Applying Vectorization Techniques for B-Spline Surface Evaluation

Roman Lygin, Sr. Technical Consulting Engineer, Intel Corporation roman.lygin@intel.com
Raul Akhmetshin, Student, Ufa State Aviation Technical University raul210192@mail.ru

Abstract

In this paper we analyze relevance of vectorization for evaluation of Non-Uniform Rational B-Spline (NURBS) surfaces broadly used in Computer Aided Design (CAD) industry to describe free-form surfaces. NURBS evaluation (i.e. computation of surface 3D points and derivatives for the given u, v parameters) is a core component of numerous CAD algorithms and can have a significant performance impact. We achieved up to 5.8x speedup using Intel® Advanced Vector Extensions (Intel® AVX) instructions generated by Intel® C/C++ compiler, and up to 16x speedup including minor algorithmic refactoring, which demonstrates high potential offered by the vectorization technique to NURBS evaluation.

Introduction

Vectorization, or Single Instruction Multiple Data (SIMD), is a parallelization technique available on modern computer processors, which allows to apply the same computational operation (e.g. addition or multiplication) to several data elements at once. For example, on a processor with a 128 bit register a single addition operation can add 4 pairs of integers (each takes 32 bits) or 2 pairs of doubles (64 bits each). With the help of vectorization one can speed up computations due to reduced time required to process the same data sets. SIMD was introduced with Intel® Architecture Processors way back in 1990es, with MMX™ technology as its first generation.

In this paper we analyze relevance of vectorization for evaluation of NURBS surfaces [1]. NURBS is a standard method used in CAD industry to describe free-form surfaces, e.g. car bodies, ship hulls, aircraft wings, consumer products and so on. Examples of 3D models (from [3]) containing NURBS surfaces are given on Fig.1:

![Fig.1. Examples of 3D models containing NURBS surfaces](image)

NURBS evaluation (i.e. a computation of surface 3D points and derivatives) is a core component of numerous CAD algorithms. For instance, in our simplified tests we observed hundreds of millions points and derivatives computed. Thus suboptimal implementation can easily become an application bottleneck.
The results presented in this paper stem from a Summer School student project arranged in 2013 by Intel Corporation. The project was focused on a research of applying the vectorization technique on Intel processors with the help of Intel Parallel Studio XE software suite [2].

**NURBS definition and evaluation**

A NURBS surface is defined by its u and v degrees, a grid of weighed control points, \{u_i\} and \{v_j\} knot vectors ([1], [9]). The fig.2 (from [4]) shows an example of a NURBS surface with its control points. In the case of distinct weights the surface is called rational, otherwise polynomial.

The knot vectors define a parametrization of the surface.

Each 3D point on the surface is influenced by the near control points and corresponding knots in the knot vector, determined in accordance with B-Spline degree.

By definition, surface point coordinates are determined with the help of basis functions defined at corresponding knot range, as given below for non-rational and rational cases respectively:

\[
S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} h_{ij} P_{ij} N_i(u) N_j(v), \quad u \in [u_k; u_{k+1}], \; v \in [v_l; v_{l+1}]
\]

\[
S(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} h_{ij} P_{ij} N_i(u) N_j(v)}{\sum_{i=0}^{n} \sum_{j=0}^{m} h_{ij} N_i(u) N_j(v)}, \quad u \in [u_k; u_{k+1}], \; v \in [v_l; v_{l+1}]
\]

- \(P_{ij}\), \(h_{ij}\) = control points with their weights;
- \(N_i(u), N_j(v)\) = basis functions;
- \(k\) – is u degree; \(l\) – is v degree.

\[
N_{ik}(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_{i,k-1}(u) + \frac{u_{i+k} - u_i}{u_{i+k} - u_{i+1}} N_{i+1,k-1}(u),
\]

\[
N_{io}(u) = \begin{cases} 1, & \text{for } u_i \leq u < u_{i+1}, \\ 0, & \text{otherwise} \end{cases}
\]

There are different computational algorithms to efficiently evaluate coordinates and derivatives of NURBS. One of these, the Boehm’s algorithm [5] is based on using Taylor series expansion for each knot span \([u_i; u_{i+1}] \times [v_j; v_{j+1}]\).

Given that for each knot span the Taylor series has a finite number of non-null coefficients, it has a polynomial form. The coefficients of respective polynomials are stored in the form of a matrix of the following kind (for details please refer to [5]):

\[
\begin{pmatrix}
X^* & Y^* & Z^* & W^* & \ldots & \frac{\partial^k X^*}{\partial u^k} & \frac{\partial^k Y^*}{\partial u^k} & \frac{\partial^k Z^*}{\partial u^k} & \frac{\partial^k W^*}{\partial u^k} \\
\frac{\partial^l X^*}{\partial v^l} & \frac{\partial^l Y^*}{\partial v^l} & \frac{\partial^l Z^*}{\partial v^l} & \frac{\partial^l W^*}{\partial v^l} & \ldots & \frac{\partial^{k+l} X^*}{\partial u^k \partial v^l} & \frac{\partial^{k+l} Y^*}{\partial u^k \partial v^l} & \frac{\partial^{k+l} Z^*}{\partial u^k \partial v^l} & \frac{\partial^{k+l} W^*}{\partial u^k \partial v^l}
\end{pmatrix}
\]
To compute the resulting value of each polynomial the Horner method [6] is used. The method involves repeated computations of sums and multiplications using the above matrix. The Horner method is briefly explained below:

Let the polynomial \( p(x) = \sum_{i=0}^{n} a_i x^i \) be defined as follows:

\[
p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 \ldots + a_n x^n = a_0 + x(a_1 + a_2 x \ldots + a_n x^{n-1})
\]

\[
= a_0 + x(a_1 + x(a_2 \ldots + a_n x^{n-2})) = \ldots,
\]

Using the above form a value of the polynomial in the point \( x = x^* \) will be computed as follows:

\[
b_n := a_n
\]

\[
b_{n-1} := a_{n-1} + b_n x^*
\]

\[
\vdots
\]

\[
p(x^*) = b_0 := a_0 + b_1 x^*
\]

Since we compute values and derivatives at once, such polynomials are computed across the entire matrix of coefficients.

**Reference implementation**

As a reference we chose Open CASCADE Technology (OCCT) [3], an open source C++ library for 3D modeling. OCCT contains data structures to represent NURBS surfaces as well as multiple surface and solid modeling algorithms (sweeping, lofting, Boolean operations, filleting and chamfering, etc). OCCT uses double precision floating point to represent coordinates and parameters, and to perform respective computations.

OCCT implementation of B-Spline surface evaluation follows the workflow outlined in [7]. The workflow uses the Boehm’s algorithm [5] to make a Taylor series expansion, and then uses the Horner method [6] to compute resulting values of each respective polynomial. The Horner method, as explained below became our optimization target.

**Test scenarios and environment**

We chose the surface-surface intersection algorithm which underlies various higher-level algorithms inside OCCT. For workloads we created 8 pairs of NURBS surfaces, both polynomial and rational, of degrees up to 5 and of up to 9x9 control points. Some of these test surfaces are shown on fig.3:

![Fig.3 Examples of test NURBS surfaces](image)
Open CASCADE has been compiled with the above Intel compiler using Visual Studio projects shipped with OCCT, and converted to use Intel compiler. The floating model option has been set to \textit{/fp:fast}.

Each test scenario (intersection of two NURBS surfaces) was executed 100 times in one application session, to ensure reasonable elapsed time (from a few seconds to a few dozens).

**Hotspot analysis**

To estimate contribution of NURBS evaluation into entire workload we used Intel® VTune™ Amplifier XE, which is part of the Intel® Parallel Studio XE suite, targeted for performance analysis and tuning. As shown on fig.3, evaluation of the 1\textsuperscript{st} derivative (method \texttt{Geom_BSplineSurface::D1()}) takes about 47\% of CPU time. Evaluation of points (\texttt{Geom_BSplineSurface::D0()}) not shown on the screenshot) took about 4\%.

Other workloads have contributions of the same order, of a few dozens per cent.

![Fig.3. Evaluation of the 1st derivative takes 47.2\% of time.](image)

\texttt{Geom_BSplineSurface::D1()} makes a few nested calls, and the method \texttt{PLib::EvalPolynomial()}, which performs the Horner method, has the largest Self time (where “Self time” is a total time of a function call excluding times of its callees) (see fig.4):

![Fig.4.](image)
Fig. 4. Evaluation of polynomial has the largest self time and contributes 57.7% into the computation of the 1st derivative.

We focused our efforts on optimization of the PLib::EvalPolynomial() with the help of SIMD-parallelism.

Measurement methodology
We measured total elapsed time of the intersection algorithm and total number of tick counts in PLib::EvalPolynomial(). We could not accurately measure elapsed time of the latter given that each call of PLib::EvalPolynomial() took only about 300 processor cycles. For measuring such short functions using QueryPerformanceCounter() (which is part of Windows* API) is impossible due to its relatively high intrusiveness. Instead, we had to use the rdtsc counter that measures processor cycles.

To ensure more repeatable results, we had to turn off Intel SpeedStep® technology and Intel Turbo Boost Technology in the processor. This allowed us to get a repeatable number of tick counts within 0.05% range. Introduction of rdtsc did not noticeably affect the elapsed time: average overhead was 0.5%.

Initial baseline
Initial measurements are shown in Table 1. Each row of the table corresponds to a respective pair of test surfaces (described above and partially shown on fig.3). For each pair of test surfaces it specifies the number of ticks spent by PLib::EvalPolynomial() measured by the rdtsc counter and elapsed time of the intersection algorithm.

<table>
<thead>
<tr>
<th>Surfaces</th>
<th>EvalPolynomial(), ticks</th>
<th>Total elapsed time, seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A &amp; 1B</td>
<td>8840607854</td>
<td>9.22</td>
</tr>
<tr>
<td>2A &amp; 2B</td>
<td>30422647467</td>
<td>30.23</td>
</tr>
<tr>
<td>3A &amp; 3B</td>
<td>20316026623</td>
<td>29.40</td>
</tr>
<tr>
<td>4A &amp; 4B</td>
<td>19315444680</td>
<td>22.89</td>
</tr>
<tr>
<td>5A &amp; 5B</td>
<td>16116457851</td>
<td>15.01</td>
</tr>
<tr>
<td>6A &amp; 6B</td>
<td>63029709847</td>
<td>60.65</td>
</tr>
<tr>
<td>7A &amp; 7B</td>
<td>21861086712</td>
<td>30.62</td>
</tr>
<tr>
<td>8A &amp; 8B</td>
<td>35438533155</td>
<td>31.95</td>
</tr>
</tbody>
</table>

Table 1. Baseline performance data.

Step 1: Algorithmic optimization
Review of the source code of PLib::EvalPolynomial() revealed suboptimal implementation. We noticed two main issues.

First, as a rule, access to matrix elements in memory was not sequential. Access was done by columns instead of rows as demonstrated in the following code excerpt:

```cpp
Standard_Real *RA = &Results;
Standard_Real *valRA;
...
for (ii = LocalRequest ; ii > 0 ; ii--) {
  valRA = &RA[Index1];
  *valRA = Par * (*valRA) + ((Standard_Real)ii) * RA[Index2] ;
  Index1 -= 9;
  Index2 -= 9;
```
Non-unit stride access is slower comparing to unit stride one as hardware prefetching (loading a few adjacent memory elements upon first access to one of them) is wasted. For larger arrays non-unit stride access may cause greater amount of cache misses when data being accessed does not fit into a processor cache memory.

Our assumption is that this may result from legacy Fortran code that was rewritten to C/C++ without paying attention to different conventions about element indexing and possible performance hit.

Secondly, we noticed use of pointer arithmetic and indirections (such as *p++), like in the excerpt below:

```c
Standard_Real *tmpRA = RA;
Standard_Real *tmpPA = PA + DegreeDimension;
...
for (jj = Degree ; jj > 0 ; jj--) {
  tmpPA -= 3;
  tmpRA = RA;
  *tmpRA = Par * (*tmpRA) + (*tmpPA); tmpPA++; tmpRA++;
  *tmpRA = Par * (*tmpRA) + (*tmpPA); tmpPA++; tmpRA++;
  *tmpRA = Par * (*tmpRA) + (*tmpPA);
  tmpPA -= 2;
}
```

Pointer arithmetic usually makes it harder for a compiler to optimize loops as well as worsen code readability and understanding.

To address the above issues we had to refactor the function code. We enabled unit-stride memory access and replaced pointer arithmetic with indexed access (p[i]). Below is a typical example of loops after the above refactoring:

```c
for (j = 1; j < Degree; j++) {
  PA -= Dimension;
  for (k = 0; k < Dimension; k++)
    resD1[k] = Par * resD1[k] + resD0[k];
  for (k = 0; k < Dimension; k++)
    resD0[k] = Par * resD0[k] + PA[k];
}
```

At last, taking into account the fact that in reality the most frequent cases are computation of the 0th and 1st derivatives for B-Splines of lower degrees, we made such computation via particular cases using the switch operator. This allowed us to avoid loops with very small trip counts. Below is a code excerpt that demonstrates this:

```c
switch (DerivativeRequest) {
  case 0 : { ... }
  case 1 : {
    switch (Dimension) {
      case 4: { ... }
      case 8: {
        for (kk = 0; kk < 8; kk++)
          resD1[kk] = resD0[kk] = PA[kk];
        PA -= 8;
        for (kk = 0; kk < 8; kk++)
          resD0[kk] = Par * resD0[kk] + PA[kk];
      ...
    }
  ...
```
The results of performance measurements of this refactored code are shown in Table 2 below:

<table>
<thead>
<tr>
<th>Surfaces</th>
<th>EvalPolynomial(), ticks</th>
<th>Speedup vs baseline</th>
<th>Total elapsed time, seconds</th>
<th>Speedup vs baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A &amp; 1B</td>
<td>3585122064</td>
<td>2,47x</td>
<td>7,56</td>
<td>1,22x</td>
</tr>
<tr>
<td>2A &amp; 2B</td>
<td>11149949049</td>
<td>2,73x</td>
<td>24,22</td>
<td>1,25x</td>
</tr>
<tr>
<td>3A &amp; 3B</td>
<td>9161768393</td>
<td>2,22x</td>
<td>25,56</td>
<td>1,15x</td>
</tr>
<tr>
<td>4A &amp; 4B</td>
<td>5861622127</td>
<td>3,30x</td>
<td>18,71</td>
<td>1,22x</td>
</tr>
<tr>
<td>5A &amp; 5B</td>
<td>6669842255</td>
<td>2,42x</td>
<td>12,17</td>
<td>1,23x</td>
</tr>
<tr>
<td>6A &amp; 6B</td>
<td>24997739703</td>
<td>2,52x</td>
<td>48,86</td>
<td>1,24x</td>
</tr>
<tr>
<td>7A &amp; 7B</td>
<td>9864809126</td>
<td>2,22x</td>
<td>27,25</td>
<td>1,12x</td>
</tr>
<tr>
<td>8A &amp; 8B</td>
<td>12699874345</td>
<td>2,79x</td>
<td>25,03</td>
<td>1,28x</td>
</tr>
<tr>
<td>Geomean</td>
<td>2,56x</td>
<td></td>
<td>1,21x</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Performance data after algorithm refactoring

Thus, average speed up of the EvalPolynomial() was 2.56x and of the total time – 1.21x.

It should be understood that as we further improved EvalPolynomial() its contribution into total time diminished, and the total time was reducing much slower than EvalPolynomial() itself.

**Step 2: Introducing vectorization**

Refactoring made at step 1 made a foundation to introduce vectorization. However, the compiler itself could not auto-vectorize the series of the loops given above. This is because the compiler has to apply conservative decision-making when deciding in favor of or against auto-vectorization. It is the developer’s responsibility to ensure fulfillment of vectorization prerequisites and/or to hint the compiler that vectorization would be safe to apply. [8] gives a good summary of prerequisites and hints, of which the following would apply in our case:

- Ensuring memory alignment
- Guaranteeing no pointer aliasing
- Hinting with pragmas

Let us consider each of these.

**Memory alignment**

To enable the compiler to apply vectorization, memory address must be aligned by 16 bytes (for SSE instructions) or by 32 bytes (for AVX instructions).

To achieve that, we had to modify Geom_BSplineSurface class where it allocates memory for NURBS coefficient computation.

Moreover, we reworked interim data structures to contain tuples \(\{x,y,z,w\}\), instead of separate coordinate triplets \(\{x,y,z\}\) and arrays of weights \(\{w\}\). This enables better data locality and allows us to avoid 2 extra calls of EvalPolynomial() for rational B-splines. For polynomial B-Splines this extra element \(W\) comes at no cost as due to data alignment this memory was padded anyway.
No pointer aliasing
By default, the compiler has to apply conservative assumptions regarding memory locations. Therefore it has to assume the worst case scenario even in the simple loops like the following one:

```c
for (k = 0; k < 4; k++)
    resD1[k] = Par * resD1[k] + resD0[k];
```

The compiler has to assume that arrays resD0 and resD1 may overlap in memory and thus there is a data dependency, e.g. resD1[k] is the same element as resD0[k-1]. With such an assumption it may not apply vectorization.

We know that there is no data overlap inside each loop, so we can communicate that to the compiler using the restrict keyword. The restrict keyword tells the compiler that pointers (of the same type) annotated by this keyword target disjunctive memory ranges. Using that keyword is only allowed if this holds true, which is the case in our example:

```c
Standard_Real * __restrict resD0 = &Results;
Standard_Real * __restrict resD1 = &Results + Dimension;
```

Hinting with pragmas
At last, once we have fulfilled prerequisites, we can safely hint the compiler that the loops can be vectorized adding pragmas as follows:

```c
#pragma vector aligned
for (k = 0; k < 4; k++)
    resD1[k] = Par * resD1[k] + resD0[k];

#pragma vector aligned
for (k = 0; k < 4; k++)
    resD0[k] = Par * resD0[k] + PA[k];
```

The above syntax communicates to the compiler that the loops can be vectorized using aligned memory access (as we already ensured alignment above). Intel compiler offers different syntaxes, including compile-time assertions (“#pragma vector always assert”). The latter can help to ensure that the code remains vectorized over time as newer compiler versions are used, modifications are introduced into the source code.

Compiler options /Qopt-report or /Qvec-report allow diagnosing successful or unsuccessful vectorized loops as well as failure reasons.

Once we have vectorized the loops in EvalPolynomial(), we tested three instruction sets – Intel SSE2, Intel AVX and Intel AVX2 – available on modern Intel processors and compatible non-Intel processors. To take advantage of vector registers, we created several particular cases for the matrix sizes – 4, 8, 12, 16, 20 – that will be used for lower B-Spline degrees (which are, to be reminded, the most frequent in real world).

To see the effects of different targeted architectures, we will analyze the following loop:

```c
#pragma vector aligned
for (k = 0; k < 16; k++)
    resD0[k] = Par * resD0[k] + PA[k];
```
**Step2A: Vectorization with SSE2 instructions**

Intel SSE2 (Intel Streaming SIMD Extensions 2) instructions have been available since Pentium® 4 processor, and thus are available on all mainstream desktop and laptop computers. Vector instructions operate on vector registers (named xmm) that have a length of 128 bit.

Intel SSE2 is always used when building for 64-bit platforms. For 32-bit platforms, the /QxSSE2 compiler option can be used to specify the optimization target.

Compiled with Intel® SSE2 instructions, the above test loop has been converted into the following assembly code:

```
movaps xmm15, xmmword ptr [rsp+0x20]
mulpd xmm9, xmm15
addpd xmm9, xmm2
movaps xmmword ptr [rbp+r12*8], xmm9
mulpd xmm10, xmm15
addpd xmm10, xmm3
movaps xmmword ptr [rsp+0xe0], xmm10
movaps xmmword ptr [rbp+r12*8+0x10], xmm10
movaps xmm10, xmmword ptr [rsp+0xd0]
mulpd xmm10, xmm15
addpd xmm10, xmm4
movaps xmmword ptr [rbp+r12*8+0x20], xmm10
mulpd xmm11, xmm15
addpd xmm11, xmm5
movaps xmmword ptr [rbp+r12*8+0x30], xmm11
[...]
```

The assembly code shows that the compiler fully unrolled the loop and at each iteration it processed 2 doubles at once (2 doubles fit into one 128 bit xmm register). So, it took 8 repeated combinations of movements, additions and multiplications (movaps, addpd, mulpd, movaps) to process entire original loop.

The following table shows performance data for the vectorized code using Intel SSE2 instructions:

<table>
<thead>
<tr>
<th>Surfaces</th>
<th>EvalPolynomial(), ticks</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
<th>Total elapsed time, seconds</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A &amp; 1B</td>
<td>1163634988</td>
<td>3,08x</td>
<td>7,60x</td>
<td>6,71</td>
<td>1,13x</td>
<td>1,38x</td>
</tr>
<tr>
<td>2A &amp; 2B</td>
<td>5965667836</td>
<td>1,87x</td>
<td>5,10x</td>
<td>23,23</td>
<td>1,04x</td>
<td>1,30x</td>
</tr>
<tr>
<td>3A &amp; 3B</td>
<td>2762891981</td>
<td>3,32x</td>
<td>7,35x</td>
<td>20,50</td>
<td>1,25x</td>
<td>1,43x</td>
</tr>
<tr>
<td>4A &amp; 4B</td>
<td>3131042272</td>
<td>1,87x</td>
<td>6,17x</td>
<td>18,33</td>
<td>1,02x</td>
<td>1,25x</td>
</tr>
<tr>
<td>5A &amp; 5B5A &amp; 5B</td>
<td>1783544660</td>
<td>3,74x</td>
<td>9,04x</td>
<td>10,65</td>
<td>1,14x</td>
<td>1,41x</td>
</tr>
<tr>
<td>6A &amp; 6B</td>
<td>8592665277</td>
<td>2,91x</td>
<td>7,34x</td>
<td>44,40</td>
<td>1,10x</td>
<td>1,37x</td>
</tr>
<tr>
<td>7A &amp; 7B</td>
<td>3014640664</td>
<td>3,27x</td>
<td>7,25x</td>
<td>24,52</td>
<td>1,11x</td>
<td>1,25x</td>
</tr>
<tr>
<td>8A &amp; 8B</td>
<td>4179043476</td>
<td>3,04x</td>
<td>8,48x</td>
<td>22,40</td>
<td>1,12x</td>
<td>1,43x</td>
</tr>
</tbody>
</table>

**Geomean** 2,81x 7,19x 1,11x 1,35x

Table 3. Performance data after applying vectorization using SSE2 instructions

Thus, for EvalPolynomial() Intel SSE2 vectorization gave 2.81x average speed-up over previous version (algorithm refactoring) and 7.19x vs initial baseline. The total time improved 1.11 and 1.35 times respectively.
Step2B: Vectorization with AVX instructions

Intel AVX (Intel Advanced Vector Extensions) instruction set was introduced with the 2nd generation of Intel® Core™ processors (previously codenamed Sandy Bridge). The vector registers (named ymm) have a length of 256 bit and thus is able to contain up to 4 doubles at once.

The compiler offers /QxAVX option to specify the AVX optimization target.

Our loop compiled with Intel AVX instructions has been converted into the following code:

```assembly
vmulpd ymm8, ymm4, ymm8
vaddpd ymm8, ymm8, ymm3
vmovupd ymmword ptr [rsi+rbp*8], ymm8
vmulpd ymm7, ymm4, ymm7
vaddpd ymm7, ymm7, ymm2
vmovupd ymmword ptr [rsi+rbp*8+0x20], ymm7
vmulpd ymm5, ymm4, ymm5
vaddpd ymm5, ymm5, ymm1
vmovupd ymmword ptr [rsi+rbp*8+0x40], ymm5
vmulpd ymm9, ymm4, ymm9
vaddpd ymm9, ymm9, ymm0
vmovupd ymmword ptr [rsi+rbp*8+0x60], ymm9
```

Similar to Intel SSE2, the Intel AVX code explicitly unrolled the loop into a sequence of moves, adds, multiples, moves. However, as the ymm registers are capable to contain as twice data as Intel SSE2, now it only takes 4 iterations to complete the job (vs 8 for Intel SSE2).

Table 4 contains performance data for Intel AVX optimized code:

<table>
<thead>
<tr>
<th>Surfaces</th>
<th>EvalPolynomial(), ticks</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
<th>Total elapsed time, seconds</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A &amp; 1B</td>
<td>825266438</td>
<td>4,34x</td>
<td>10,71x</td>
<td>6,61</td>
<td>1,14x</td>
<td>1,40x</td>
</tr>
<tr>
<td>2A &amp; 2B</td>
<td>4426314500</td>
<td>2,52x</td>
<td>6,87x</td>
<td>22,63</td>
<td>1,07x</td>
<td>1,34x</td>
</tr>
<tr>
<td>3A &amp; 3B</td>
<td>2212996058</td>
<td>4,14x</td>
<td>9,18x</td>
<td>20,04</td>
<td>1,27x</td>
<td>1,47x</td>
</tr>
<tr>
<td>4A &amp; 4B</td>
<td>1811445170</td>
<td>3,24x</td>
<td>10,66x</td>
<td>17,09</td>
<td>1,10x</td>
<td>1,34x</td>
</tr>
<tr>
<td>5A &amp; 5B &amp; 5A &amp; 5B</td>
<td>1287805552</td>
<td>5,18x</td>
<td>12,51x</td>
<td>10,52</td>
<td>1,16x</td>
<td>1,43x</td>
</tr>
<tr>
<td>6A &amp; 6B</td>
<td>6387306940</td>
<td>3,91x</td>
<td>9,87x</td>
<td>42,71</td>
<td>1,14x</td>
<td>1,42x</td>
</tr>
<tr>
<td>7A &amp; 7B</td>
<td>2479663046</td>
<td>3,98x</td>
<td>8,82x</td>
<td>24,04</td>
<td>1,13x</td>
<td>1,27x</td>
</tr>
<tr>
<td>8A &amp; 8B</td>
<td>2440320202</td>
<td>5,20x</td>
<td>14,52x</td>
<td>21,36</td>
<td>1,17x</td>
<td>1,50x</td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td>3,97x</td>
<td>10,16x</td>
<td></td>
<td>1,15x</td>
<td>1,39x</td>
</tr>
<tr>
<td>vs SSE2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,41x</td>
<td>1,03x</td>
</tr>
</tbody>
</table>

Table 4. Performance data after applying vectorization using AVX instructions

Step2C: Vectorization with AVX2 instructions

The 4th generation of Intel® Core™ processors (previously codenamed Haswell) introduced an extension to the Intel AVX instruction set. In particular, the new instruction FMA (fused multiply-accumulate) was added, which allows performing multiplication and addition at once.

The /QxCORE-AVX2 compiler option specifies the respective optimization target.

Our test loop has now been compiled as follows:
vmovupd ymmword ptr [rdi+0x60], ymm7
vmovupd ymmword ptr [rdi+0x60], ymm2
vfadd213pd ymm7, ymm8, ymmword ptr [r8]
vfadd213pd ymm6, ymm8, ymmword ptr [r8+0x20]
vfadd213pd ymm5, ymm8, ymmword ptr [r8+0x40]
vfadd213pd ymm4, ymm8, ymmword ptr [r8+0x60]
vmovupd ymmword ptr [rdi+0x60], ymm4

Thus, the compiler has converted each source line into one assembly instruction, taking advantage of available FMA instruction.

Performance data for AVX2 instruction code is given below in table 5.

<table>
<thead>
<tr>
<th>Surfaces</th>
<th>EvalPolynomial(), ticks</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
<th>Total elapsed time, seconds</th>
<th>Speedup vs alg. opt.</th>
<th>Speedup vs baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A &amp; 1B</td>
<td>821391128</td>
<td>4,36x</td>
<td>10,76x</td>
<td>6,63</td>
<td>1,14x</td>
<td>1,39x</td>
</tr>
<tr>
<td>2A &amp; 2B</td>
<td>4273520209</td>
<td>2,61x</td>
<td>7,12x</td>
<td>22,61</td>
<td>1,07x</td>
<td>1,34x</td>
</tr>
<tr>
<td>3A &amp; 3B</td>
<td>2247262381</td>
<td>4,08x</td>
<td>9,04x</td>
<td>21,26</td>
<td>1,20x</td>
<td>1,38x</td>
</tr>
<tr>
<td>4A &amp; 4B</td>
<td>1648927480</td>
<td>3,55x</td>
<td>11,71x</td>
<td>16,90</td>
<td>1,11x</td>
<td>1,35x</td>
</tr>
<tr>
<td>5A &amp; 5B5A &amp; 5B</td>
<td>1248015314</td>
<td>5,34x</td>
<td>12,91x</td>
<td>10,57</td>
<td>1,15x</td>
<td>1,42x</td>
</tr>
<tr>
<td>6A &amp; 6B</td>
<td>6099783002</td>
<td>4,10x</td>
<td>10,33x</td>
<td>42,87</td>
<td>1,14x</td>
<td>1,41x</td>
</tr>
<tr>
<td>7A &amp; 7B</td>
<td>2419592765</td>
<td>4,08x</td>
<td>9,04x</td>
<td>23,95</td>
<td>1,14x</td>
<td>1,28x</td>
</tr>
<tr>
<td>8A &amp; 8B</td>
<td>2165618298</td>
<td>5,86x</td>
<td>16,36x</td>
<td>21,17</td>
<td>1,18x</td>
<td>1,51x</td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td>4,14x</td>
<td>10,61x</td>
<td></td>
<td>1,14x</td>
<td>1,38x</td>
</tr>
<tr>
<td>vs SSE2</td>
<td></td>
<td>1,47x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vs AVX</td>
<td></td>
<td>1,04x</td>
<td></td>
<td></td>
<td></td>
<td>0,99x</td>
</tr>
</tbody>
</table>

**Table 5. Performance data after applying vectorization using AVX instructions**

We need to note, however, that the compiler 14.0.1 generated the FMA instructions for one loop only in every pair of loops. Thus, this issue did not allow us to retrieve full performance potential, which resulted in collected performance data (note that the total time even increased by 0.5% although this is within measurements precision). This issue is planned to be addressed in one of the future product releases.

Given the above issue, it is difficult to make a solid conclusion on the FMA instruction benefits in this particular case but anyway almost all the tests did demonstrate some speed-up on EvalPolynomial() vs Intel AVX instruction set.

**Conclusion**

We have shown that applying vectorization can be extremely beneficial for computational algorithms used in NURBS evaluation, at least when the Horner method is used. We have observed up to 5.86x speed-up due to vectorization itself, and more than 16.36x speed-up including preliminary algorithmic refactoring. Using Intel software one can pinpoint and optimize code to benefit from vectorization.

Vectorization may require some prerequisites (memory alignment, hinting the compiler with the help of keywords and pragmas) yet these preparatory modifications will likely be very limited and local.

The next steps with respect to Open CASCADE Technology could be applying similar optimizations to other functions (PLib::RationalDerivative(), PLib::NoDerivativeEvalPolynomial()) that remain hotspots, optimizations for calculating the 2nd derivative, applying similar techniques to B-Spline curves, etc. The
research could also be extended towards an evaluation of multi-points at once, where both NURBS and elementary geometry could benefit. Multi-point evaluation can be extended to cover not only CPU, but also other computational resources (integrated graphics, Intel® Xeon Phi™ coprocessor).

Acknowledgment
The authors would like to thank Andrey Nikolaev, a technical lead in the Numerics software team at Intel and Andrey Betenev, an Open CASCADE Technology architect, for their participation in project reviews; Igor Vorobtsov and Georg Zitzlsberger, technical consulting engineers at Intel, for their recommendations about compiler optimizations and reviews of this paper.

References
[7]. B-Spline curve and surface evaluation, http://www.infogoaround.org/JBook/EvalBsp.html