and non-symmetric system of linear algebraic equations.

The discretization process involves selecting nodes on the boundary, where unknown values are considered. Interpolation functions relate such nodes to the approximated displacements and tractions distributions on the respective boundary elements. For linear 2-D elements, nodes are placed at, or near, the end of the elements and the interpolation function is a linear combination of the two nodal values. High-order elements, quadratic or cubic, can be used to better represent curved boundaries using three and four nodes, respectively.

Once the boundary solution has been obtained, interior point results can be computed directly from the basic integral equation in a post-processing routine.

2.1 Differential Equation

Elastostatic problems are governed by the well-known Navier equilibrium equation. Using the so-called Cartesian tensor notation, may be written for a domain Ω in the form :

$$G\, u_{j,kk} + \frac{G}{1-2\,\nu}\, u_{k,kj} + b_j = 0 \qquad in \quad \Omega \tag{1}$$

subject to the boundary conditions :

$$\begin{aligned} u &= \bar{u} \qquad on\ \Gamma_1 \qquad and \\ p &= \bar{p} \qquad on\ \Gamma_2 \end{aligned} \tag{2}$$

where u are displacements, p are surface tractions, $\bar{u}$ and $\bar{p}$ are prescribed values and the total boundary of the body is $\Gamma = \Gamma_1 + \Gamma_2$. G is the shear modulus, ν is Poisson's ratio and b_j is the body force component. Notice that the subdivision of Γ into two parts is conceptual, i.e., the same physical point of Γ can have the two types of boundary conditions in different directions.

2.2 Integral Equation

An integral equation, equivalent to Eqs. (1) and (2), can be obtained through a weighted residual formulation or Betty's reciprocal theorem. This equation, also known as Somigliana's identity for displacements, can be written as :

$$u_i(\xi) = \int_\Gamma u^*_{ij}(\xi, x)\; p_j(x)\; d\Gamma(x) - \int_\Gamma p^*_{ij}(\xi, x)\; u_j(x)\; d\Gamma(x) \tag{3}$$

where $b_i = 0$ was assumed for simplicity and the starred tensors, u^*_{ij} and p^*_{ij}, represent the displacement and traction components in the direction j at the field point x due a unit point load applied at the source point ξ in i direction.

In order to obtain an integral equation involving only variables on the boundary, one can take the limit of Eq. (3) as the point ξ tends to the boundary Γ. This limit has to be carefully taken since the boundary integrals become singular at ξ. The resulting equation is :

$$c_{ij}(\xi)\; u_j(\xi) + \int_\Gamma p^*_{ij}(\xi, x)\; u_j(x)\; d\Gamma(x) = \int_\Gamma u^*_{ij}(\xi, x)\; p_j(x)\; d\Gamma(x) \tag{4}$$

where the coefficient c_{ij} is a function of the geometry of Γ at the point ξ and the integral on the left is to be computed in a Cauchy principal value sense.

2.3 Discretization

Assuming that the boundary Γ is discretized in N elements, Eq. (4) can be written in the form :

$$c_{ij}\; u_j + \sum_{k=1}^{N} \int_{\Gamma_k} p^*_{ij}\; u_j\; d\Gamma = \sum_{k=1}^{N} \int_{\Gamma_k} u^*_{ij}\; p_j\; d\Gamma \tag{5}$$

The substitution of displacements and tractions by element approximated interpolation functions in Eq. (5) leads to :

$$\mathbf{c_i}\ \mathbf{u_i} + \sum_{\mathbf{k=1}}^{\mathbf{N}} \mathbf{h}\ \mathbf{u} = \sum_{\mathbf{k=1}}^{\mathbf{N}} \mathbf{g}\ \mathbf{p} \tag{6}$$

which can be rearranged in a simpler matrix form :

$$\mathbf{H}\ \mathbf{u} = \mathbf{G}\ \mathbf{p} \tag{7}$$

By applying the prescribed boundary conditions, the problem unknowns can be grouped on the left-hand side of Eq. (7) to obtain a system of equations ready to be solved by standard methods.

This system of linear equations can be written as :

$$\mathbf{A}\ \mathbf{x} = \mathbf{f} \tag{8}$$

where $\mathbf{A}$ is a dense square matrix, vector $\mathbf{x}$ contains the unknown tractions and displacements nodal values and vector $\mathbf{f}$ is formed by the product of the prescribed boundary conditions by the corresponding columns of matrices $\mathbf{H}$ and $\mathbf{G}$. Note that Eq. (8) can be assembled directly from the elements $\mathbf{h}$ and $\mathbf{g}$ without need to generate first Eq. (7).

2.4 Internal Points

Since Somigliana's identity provides a continuous representation of displacements at any point $\xi \in \Omega$, it can also be used to generate the internal stresses. The discretization process, described above, can also be applied now in a post-processing routine.

3 The Application

The program reviewed here is a well-known code presented by Telles [1] for the solution of two dimensional elastostatic problems using linear boundary elements.

The `INPUT` routine reads the program data, `MATRX` routine computes matrix $\mathbf{A}$ and the right hand side vector $\mathbf{f}$, stored in array $\mathbf{XM}$, while the `OUTPT` routine prints the boundary solution, computes and prints boundary stresses and internal displacements and stresses. The original `SLNPD` routine is here replaced by the LAPACK solver `SGESV` [2].

Routine `MATRX` generates the system of equations by assembling directly matrix $\mathbf{A}$ without creating the global $\mathbf{H}$ and $\mathbf{G}$ matrices. This is done by considering the prescribed boundary conditions for the node under consideration before assembling. The leading diagonal submatrices corresponding to $\mathbf{H}$ are calculated using rigid body translations. Consequently, when the boundary is unbounded a different type of rigid body consideration needs to be applied.

The element influence coefficients are computed calling subroutine `FUNC`. This subroutine computes all the element integrals required for the system of equations, internal displacements and internal stresses. Numerical integrals are performed over non-singular elements by using Gauss integration. For elements with the singularity at one of its extremities the required integrals are computed analytically to obtain more accurate results.

The boundary stresses are evaluated using subroutine `FENC` that employs the interpolated displacements and tractions to this end. Here, the contribution of adjacent elements to the common boundary nodes is automatically averaged for non-double nodes. The internal displacements and stresses are obtained by integrating over the boundary elements using subroutine `FUNC`.

The solver is usually the most time consuming routine in BEM programs and various studies have been published on this matter. However, the generation of the equations system as well as the computing of internal points together can take the most part of the processing time [10] and demand special care. While many high-performance solvers are available from standard libraries (LAPACK, PETSc, etc), those two procedures are usually implemented by the researchers and greatly limit the speedup if not properly optimized. Hence, the vectorization programming techniques are here applied to the generation of the system of equations and the evaluation of internal point results since they can be implemented following the same techniques.

4 The Streaming SIMD Extensions

Computers have been originally classified by Flynn's taxonomy [7] according to instructions and data streams as Single-Instruction-Single-Data (SISD), Single-Instruction-Multiple-Data (SIMD), Multiple-Instruction-Single-Data (MISD) and Multiple-Instruction-Multiple-Data (MIMD).

As the name suggests, the SIMD model applies to systems where a single instruction processes a vector data set, instead of scalar operands.

One of the first SIMD implementations was the MMX technology introduced with Pentium computers intended to enhance the performance of multimedia applications. The 57 MMX instructions can process simultaneously 64-bit data sets of integer type.

With the release of the Pentium III processor an instruction set called Streaming SIMD Extensions (SSE) was added to Intel 32-bit architecture. SIMD extension comprises 70 instructions, 12 for integer operations and 50 for single-precision floating-point operations. The remaining 8 instructions are for cache control and data prefetch.

SSE features eight 128-bit dedicated registers used by SIMD floating-point instructions to process four single-precision (32 bit) values simultaneously. Unlike MMX, where vector registers are mapped onto FP registers, SSE and FP registers can be used at the same time. Hence, SSE and FP/MMX instructions can be simultaneously used.

While MMX operates on integers and SSE are essentially single-precision floating-point instructions, the SSE2 set introduced with Pentium 4 added support for double-precision (64-bits) floating-point operations and extended MMX integer instructions from 64 to 128 bits. The 144 instructions implemented by SSE2 were followed by 13 SSE3 instructions, that also support complex floating-points operations.

4.1 Auto-vectorization Compilers

The first and quickest way to implement Streaming SIMD Extensions is auto-vectorization, a feature present on most recent compilers such as Intel Fortran and C/C++.

Compilers used by scientific and engineering programmers have always included optimizing options to enhance performance. The SSE technology is now also incorporated by Fortran and C/C++ compilers, which offer compiler options to generate vectorized SSE code automatically. Once SSE compiler options are set, the compiler will search the code for vectorization opportunities, automatically replacing scalar operations by vector instructions whenever possible.

Unfortunately, like other optimization procedures, the compiler ability to generate vector code automatically is restrained by a number of factors such as problem complexity and bad programming techniques, among others. Even simple pieces of code may not be addressed correctly by auto-vectorization compilers.

For example, consider the following sample :

```
do  i = 1,2 ! @ line 445
  do j = 1,4
    G(i,j) = 0.
    H(i,j) = 0.
  enddo
enddo
```

Compiling the sample code with `-arch SSE -vec-report3` options, the Intel Fortran Compiler produces the following warnings :

```
sample.f90(445) : (col. 6) remark: loop was not vectorized: not inner loop.
sample.f90(446) : (col. 8) remark: loop was not vectorized: low trip count.
```

These messages generated in the compilation process state that the `i` loop in line 445 cannot be vectorized because it is an outer loop while the `j` loop in line 446 cannot be automatically vectorized due to its reduced number of iterations.

The nested loops can be replaced by equivalent Fortran matrix expressions :

```
G = 0. ! @ line 445
H = 0.
```

resulting in the following compiling messages :

```
sample.f90(445) : (col. 6) remark: loop was not vectorized:
vectorization possible but seems inefficient.
sample.f90(446) : (col. 6) remark: loop was not vectorized:
vectorization possible but seems inefficient.
```

The compiler is still not able to automatically vectorize that portion of code. It can be expected that more complex codes will pose greater restrictions for auto-vectorization and this example clearly shows that auto-vectorization is not just a matter of recompiling old codes with new compiler options. Programmers must rewrite their codes in order to minimize compiler limitations to generate vector executables.

While vectorization works well for long loops, nested loops with few iterations and many other common constructions usually found in engineering and scientific codes are not vectorized automatically. However, basic modifications can be applied to existing codes, helping compilers to vectorize short trip counts with SSE instructions. The implementation of such techniques in the application here considered was the object of a previous work [11] and is not discussed here.

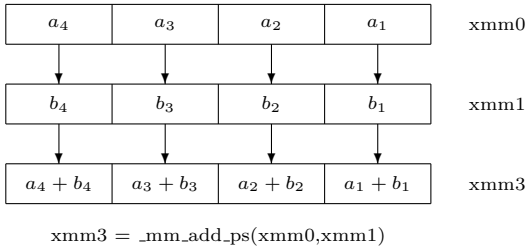
4.2 Compiler Intrinsics

Auto-vectorization options are available in Fortran as well as in C/C++ compilers. While Fortran users must rely on the compiler ability to generate SSE executables, the C/C++ language alternatively allows to insert explicit vector functions in the code, giving programmers more control to the vectorization process.

C/C++ compiler intrinsics provide the user with new data types and a set of vector functions. Thus, one benefit of SSE intrinsics is the use of C/C++ syntax of function calls and variables instead of assembly instructions and hardware registers. Intrinsics are expanded inline to eliminate call overhead.

Vector addition, subtraction, multiplication and division can be performed using the intrinsic functions `_mm_add_ps`, `_mm_sub_ps`, `_mm_mul_ps` and `_mm_div_ps`, respectively. These functions perform one operation on two sets of four floating-point single-precision values, simultaneously, as illustrated in Fig. 1.

Fig. 1. SSE Packed Addition



SSE provides a large set of vector operations. For a full description of all SSE instruction set the reader is referred to Bik [3, 4]. A more extensive C/C++ vector implementation of the original Fortran code with SSE compiler intrinsics will be addressed in the next section.

5 An SSE Implementation

In the application under study, an equation system is generated in routine `MATRX` with its influence coefficients computed by subroutine `FUNC`. This routine evaluates all the non-singular element integrals using Gauss integration.

For elements with the singularity at one of its extremities the required integrals are computed analytically. In the first case, a set of small matrix operations are initially computed, as follows :

$$\begin{bmatrix} UL_{11}\ UL_{12} \\ UL_{21}\ UL_{22} \end{bmatrix} = -C1 \left[C2\ logR \begin{bmatrix} 1\ 0 \\ 0\ 1 \end{bmatrix} - \begin{bmatrix} DR_{11}\ DR_{12} \\ DR_{21}\ DR_{22} \end{bmatrix} \right]$$

Due to its particular dimensions, these 2x2 matrices can be replaced by small vectors of size 4 and the matrix computation above can be performed with vector operations. Specially suited for SSE, these operations can be executed by SSE intrinsic functions as follows :

```
float tmp = C2 * log(R);

__declspec(align(16)) float work[4],ul[4];
__m128 xmm0,xmm1,xmm2,xmm3;

work[0] = DR[1] * DR[1];
work[1] = DR[1] * DR[2];
work[2] = DR[2] * DR[1];
work[3] = DR[2] * DR[2];

xmm1 = _mm_load_ps(work);      //                          DR
xmm2 = _mm_set_ps(1.,0.,0.,1); //                      D
xmm3 = _mm_set_ps1(-C1);       // -C1
xmm0 = _mm_set_ps1(tmp);       //        C2 * log R
xmm0 = _mm_mul_ps(xmm0,xmm2);  //        C2 * log(R) * D
xmm0 = _mm_sub_ps(xmm0,xmm1);  //        C2 * log(R) * D - DR
xmm0 = _mm_mul_ps(xmm0,xmm3);  // -C1 * (C2 * log(R) * D - DR)
_mm_store_ps(ul,xmm0);
```

SSE define the new data type `__m128` to allocate 128-bit blocks of memory that cannot be accessed directly. Intrinsic functions must be used to initialize SSE data and move it from and to floating-point arrays. These arrays are required to be aligned in 16-byte boundaries.

In the example above, the array `work` is used to store four single-precision floating-point (FP) values. Another four small arrays `xmm0`, `xmm1`, `xmm2` and `xmm3` of type `__m128` can also hold the same number and type of data. While the elements of `work` can be assigned and used in FP operations, as usual, this kind of array cannot be used as argument for intrinsic functions in vector operations. Here, function `_mm_load_ps` is used to copy to `xmm1` the values stored in `work` and `_mm_store_ps` is used to move data from `xmm0` to `work` at the end of computations. Arrays `xmm2`, `xmm3` and `xmm0` are initialized with the function `_mm_set_ps`.

In the code sample just presented, basic SSE intrinsics perform vector addition, subtraction, multiplication and division. As shown, these functions perform one operation on two sets of four floating-point single-precision values, simultaneously. The result of a vector operation on two vectors can be stored in any of these vector as well as in a third vector.

As shown, 2x2 matrices can be converted into vectors of size 4 and matrix operations can be performed with vector instructions. Thus, a very simple approach is to use SSE to evaluate those matrices leaving some intermediate operations to be executed with scalar instructions.

In the original algorithm, those matrices are computed from 2 to 6 times, accordingly to the number of Gauss integration points defined by an empiric formula. Alternatively, a fully vector implementation of the matrix computation above can be achieved by using 4 Gauss integration points and evaluating all four values of each coefficient at once, including the intermediate values.

In the application under observation, for each integration point i, the matrix coefficients can be computed as follows :

$$\begin{aligned}
XMXI_i &= CTE_i * DXY1 + XXS \\
YMYI_i &= CTE_i * DXY2 + YYS \\
R_i &= \sqrt{XMXI_i * XMXI_i + YMYI_i * XMXI_i} \\
DR1_i &= XMXI_i \ / \ R_i \\
DR2_i &= YMYI_i \ / \ R_i \\
UL11_i &= DR1_i * DR1_i - C2 * \log R_i \\
UL22_i &= DR2_i * DR2_i - C2 * \log R_i \\
UL12_i &= DR1_i * DR2_i
\end{aligned}$$

Initially, using two dimensional arrays and executed with scalar instructions, the computations presented above - including the intermediate operations - can be performed on vectors so that four values are evaluated during each operation. A possible SSE implementation of the vector computation discussed is presented in Listing 1.

Most of the operations in the vector implementation shown in Listing **??** can be performed with basic memory and arithmetic SSE instructions, already discussed in the previous section. However, SSE does not include instructions to evaluate sines, cosines, logarithms and other trigonometric functions. To bypass this limitation, vector implementations make calls to the Short Vector Math Library (SVML), an Intel library intended for use by the Intel compiler vectorizer.[4]

For each integration point i, **UL** and **PL** are used to compute another matrix, **G**, as follows :

$$\begin{bmatrix} G_{11} & G_{12} & G_{13} & G_{14} \\ G_{21} & G_{22} & G_{23} & G_{24} \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} & G_{23} & G_{24} \\ G_{21} & G_{22} & G_{23} & G_{24} \end{bmatrix} + \left[\begin{bmatrix} UL^i_{11} & UL^i_{12} \\ UL^i_{21} & UL^i_{22} \end{bmatrix} * B^i_1 \begin{bmatrix} UL^i_{11} & UL^i_{12} \\ UL^i_{21} & UL^i_{22} \end{bmatrix} * B^i_2 \right] * W^i$$

The 2x4 matrix above can be splitted into two 2x2 matrices, as follows :

$$\begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} + \begin{bmatrix} UL^i_{11} & UL^i_{12} \\ UL^i_{21} & UL^i_{22} \end{bmatrix} * B^i_1 * W^i$$

[4] http://softwarecommunity.intel.com/articles/eng/3527.htm

Listing 1. An SSE implementation

```
xmm0 = _mm_set_ps1(DXY[0]);   // DXY1
xmm1 = _mm_set_ps1(DXY[1]);   // DXY2
xmm2 = _mm_load_ps(CTEv4);    // .5 * (XI + 1)
xmm0 = _mm_mul_ps(xmm0,xmm2); // .5 * (XI + 1) * DXY1
xmm1 = _mm_mul_ps(xmm1,xmm2); // .5 * (XI + 1) * DXY2
xmm3 = _mm_set_ps1(X[II]-XS); // X[II] - XS
xmm4 = _mm_set_ps1(Y[II]-YS); // Y[II] - YS
xmm0 = _mm_add_ps(xmm0,xmm3); // XMXY = .5 * (XI + 1) * DXY1 + X[II] - XS
xmm1 = _mm_add_ps(xmm1,xmm4); // YMYI = .5 * (XI + 1) * DXY2 + Y[II] - YS
xmm2 = _mm_mul_ps(xmm0,xmm0); // XMXI^2
xmm3 = _mm_mul_ps(xmm1,xmm1); // YMYI^2
xmm2 = _mm_add_ps(xmm2,xmm3); // XMXI^2 + YMYI^2
xmm2 = _mm_sqrt_ps(xmm2);     // R = sqrt(XMXI^2 + YMYI^2)
xmm0 = _mm_div_ps(xmm0,xmm2); // DR1 = XMXI / R
xmm1 = _mm_div_ps(xmm1,xmm2); // DR2 = YMYI / R
xmm6 = _mm_set_ps1(BN[0]);    // BN1
xmm7 = _mm_set_ps1(BN[1]);    // BN2
xmm3 = _mm_mul_ps(xmm0,xmm6); // DR1 * BN1
xmm4 = _mm_mul_ps(xmm1,xmm7); // DR2 * BN2
xmm5 = _mm_mul_ps(xmm0,xmm1); // UL12 = DR1 * DR2
xmm3 = _mm_add_ps(xmm3,xmm4); // DRDN = DR1 * BN1 + DR2 * BN2
xmm6 = _mm_mul_ps(xmm6,xmm1); // DR2 * BN1
xmm7 = _mm_mul_ps(xmm7,xmm0); // DR1 * BN2
_mm_store_ps(ul12v4,xmm5);
xmm0 = _mm_mul_ps(xmm0,xmm0); // DR1 * DR1
xmm1 = _mm_mul_ps(xmm1,xmm1); // DR2 * DR2
xmm5 = _mm_add_ps(xmm5,xmm5); // 2 * DR1 * DR2
xmm7 = _mm_sub_ps(xmm7,xmm6); // DR1 * BN2 - DR2 * BN1
xmm4 = vmlsLn4(xmm2);         // log R
xmm5 = _mm_mul_ps(xmm5,xmm3); // 2 * DR1 * DR2 * DRDN
_mm_store_ps(ul11v4,xmm0);
_mm_store_ps(ul22v4,xmm1);
```

Using 4 integrations points, one can easily find that :

$$
\begin{aligned}
G_{11} &= UL^1_{11} * B^1_1 * W^1 + UL^2_{11} * B^2_1 * W^2 + UL^3_{11} * B^3_1 * W^3 + UL^4_{11} * B^4_1 * W^4 \\
G_{21} &= UL^1_{21} * B^1_1 * W^1 + UL^2_{21} * B^2_1 * W^2 + UL^3_{21} * B^3_1 * W^3 + UL^4_{21} * B^4_1 * W^4 \\
G_{12} &= UL^1_{12} * B^1_1 * W^1 + UL^2_{12} * B^2_1 * W^2 + UL^3_{12} * B^3_1 * W^3 + UL^4_{12} * B^4_1 * W^4 \\
G_{22} &= UL^1_{22} * B^1_1 * W^1 + UL^2_{22} * B^2_1 * W^2 + UL^3_{22} * B^3_1 * W^3 + UL^4_{22} * B^4_1 * W^4
\end{aligned}
$$

Since all values of **UL** are stored in vectors, it is quite simple to perform the multiplications of each value by the respective four values stored in B_1 and W. However, there is no SSE instruction to perform the sum of the elements of a vector needed in the computation of **G**.

Using the SSE unpack and move instructions, the values stored on four vectors can be reordered to obtain the same effect of a matrix transposition, although here the operations are performed on vectors. The `_mm_unpackhi_ps` and `_mm_unpacklo_ps` instructions select and interleave respectively the upper (`hi`) or lower (`lo`) two values from two vectors a and b. The `_mm_move_hl` instruction moves the upper two elements of b to the lower values (`hl`) while `_mm_move_lh_ps` moves the lower two elements of b to the upper values (`lh`) of the target vector.

Listing 2 shows how the SSE unpack and move instructions can be used in the computation of the first half of matrix **G**.

Listing 2. SSE matrix transposition

```
// computing G1
xmm7 = _mm_set_ps1(C);
xmm6 = _mm_set_ps1(C1);
xmm6 = _mm_mul_ps(xmm6,xmm7);
xmm5 = _mm_load_ps(B1Wv4);
xmm5 = _mm_mul_ps(xmm5,xmm6);
xmm0 = _mm_load_ps(ul11v4);
xmm1 = _mm_load_ps(ul22v4);
xmm2 = _mm_load_ps(ul12v4);
xmm3 = _mm_load_ps(ul12v4);
xmm0 = _mm_mul_ps(xmm0,xmm5);
xmm1 = _mm_mul_ps(xmm1,xmm5);
xmm2 = _mm_mul_ps(xmm2,xmm5);
xmm3 = _mm_mul_ps(xmm3,xmm5);
xmm4 = _mm_unpackhi_ps(xmm0,xmm1);
xmm5 = _mm_unpackhi_ps(xmm2,xmm3);
xmm6 = _mm_unpacklo_ps(xmm0,xmm1);
xmm7 = _mm_unpacklo_ps(xmm2,xmm3);
xmm0 = _mm_movelh_ps(xmm6,xmm7);
xmm1 = _mm_movehl_ps(xmm7,xmm6);
xmm2 = _mm_movelh_ps(xmm4,xmm5);
xmm3 = _mm_movehl_ps(xmm5,xmm4);
xmm0 = _mm_add_ps(xmm0,xmm2);
xmm1 = _mm_add_ps(xmm1,xmm3);
xmm0 = _mm_add_ps(xmm0,xmm1);
_mm_store_ps(v4G1,xmm0);
```

Most of the matrix operations performed by the application can be vectorized with compiler intrinsics. However, due to the limited number of SSE registers,

the vectorization of the code demands special care since several implementations of an algorithm are possible. The most efficient SSE algorithm can only be determined with the use of specialized tools and profiling procedures.

The most recent processors allow the parallel execution of up to three SSE instructions. Hence, an efficient implementation also depends of the optimal use of the vector functional units. The user should rely on the optimization routines available in the compiler to find a good implementations for an algorithm.

Finally, well-known optimization techniques, usually applied to scalar codes, can also be used in the implementation of vector algorithms in order to replace long latency instructions and to reduce data dependence. Data dependence is the major obstacle to automatic or manual vectorization of any algorithm. Even well-written programs enclose data dependencies due to the nature of the applications. High performance techniques are presented by the authors in previous works [8, 9] and will not be addressed here.

6 Results Summary

The Fortran and C/C++ implementations evaluated here runs on a SGI Altix XE 1200 cluster with 11 nodes, comprising 8 Quad-Core Xeon 2.66GHz (X5355) processors and 8 GB memory per node. The operating system is Linux SLES 10.1, ProPack 5 SP3 and the Intel Fortran 10 and Intel C/C++ 10 compilers are used.

The first case study corresponds to a square plate under biaxial load, as found in [1], with nodes distributed along the boundary. A second example is also taken from Telles [1] and refers to the problem of a cylindrical cavity under internal pressure. The schematic description of both problems is shown in figures 2 and 3, respectively.

Figure 2. A square plate under biaxial load

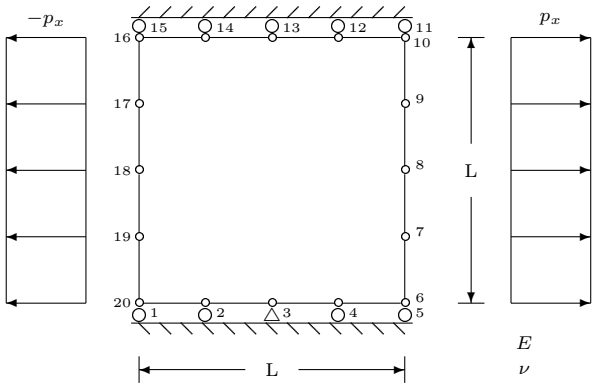


Figure 3. A cylindrical cavity under internal pressure

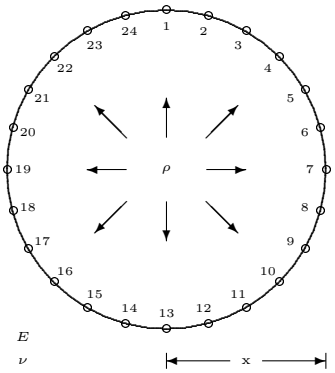


Table 1. Square plate - 10000 nodes

time (s)	Original	Autovectorization	SSE Intrinsics
real	45.822	32.918	12.880
user	1.184	0.956	1.160
sys	0.47	0.33	0.14

Table 2. Cylindrical cavity - 10000 nodes

time (s)	Original	Autovectorization	SSE Intrinsics
real	42.958	32.478	12.028
user	1.100	0.900	1.002
sys	0.44	0.33	0.10

The first column of tables 1 and 2 indicates the processing time of the original version of the code, with no vectorization. The second column refers to the autovectorized implementation while the third column shows the wall-clock time of the code vectorization with SSE intrinsics.

7 Conclusions

This paper introduces the Streaming SIMD Extensions (SSE), also known as multimedia instructions, and its application to engineering codes. The SSE instruction set enhances the Intel architectures with instructions that handle a set of floating-point values stored in vectors, simultaneously, instead of scalar variables. These vector operations can enhance the performance of modern processors significantly.

In the first part of the work [11] auto-vectorization techniques were presented. Here, explicit vector/SIMD instructions or compiler intrinsics are addressed in some detail and its use is demonstrated in a numerical application to solve two-dimensional elastostatic problems. The proposed implementation illustrates the basic concepts underlying SSE and provides guidelines to generate vector executables with C/C++ compiler intrinsics. The techniques presented are applied to a boundary element code but other methods can equally be addressed with the same techniques.

The results show a reduction in the runtime of 30% using auto-vectorization techniques while the implementation with SSE intrinsics yields a reduction of over 70% when compared to the original code.

SIMD extensions are currently found in most current processors, hence the knowledge of SIMD programming appears to be a decisive factor in the future of high performance computing [5, 6]. The implementation of boundary element codes on the STI Cell Broadband Engine processor using SIMD instructions has been presented in a previous work [12].

Intel microarchitectures, includes the SSE4 and the new AVX instruction set. The Intel Advanced Vector Extensions provide wider vector registers allowing more simultaneous floating point operations. Thus, the procedures introduced here arise as an additional and important optimization tool for numerical applications on today's and future processor architectures.

References

1. Brebbia CA, Telles JCF, Wrobel LC. Boundary elements techniques : theory and applications in engineering. Berlin: Springer Verlag; 1984.
2. Dongarra J et al. LAPACK users guide. 3rd ed. SIAM; 1999.
3. Bik AJC. Software vectorization handbook. Intelpress; 2004.
4. Gerber R, Bik AJC, Smith KB, Tian X. The software optimization cookbook. 2nd ed. Intel Press; 2006.
5. Patterson DA, Hennessy JL. Computer organization and design: the hardware/software interface, 3rd ed. revised. Elsevier-Morgan Kaufmann; 2007.

6. Hennessy JL, Patterson DA. Computer arquitecture: a quantitative approach, 4th ed. Elsevier-Morgan Kaufmann; 2007.
7. Flynn MJ. Very high-speed computing systems. Proc. IEEE. 1966. 54/12:1901-1909.
8. Cunha MTF, Telles JCF, Coutinho ALGA. On the parallelization of boundary element codes using standard and portable libraries. Engineering Analysis with Boundary Elements. 2004. 28/7:893-902. doi: 10.1016/j.enganabound.2004.02.002
9. Cunha MTF, Telles JCF, Coutinho ALGA. A portable implementation of a boundary element elastostatic code for Shared and Distributed Memory Systems. Advances in Engineering Software. 2004. 37/7:893-902. doi: 10.1016/j.advengsoft.2004.05.007
10. Cunha MTF, Telles JCF, Coutinho ALGA. Parallel boundary elements : a portable 3-D elastostatic implementation for shared memory systems. Lecture Notes in Computer Science. 2005. 3402:514-526.
11. Cunha MTF, Telles JCF, Ribeiro FLB. Streaming SIMD extensions applied to boundary element codes. Advances in Engineering Software. 2008. doi: 10.1016/j.advengsoft.2008.01.003
12. Cunha MTF, Telles JCF, Coutinho ALGA. On the implementation of boundary element engineering codes on the Cell Broadband Engine. Proceedings of the 8th International Meeting High Performance Computing for Computational Science. VECPAR 2008.