Fast and Vectorizable Alternatives to Binary Search

Fabio Cannizzo*

25 May 2013

ver 1.0

1 Abstract

Given an array \(X\) of strictly ordered floating point numbers \(\{X_i\}_{i=0}^N\) and a set of floating point numbers \(\{Z_j\}_{j=1}^M\) belonging to the interval \([X_0, X_N)\), a common problem in numerical methods algorithms is to find the indices of the largest numbers in the array \(X\) which are smaller or equal than the numbers \(Z_j\).

This problem arises for instance in the context of piece-wise interpolation, where a domain \([X_0, X_N)\) is partitioned in sub-intervals \(\{[X_i, X_{i+1})\}_{i=0}^{N-1}\) and different interpolation functions \(g_i(x)\) are associated with each sub-interval. To compute the interpolated value for a number \(Z_j\), the index \(i\) of the sub-interval containing it needs to be resolved first.

The general solution to this problem is the binary search algorithm, which has complexity \(O(M \log_2 N)\). The classical and well known implementation of the algorithm requires a control flow branch, which incurs penalties on many CPU architectures, and is not vectorizable, i.e. it does not benefit from the vectorial capabilities of modern CPUs.

In some special cases, when either the \(X_i\) or the \(Z_j\) numbers exhibit particular patterns, more efficient algorithms are available. Examples are when the numbers \(X_i\) are equally spaced or when the numbers \(Z_j\) are sorted, where the problem can be solved with complexity \(O(M)\) and \(O(M + N)\) respectively. However no generic alternative exists.

This paper describes an improvement to the binary search algorithm, which avoid the control flow branch, thus making it generally faster. The complexity of the algorithm is still \(O(M \log_2 N)\), but performance improves by a proportionality factor \(\alpha \frac{2}{\sqrt{d}}\), where \(\alpha\) is a constant smaller than one associated with the performance gain due to the removal of the branch and, assuming perfect vectorization\(^1\), \(d\) is the number of floating point numbers which can be processed simultaneously\(^2\).

Next it proposes a new vectorizable algorithm based on a indexing technique, which reduces complexity of search operations to \(O(M)\), at the cost of introducing an initial overhead to compute the index and requiring extra memory for its storage. The algorithm has general applicability, but the relative magnitude of such extra costs, which are related to the layout of the numbers \(X_i\), in some particular cases might make its use not efficient.

Some benchmark test results using SSE2 instructions demonstrate that with \(N = 1025\) the proposed algorithm is about 26 times faster than the classical binary search in single precision and 15 times faster in double precision.\(^3\)

2 Problem Statement

Let:

- \(X\) be an array of \(N + 1\) floating point numbers sorted in ascending order, i.e. \(X_{i-1} < X_i, i = 1...N\). The

\(^*\)DISCLAIMER: Although Fabio Cannizzo is employed by Standard Chartered at the time this paper is written, this paper has been produced by Fabio Cannizzo in a personal capacity and Standard Chartered is not associated or responsible for its content in any way.

\(^1\)perfect vectorization means that all scalar instructions used in the algorithm have a vectorial equivalent

\(^2\)\(d\) depends on the chosen set of vectorial instructions and floating point representation (e.g. with SSE instructions in single precision \(d = 4\)).

\(^3\)test results are dependent on the hardware used
array \( X \) is assumed to be stored in a container supporting access by index in constant time. The numbers in \( X \) exhibit no special pattern.

- \( Z \) be an unordered array of \( M \) floating point numbers contained between the first and the last element of the array \( X \), i.e. \( Z_j \in [X_0, X_N], j = 0 \ldots M - 1 \).

For each number \( Z_j \) we want to find the index \( i \) of the largest element in the array \( X \) such that \( X_i \leq Z_j \). We denote such array of indexes as \( \{I_j\}_{j=0}^{M-1} \). Note that the exclusion of the last point \( X_N \) does not imply any loss of generality, as \( X_N \) could be simply replaced with the immediately next machine representable floating point number.

It is assumed that \( M \) is large and memory is not scarce, therefore it is worth investing up-front in a preliminary analysis of the structure of the array \( X \) and in the creation of an auxiliary data structure in order to later achieve superior computational performance when computing the indices \( \{I_j\}_{j=0}^{M-1} \).

## 3 Binary Search

### 3.1 Classical Binary Search

In absence of any special pattern of the values \( X_i \), the classical solution to this problem is the binary search algorithm, which in scalar version has complexity \( O(\log_2 N) \). A simple scalar pseudo-code implementation, as described in [1], is given in algorithm 1:

**Algorithm 1** Classical Binary Search (scalar implementation)

```plaintext
function BinarySearch(input: \{X\}_{i=0}^{N}, z, output: i)
    low ← 0
    high ← N
    while high - low > 1 do
        mid ← (low + high)/2
        if z < X_{mid} then
            high ← mid
        else
            low ← mid
        end if
    end while
    i ← low
end function
```

A close analysis of this implementation highlights the following weaknesses: the body of the loop contains a branch instruction and for an arbitrary \( z \) the chances for the boolean condition to be true are about 50\%, which makes branch prediction algorithms used by modern CPUs ineffective; the algorithm is not easily vectorizable because, for different \( z \) the boolean condition may resolve differently causing the program flow to take different code paths and requiring a different number of iterations for the loop to complete.

Because proper vectorization is not possible, a vectorial implementation is obtained trivially iterating on all elements of the array \( Z \) in steps of one element, with complexity \( O(M \log_2 N) \).

### 3.2 Binary Search Revisited

An improved variation of binary search is proposed in [2] and is based on the observations that some unnecessary extra iterations in the loop do not change the result and that the maximum possible number of potentially required iterations is \( \lceil \log_2 N \rceil \).

Let \( p \) be the number of bits necessary to represent the number \( N \), which is \( p = 1 + \lfloor \log_2 N \rfloor \), \( b_k \) be the binary value taken by the \( k \)-th bit of the sought index \( i \), \( c_k = 2^{k-1} \) and \( a_k = b_k c_k \), the sought index \( i \) has binary representation \( \sum_{k=1}^{p} a_k \).
The bits of the index can be resolved one by one starting from the highest order one as follows: first, if \( z \geq X_c \), then the \( p \)-th bit of the sought index \( i \) must be set, i.e. \( b_p = 1 \), next, if \( z \geq X_{a_p+c_p-1} \) then the \((p-1)\)-th bit of the index \( i \) must be set, i.e. \( b_{p-1} = 1 \), and so on, the values of the remaining bits are obtained iterating the procedure.

This approach fixes the number of iterations to \( p - 1 \), so no longer dependent \( z \), however, as the algorithm proceeds it can happen that the candidate index of the vector \( X \) exceeds the size of the vector, therefore, it requires a double boolean condition with short boolean evaluation, which makes it difficult to vectorize.

A scalar implementation of the algorithm, conceptually equivalent to what proposed in [2], is given in 2, where the pre-computed parameter \( P = c_p \).

### Algorithm 2 Binary Search Revisited (scalar implementation)

```plaintext
function BinarySearch(input: \{X\}_{i=0}^N, z, P, output: i)
    i ← 0
    k ← P
    repeat
        r ← i \| k
        if \( r < N \) \&\& \( z \geq X_r \) then
            i ← r
        end if
        k ← k/2
    until k = 0
end function
```

### 3.3 Vectorizable Binary Search

The first boolean condition in algorithm 2 can be avoided with a simple trick. Noting that \( Z_j < X_N \) for any \( j \), if the array \( X \) is extended to the right side by padding it with the last entry \( X_N \) up to a size equal to the largest possible index representable with \( p \) bits (i.e. \( Q = 2^p - 1 \)), the condition \( z \geq X_r \) would resolve to \texttt{false} for any \( r > N \) generated by the algorithm.

This yields algorithm 3, which has still complexity \( O(\log_2 N) \), but does not contain any code branch (only a conditional assignment, which can be implemented efficiently with modern CPUs instruction set) and is easily vectorizable.

### Algorithm 3 Vectorizable Binary Search (scalar implementation)

```plaintext
function BinarySearch(input: \{X\}_{i=0}^Q, z, P, output: i)
    i ← 0
    k ← P
    repeat
        r ← i \| k
        if \( z \geq X_r \) then
            i ← r
        end if
        k ← k/2
    until k = 0
end function
```

A vectorial implementation is obtained vectorizing the function and iterating on all elements of the array \( Z \) in steps of \( d \) elements, where \( d \) is the number of floating point numbers which can be processed simultaneously \(^4\). Note that, working with Intel SIMD instructions perfect vectorization is not achievable, because the indices contained in vector \( r \) are not contiguous, therefore the \( d \) elements of the vector \( X_r \) must be fetched from memory sequentially.

If memory is scarce, an alternative to extending and padding the array \( X \) is to take at every iteration the maximum between \( r \) and \( N \), which can be resolved by the compiler without branching.

\(^4\) \( d \) depends on the family of SIMD instruction used and on the chosen number representation, e.g. \texttt{single} or \texttt{double} precision.
4 Direct Search

A scalar algorithm with complexity $O(1)$ can be obtained via construction of an index which maps from an appropriately constructed new set of sub-intervals $\{[Y_t, Y_{t+1})\}_{t=0}^R$ to the indices of the original sub-intervals $\{[X_{i-1}, X_i)\}_{i=1}^N$.

The interval $[X_0, X_N)$ is partitioned into $R$ equally spaced sub-intervals $\{[Y_t, Y_{t+1})\}_{t=0}^R$ of length $H$, chosen to be just slightly smaller than the length of the smallest interval $\{[X_{i-1}, X_i)\}_{i=1}^N$:

\begin{align*}
R &= 1 + \left\lceil \frac{X_N - X_0}{\min \{X_i - X_{i-1}\}_{i=1}^N} \right\rceil \\
H &= \frac{X_N - X_0}{R} \tag{2}
\end{align*}

The extrema of the new sub-intervals are $\{Y_t = X_0 + t H\}_{t=0}^R$. Note that, since $H$ is strictly smaller than the smallest sub-interval $\{[X_{i-1}, X_i)\}_{i=1}^N$, it is guaranteed that every sub-interval $\{[Y_t, Y_{t+1})\}_{t=0}^R$ contains at most one single element of the array $X$.

Next an array $K$ of integer numbers $\{K_t\}_{t=0}^R$ is pre-computed, containing the indices $i$ of the smallest numbers in $X$ such that $X_i > Y_t$. These indices map the intervals $\{[Y_t, Y_{t+1})\}_{t=0}^R$ to the numbers in vector $X$, as illustrated in figure 1.

For a given number $z \in [X_0, X_N)$ the index $i$ such that $z \in [X_i, X_{i+1})$ can then be obtained as follows:

1. compute the index $t$ such that $z \in [Y_t, Y_{t+1})$ using formula

\begin{equation}
t = \left\lfloor \frac{z - X_0}{H} \right\rfloor \tag{3}
\end{equation}

2. read the correspondent index $i$ stored in the array $K$

3. verify if the numbers $z$ is to the right or to the left of $X_i$, and, if it is to the left, decrement the index $i$ by one.

The pseudocode is given in algorithm 4.

Similarly to algorithm 3, working with Intel SIMD instructions perfect vectorization is not achievable, as fetching from memory the relevant elements of $K_t$ and $X_i$ must be done sequentially.

4.1 Construction of the Index

Since $H$ is affected by rounding error and so are the subtraction and the division in formula (3), it is possible that the index $t$ is computed incorrectly.
Algorithm 4 Direct Search (scalar implementation)

```
function DirectSearch(input: \( \{X\}_{i=0}^{N} \), \( z \), \( \{K\}_{t=0}^{R} \), \( H \), output: \( i \))
    \( t \leftarrow \left\lfloor \frac{z-X_0}{H} \right\rfloor \)
    \( i \leftarrow K_t \)
    if \( z < X_i \) then
        \( i \leftarrow i - 1 \) \hfill \triangleright \text{conditional assignment}
    end if
end function
```

To overcome the problem the construction of the index \( K \) must take into account the rounding errors which may later occur using formula (3). This can be achieved by imposing that the index corresponding to an hypothetical set of numbers \( Z \) identical to the numbers \( \{X_i\}_{i=0}^{N-1} \) is resolved correctly, i.e. setting:

\[
K_t = i + 1, \text{ for all } t \text{ such that } t = \left\lfloor \frac{X_i - X_0}{H} \right\rfloor, \quad i = 1 \ldots N
\]

\[
K_t = K_{t+1}, \text{ for all other possible values of } t
\]

The pseudocode is given in algorithm 5.

Algorithm 5 Direct Search (index construction)

```
function BuildIndex(input: \( \{X\}_{i=0}^{N} \), \( H \), output: \( \{K\}_{t=0}^{R} \))
    \( K_R \leftarrow N \)
    \( b \leftarrow R - 1 \)
    \( i \leftarrow N - 2 \)
    while \( i > 0 \) do
        \( t \leftarrow \frac{X_i - X_0}{H} \)
        \( j \leftarrow \lfloor t \rfloor \)
        while \( b > j \) do
            \( K_b \leftarrow j + 1 \)
            \( b \leftarrow b - 1 \)
        end while
        if \( t = j \) then
            \( K_b \leftarrow j + 1 \)
            \( b \leftarrow b - 1 \)
        end if
        \( i \leftarrow i - 1 \)
    end while
end function
```

4.2 Initial Setup Cost

The initial setup cost due to the computation of the array \( K \) has cost \( O(R + N) \). The amount of extra memory required for storing the index is \( R B \), where \( B \) is the number of bytes required to store the maximum possible index \( N \), rounded to the next power of 2 to ensure efficient fetching from memory. Note that if \( R \) is very large or if memory is scarce the algorithm may not be efficient. This limitation is not discussed further in this paper, as it is not possible to derive a unique criteria for applicability, which would be dependent on many contingent factors, e.g. memory availability, hardware and compiler used.

5 Test Results

The table below compares the throughput of algorithm 1, 3 and 4. For each algorithm there are 4 possible implementations, where the set of instructions used is either scalar or vectorial with vectors of 128 bits (Intel SSE instruction set), and the used floating point representations are single and double precision.
For each floating point representation the results are expressed as the number of indices resolved per unit of time, relative to the respective scalar implementation of algorithm 1. The time to setup the index is excluded from the reported results.

Test results have been produced using the C++ source code available from github\(^5\), compiled for Windows 64-bits platform with Visual Studio 2013 on a machine with a Intel Sandy Bridge CPU.

The size of the vector $X$ used in the tests is 1025. The array $Z$ is stored aligned on a 32 bytes boundary, for efficient vectorial memory access. The ratio between the smallest and the largest interval is 480 and $R$ ranges between 22000 and 27000.

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th></th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
<td>SSE</td>
<td>scalar</td>
</tr>
<tr>
<td></td>
<td>d=1</td>
<td>d=4</td>
<td>d=1</td>
</tr>
<tr>
<td>algorithm 1</td>
<td>1.00</td>
<td>1.13</td>
<td>1.00</td>
</tr>
<tr>
<td>algorithm 3</td>
<td>2.81</td>
<td>3.95</td>
<td>2.80</td>
</tr>
<tr>
<td>algorithm 4</td>
<td>12.70</td>
<td>26.27</td>
<td>8.96</td>
</tr>
</tbody>
</table>

Note that the throughput increase factor due to vectorization is less than then $d$. This is because, using Intel SIMD instructions, as explained in previous sections, perfect vectorization is not achievable.

References


\(^5\)https://github.com/fabiocannizzo/fastbinarysearch.git