Cunha, Coutinho & Telles

Federal University of Rio de Janeiro

Vectorization of Engineering Codes with Multimedia Instructions

presented by

Manoel T. F. Cunha

Outline

- This work addresses the vectorization of scientific and engineering codes using Streaming SIMD Extensions (SSE) also known as multimedia instructions.

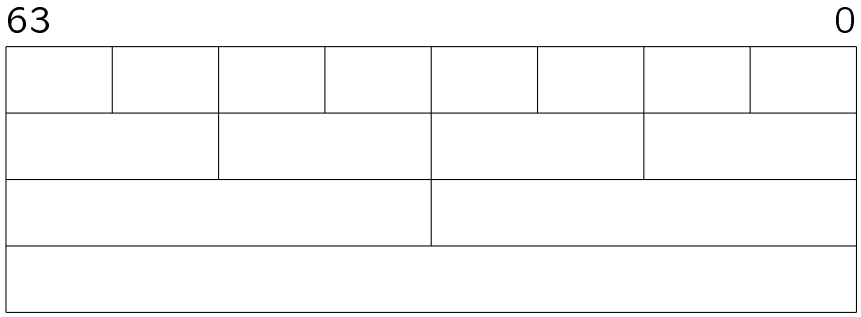
- The vector implementation of an engineering program for the solution of bidimensional elastostatic problems with the boundary element method is presented.

- Well known techniques used for decades.
- Only one vector computer is now on top500.org.
- Benefits of vector computing were not forgotten !

- SIMD multimedia instruction sets :
  - Intel MMX/SSE, AMD 3D-Now, Apple Altivec, STI Cell BE.
- Highly suited to handle computing intensive tasks :
  - audio/video compression, encryption/decryption, digital signal and data stream processing.
- Not so well explored for engineering and scientific computing !

Matrix Math Extensions

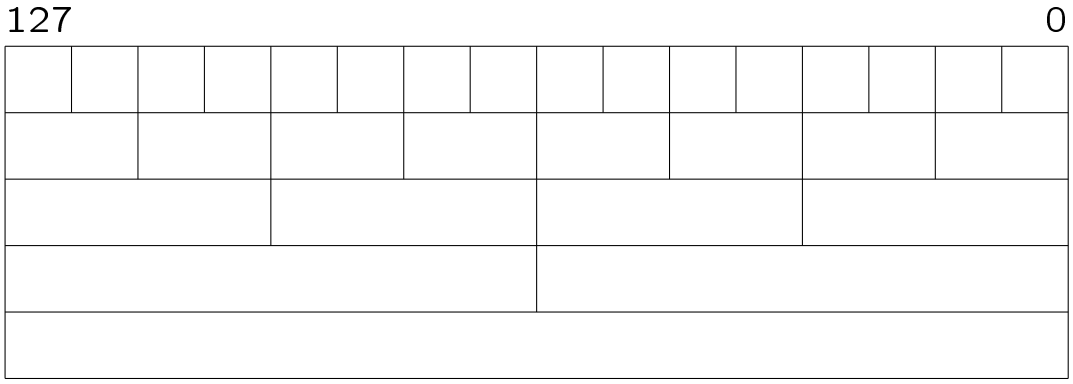
- A technology introduced with Intel Pentium processors. P5. Jan-97.
- Intended to enhance the performance of multimedia applications.
- 56 MMX instructions can process simultaneously 64-bit data sets of integer type :



8×8 bits (byte)
4×16 bits (integer)
2×32 bits (long)
1×64 bits

Streaming SIMD Extensions

- **SSE** : Pentium III processors. P6. Feb/99.
  - 70 instructions : 12 integer, 50 single precision FP, 8 cache control/data prefetch.
  - 8 128-bit dedicated registers allow to execute SSE and FP simultaneously.
- **SSE2** : Pentium 4 processors. NetBurst. Dec/00.
  - 144 instructions. Added support for double-precision FP operations.
- **SSE3** : +13 instructions and support for complex FP operations. Feb/04.
  - Supplemental SSE3 : Core 2 Duo/Xeon 5100. Core. Jul/06. +32 instructions.
- **SSE4** : +54 instructions.



16 × 8 bits (byte)
8 × 16 bits (integer)
4 × 32 bits (single)
2 × 64 bits (double)
1 × 128 bits (complex)

- The easiest and quickest way to implement SSE is auto-vectorization.
- Fortran and C/C++ compilers offer SSE compiler options to generate vectorized 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.
- Like other optimizations, the compiler ability to generate vector code automatically is restrained by a number of factors : problem complexity, bad programming techniques, etc.

Intel Fortran : `-arch SSE -vec-report3`

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

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.

G = 0. ! @ line 445

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

- Complex codes will pose greater restrictions for auto-vectorization.
- **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.

Compiler Intrinsics

- Fortran users depend on compiler ability to generate SSE executables.
- C/C++ : explicit vector functions inserted in the code.
  - More control of the vectorization process.
- Compiler intrinsics provide a new set of vector functions and data types.
  - 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.

- New data type to allocate 128-bit memory blocks that cannot be accessed directly.

```
__m128 xmm0,xmm1,xmm2,xmm3;
```

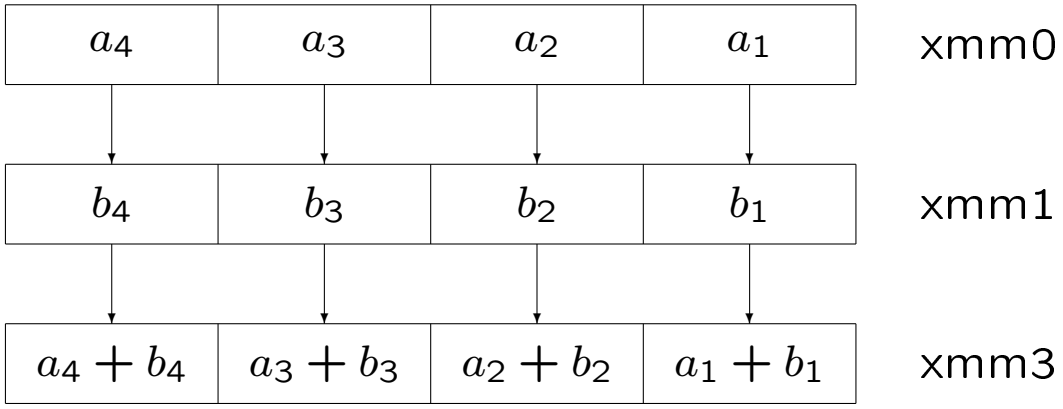
- Intrinsics must be used to initialize data and move it to/from F.P. arrays.

```
xmm2 = _mm_set_ps(1.,0.,0.,1);
xmm3 = _mm_set_ps1(-C1);
xmm1 = _mm_load_ps(work);
_mm_store_ps(ul,xmm0);
```

- F.P. arrays are required to be aligned in 16-byte boundaries.

```
__declspec(align(16)) float work[4],ul[4];
```

- SSE instructions perform one operation on two sets of four S.P. F.P. values.



Boundary Element Method

- A technique for the numerical solution of partial differential equations subjected to initial and boundary conditions.
  - Only the boundary is discretized
  - Integration over all the elements
  - Reduced and dense system of equations

- Differential Equation :

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

- Boundary Integral Equation :

$$c_{ij}\, u_j + \int_\Gamma p^*_{ij}\, u_j\, d\Gamma = \int_\Gamma u^*_{ij}\, p_j\, d\Gamma \tag{2}$$

- Discretized Equation :

$$c_{ij}\, u_j + \sum_{k=1}^{N} \int_{\Gamma_k} p^*_{ij}\, u_j\, d\Gamma = \sum_{k=1}^{N} \int_{\Gamma_k} u^*_{ij}\, p_j\, d\Gamma \tag{3}$$

- Linear System of Equations :

$$\mathbf{A}\,\mathbf{x} = \mathbf{f} \tag{4}$$

- The solver is usually the most time consuming routine.
  - Addressed by high performance libraries : LAPACK ...
- The generation of equations system and internal points computation :
  - Together can take the most part of the processing time.
  - Usually implemented by researchers.
  - Greatly limit the speedup if not properly optimized.

- An equation system is generated and the influence coefficients for non-singular elements are evaluated using Gauss integration.

- A set of small matrix operations are computed. Ex.:

$$\begin{bmatrix} UL_{11} & UL_{12} \\ UL_{21} & UL_{22} \end{bmatrix} = -C1 \left[\begin{bmatrix} C2logR & 0 \\ 0 & C2logR \end{bmatrix} - \begin{bmatrix} DR_{11} & DR_{12} \\ DR_{21} & DR_{22} \end{bmatrix} \right]$$

- Those 2x2 matrices can be converted into vectors of size 4 and matrix operations can be performed with SIMD instructions.

- Some intermediate operations to be executed with scalar instructions.

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

```
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);
```

- In the original algorithm, those matrices are computed from 2 to 6 times, accordingly to the number of Gauss integration points defined by the chosen integration rule.

- Alternatively, a fully vector implementation of the matrix computation can be achieved by using 4 Gauss integration points and evaluating all four values of each coefficient at once, including the intermediate values.

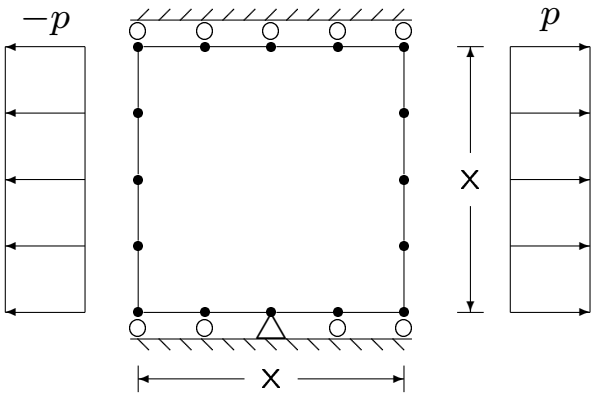
- Initially using two-dimensional arrays and executed with scalar instructions, the computation - including the intermediate operations - are now performed on vectors and four values are evaluated in each operation.

- Most of the operations can be performed with basic memory and arithmetic SIMD instructions.

```
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);
```

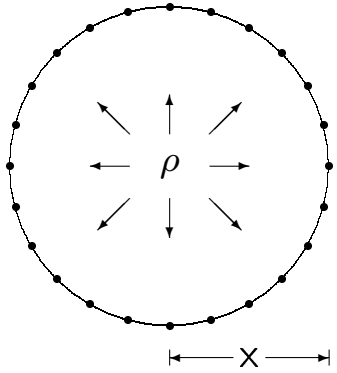
Results

SGI Altix XE 1200 cluster. 11 nodes, 8 Quad-Core Intel Xeon 2.66GHz (X5355) processors and 8 GB memory per node. Linux SLES 10.1, Intel Fortran and C/C++ 10 compilers.



A square plate under bi-axial load

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



A cylindrical cavity under internal pressure

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

Conclusions

- 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.

- The techniques presented are applied to a boundary element code but other methods can equally be addressed with the same techniques.

- Multimedia instructions sets are found in most current processors, hence vector programming seems to be a decisive tool in the future of hpc.

- Intel Advanced Vector Extensions (Intel AVX) is here !
  - A 256 bit instruction set extension designed for floating point intensive applications.
  - 80% of top500.org computers are using Intel processors !

Bibliography

- Bik AJC. *Software Vectorization Handbook*. Intelpress. 2004.

- 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

- Cunha MTF, Telles JCF, Coutinho ALGA. *On the implementation of boundary element engineering codes on the Cell Broadband Engine*. Lecture Notes in Computer Science. 2008. doi: 10.1007/978-3-540-92859-1

Cunha, Coutinho & Telles

Federal University of Rio de Janeiro

Vectorization of Engineering Codes with Multimedia Instructions

presented by

Manoel T. F. Cunha

www.ccad.ufpr.br/manoel

manoel@ufpr.br