Filtering in a Multi-Threaded Environment

Jake Gunther

December 16, 2017

1 Introduction

Causal finite impulse response (FIR) filters perform convolution between an input signal $x[n]$ and an array of impulse response coefficients $h[n], n = 0, 1, \cdots, N-1$. The convolution formula is given by

$$y[n] = \sum_{k=n-N+1}^{n} h[n-k]x[k] \tag{1}$$

$$= \sum_{k=0}^{N-1} h[k]x[n-k], \tag{2}$$

where $y[n]$ is the filter output signal. If the input signal has length M and the impulse response has length N, then the output has length $L = M + N - 1$. Assuming a zero filter initial state, first $N-1$ outputs are the startup transient produced while the filter memory fills with input data. The last $N-1$ outputs are the ending transient while data empties out of the filter memory. The ending transient is produced by inputing $N-1$ zeros after the input data is exhausted. Assuming more input data than filter coefficients ($M \geq N$), the middle $M - N + 1$ outputs are valid outputs.

In the linear system of equations

$$\mathbf{y} = \mathbf{H}\mathbf{x}, \qquad \begin{array}{l} \mathbf{x} : M \times 1 \text{ vector} \\ \mathbf{H} : L \times M \text{ matrix} \\ \mathbf{y} : L \times 1 \text{ vector} \\ L = M + N - 1 \end{array}$$

the nth element of $\mathbf{y}$ is computed as the inner product of the nth row of $\mathbf{H}$ and the $\mathbf{x}$ vector. This inner product has the form

$$y[n] = \sum_{k=0}^{M-1} H[n,k]x[k]. \tag{3}$$

where $H[n,k]$ is the (n,k)th element of the matrix $\mathbf{H}$, i.e. n is the row index and k is the column index. Notice the similarity between matrix multiplication (3) and convolution (1). If $\mathbf{H}$ is the structured matrix defined by

$$H[n,k] = \begin{cases} h[n-k], & 0 \leq n-k \leq N-1 \\ 0, & \text{otherwise,} \end{cases} \tag{4}$$

then (3) and (1) are identical. Convolution may be implemented as multiplication of a structured (Toeplitz) matrix containing the impulse response and a vector containing the input signal! For your information, when the elements of a matrix are a function of the difference between the row and column indexes, i.e. $H[n,k] = h[n-k]$, as in (4), then the matrix is said to have Toeplitz structure. If, on the other hand, the elements of a matrix are a function of the sum of the row and column indexes, i.e. $H[n,k] = h[n+k]$, then the matrix is said to have Hankel structure. Toeplitz matrices are constant along parallels of the main diagonal. Hankel matrices are constant along parallels of the main anti-diagonal. The main diagonal are the matrix elements having equal row and column indexes, e.g. $H[i,i]$. The matrices below illustrate these ideas in the

setting of 4×3 matrices.

$$\begin{bmatrix} * & 0 & 0 \\ 0 & * & 0 \\ 0 & 0 & * \\ 0 & 0 & 0 \end{bmatrix} \quad \text{(main diagonal)}$$

$$\begin{bmatrix} 0 & 0 & * \\ 0 & * & 0 \\ * & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{(main anti-diagonal)}$$

$$\begin{bmatrix} c & b & a \\ d & c & b \\ e & d & c \\ f & e & d \end{bmatrix} \quad \text{(Toeplitz)}$$

$$\begin{bmatrix} a & b & c \\ b & c & d \\ c & d & e \\ d & e & f \end{bmatrix} \quad \text{(Hankel)}$$

In these examples, notice that each row (column) of a Toeplitz matrix is a right (downward) shift of the previous row (column). In a Hankel matrix each row (column) is a left (upward) shift of the previous row (column). The shift structure of Toeplitz and Hankel matrices mimics the way that data shifts through the memory of a filter. Therefore, these matrices are useful to model convolution and filtering.

To further illustrate the connection between convolution and matrix-vector multiplication, consider an example in which the filter impulse response has length $N = 4$, the input signal has length $M = 10$, and the convolution result has length $L = M + N - 1 = 13$:

$$\{h[0], h[1], h[2], h[3]\} \quad \text{impulse response,} \tag{5}$$

$$\{x[0], x[1], \cdots, x[9]\} \quad \text{input signal,} \tag{6}$$

$$\{y[0], y[1], \cdots, y[12]\} \quad \text{output signal.} \tag{7}$$

Consider the matrix-vector multiplication depicted in Fig. 1. The Toeplitz structure in $\mathbf{H}$ is evident and it agrees with (4). The transient output samples in $\mathbf{y}$ are colored red. The corresponding rows in $\mathbf{H}$ illustrate that the transient outputs are not valid filter outputs because they contain incomplete impulse responses, whereas all the other rows of $\mathbf{H}$ contain a complete set of impulse response coefficients. Furthermore, the shifting of the impulse response across the columns of $\mathbf{H}$ corresponds to the manner in the input signal shifts through filter memory.

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \end{bmatrix}}_{\mathbf{y}} = \underbrace{\begin{bmatrix} h_0 & & & & & & & & & \\ h_1 & h_0 & & & & & & & & \\ h_2 & h_1 & h_0 & & & & & & & \\ h_3 & h_2 & h_1 & h_0 & & & & & & \\ & h_3 & h_2 & h_1 & h_0 & & & & & \\ & & h_3 & h_2 & h_1 & h_0 & & & & \\ & & & h_3 & h_2 & h_1 & h_0 & & & \\ & & & & h_3 & h_2 & h_1 & h_0 & & \\ & & & & & h_3 & h_2 & h_1 & h_0 & \\ & & & & & & h_3 & h_2 & h_1 & h_0 \\ & & & & & & & h_3 & h_2 & h_1 \\ & & & & & & & & h_3 & h_2 \\ & & & & & & & & & h_3 \end{bmatrix}}_{\mathbf{H}} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}}_{\mathbf{x}}$$

Figure 1: Illustration of convolution as the product of a Toeplitz structured matrix and a vector: $\mathbf{y} = \mathbf{H}\mathbf{x}$.

The preceeding development grew out of the observation that the convolution formula in (1) was strikingly similar to the formula (3) for matrix-vector multiplication. The other formula for convolution (2) also resembles matrix-vector multiplication. However, in this case, the Toeplitz structured matrix is constructed from the input samples $x[n]$. To that end, define the $L \times N$ matrix $\mathbf{X}$ as follows

$$X[n,k] = \begin{cases} x[n-k], & 0 \le n-k \le M-1 \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

where $X[n,k]$ is the (n,k)th element of the matrix $\mathbf{X}$. Returning to the example with signals depicted in (5)-(7), consider the matrix vector multiplication depicted in Fig. 2. The Toeplitz structure in $\mathbf{X}$ is evident and it agrees with (8). In this example, the same comments apply to the elements colored in red. These are transients in which the output is computed without the full impulse response sequence. Notice that the rows of $\mathbf{X}$ explicitly show how the data shift through the memory of the filter over time to produce the output signal.

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \end{bmatrix}}_{\mathbf{y}} = \underbrace{\begin{bmatrix} x_0 & & & \\ x_1 & x_0 & & \\ x_2 & x_1 & x_0 & \\ x_3 & x_2 & x_1 & x_0 \\ x_4 & x_3 & x_2 & x_1 \\ x_5 & x_4 & x_3 & x_2 \\ x_6 & x_5 & x_4 & x_3 \\ x_7 & x_6 & x_5 & x_4 \\ x_8 & x_7 & x_6 & x_5 \\ x_9 & x_8 & x_7 & x_6 \\ & x_9 & x_8 & x_7 \\ & & x_9 & x_8 \\ & & & x_9 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}}_{\mathbf{h}}$$

Figure 2: Illustration of convolution as the product of a Toeplitz structured matrix and a vector: $\mathbf{y} = \mathbf{X}\mathbf{h}$.

2 Single-Threaded Filtering

The preceeding section shows that convolution may be cast as matrix-vector multiplication. This section and the next develop filtering algorithms based on two different interpretations of matrix-vector multiplication. These build on the matrix structures in Figs. 1 and 2. Let's do the easy one first.

In the matrix-vector product

$$\mathbf{y} = \mathbf{X}\mathbf{h},$$

the nth element of $\mathbf{y}$ is computed as the inner product of the nth row of $\mathbf{X}$ and $\mathbf{h}$. To illustrate, consider the calculation of y_6 in Fig. 2. We have

$$y[6] = \begin{bmatrix} x_6 & x_5 & x_4 & x_3 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} = \sum_{k=0}^{3} h[k]x[6-k]. \tag{9}$$

Notice that this agrees with (2) with $n = 6$ and $N = 4$. The inner product interpretation matrix-vector multiplication and convolution leads us to think of constructing an array in computer memory and shifting samples of the input linearly across the array as they become available. After each new sample is input to the array, the inner product is computed between the data array and a coefficient array. Use of a circular buffer instead of a linear shift buffer avoids overhead of copying samples across the array, but leads to slightly more complicated indexing. The code listing below implements convolution. The inner-most for loop computes the inner product using circular (modulo) indexing of the data array. This code is suitable for a single-threaded compute environment.

```c
#include <stdio.h>
#include <string.h>

#define N  4 /* impulse response length */
#define M 10 /* input signal length */
#define L N+M-1 /* output signal length */

int main(int argc, char* argv[]) {
  float h[N] = {0.25, 0.25, 0.25, 0.25};
  float x[N] = {0.0 , 0.0 , 0.0 , 0.0 };
  float in[L] = {-5,-4,-3,-2,-1,0,1,2,3,4,0,0,0};
  float out[L];
  float y;
  int n, k, i=0;

  for(n=0; n<L; n++) {
    x[i] = in[n]; // input into circular array
    y = 0.0;      // initialize accumulator
    for(k=0; k<N; k++) {
      y += h[k] * x[(i+k) % N]; // MAC, circ index
    }
    out[n] = y;
    i = (i-1+N) % N; // update circular index
  }

  return 0;
}
```

Remark. The input array is initialized to be length $L = 13$. Only $M = 10$ input samples are set, and the final three samples are set to zero. These zero input samples are added so that the ending convolution transient would be produced by this code.

3 Multi-Threaded Filtering

To develop a concept for filtering in a multi-threaded compute environment, consider the product $\mathbf{y} = \mathbf{H}\mathbf{x}$ depicted in Fig. 1. Now, instead of thinking about the elements of $\mathbf{y}$ being computed one at a time as a sequence of inner products, let's change our perspective to the concept of computing the entire $\mathbf{y}$ vector all at once, i.e. in parallel, by a linear combination of the columns of $\mathbf{H}$. The coefficients in this linear combination are the elements of $\mathbf{x}$, which are the samples of the input signal. To aid this line of thinking, the equation below shows the linear combination of the columns of $\mathbf{H}$ explicitly.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} + x_1 \begin{bmatrix} \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} + x_2 \begin{bmatrix} \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} + x_3 \begin{bmatrix} \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \\ \\ \end{bmatrix} + x_4 \begin{bmatrix} \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \\ \end{bmatrix} + x_5 \begin{bmatrix} \\ \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \\ \end{bmatrix} + x_6 \begin{bmatrix} \\ \\ \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \\ \end{bmatrix} + x_7 \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \\ \end{bmatrix} + x_8 \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ \end{bmatrix} + x_9 \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}.$$

When convolution is viewed from this perspective, the contribution of each input sample to a set of output samples can be seen explicitly. Let $\mathbf{y}$ be an array of accumulators in computer memory that are initialized to zero. As each input sample is received, let it scale a column of $\mathbf{H}$ and accumulate the products into the array of accumulators. The multiply-accumulate operations may be performed in parallel. The esence of efficient filtering in a multithreaded environment is to exploit the use of parallel accumulators. In the inner product model of convolution, memory is used to store a history of input samples. In the linear combination model of convolution, memory is used for accumulators and the inputs are used one at a time as they are received. No input sample storage is needed. For example, when x_6 is received at $n = 6$, all the previous samples have been processed and the $\mathbf{y}$ accumulators holds complete sums, y_5 and the outputs preceeding it, and partial sums, y_6 and the outputs following it. At $n = 6$, x_6 scales the appropriate column of $\mathbf{H}$, and the products are accumulated into $\mathbf{y}$. After the accumulation y_6 joins the list of completed convolution sums.

Obviously for very long input signals, it is unreasonable to have a separate accumulator for each output sample. Because most of the elements in any column are zero, only a few multiplies and accumulations are needed. Below we work out the details of indexing to make the calculations as efficient as possible.

An assessment of the convolution sums in (1) and (2) reveals a relationship among the input and output samples. An understanding of this relationship helps in the development of filtering algorithms in a parallel processing setting. First note that $y[n]$ depends on input samples $x[n], x[n-1], \cdots, x[n-N+1]$, i.e. the output at time n depends on the input at time n and $N-1$ past inputs. Conversely, the input $x[n]$ at time n contributes to the output $y[n]$ at time n and to $N-1$ future outputs.

Segment the samples of the output signal $y[n]$ into blocks of length N, where for convenience N is the length of the impulse response sequence $h[n]$. Mathematically, blocking is performed using the change of variables $n = uN + i$, where $u = \lfloor n/N \rfloor$ and $i = n\%N = n - uN \in \{0, 1, \cdots, N-1\}$. In words we say that the nth sample lies at the ith sample in the uth block. The samples in the uth block are

$$\{y[uN], y[uN+1], y[uN+2], \cdots, y[uN+N-1]\}. \tag{10}$$

The input sample $x[n] = x[uN+i]$ contributes to the following outputs:

$$\underbrace{\{y[uN+i], \cdots, y[uN+N-1],}_{u\text{th block}} \underbrace{y[(u+1)N], \cdots, y[(u+1)N+i-1]\}}_{(u+1)\text{th block}}. \tag{11}$$

Some of those samples lie in the uth block and some may spill over into the $(u+1)$th block.

The reason for blocking the input is to limit to N the number of accumulators needed to compute all the samples of the output $y[n]$. Consider the convolution sum in (2) for two cases:

- $n = uN + l$ where $l = i, i+1, \cdots, N-1$, and
- $n = (u+1)N + l$ where $l = 0, 1, \cdots, i-1$.

These cases correspond to outputs that depend on $x[uN+i]$ as seen in (11).

In the first case ($i \le l \le N-1$), we have

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{12}$$

$$y[uN+l] = \sum_{k=0}^{N-1} h[k]x[uN+l-k] \tag{13}$$

$$= \underbrace{\sum_{k=l-(i-1)}^{N-1} h[k]x[uN+l-k]}_{\text{(past inputs)}} + \underbrace{h[l-i]x[uN+i]}_{\text{(current input)}} + \underbrace{\sum_{k=0}^{i-1} h[k]x[uN+l-k]}_{\text{(future inputs)}}. \tag{14}$$

Here the convolution sum has been separated into terms involving only past input samples, the current input sample, and future input samples. Because the future inputs are not available, that term may be dropped. Furthermore, define the lth accumulator value at time i to be

$$a_l^i = \sum_{k=l-i}^{N-1} h[k]x[uN+l-k]. \tag{15}$$

Notice that this is a partial sum unless $l = i$ in which case, the lower limit is zero and the partial sum becomes the full convolution sum in (2). With these definitions, we may deduce the following accumulator recursion from (14):

$$a_l^i = a_l^{i-1} + h[l-i]x[uN+i]. \tag{16}$$

There are N different accumulators, $a_0^i, a_1^i, \cdots, a_{N-1}^i$ and they may all be updated in parallel. Each update uses the current input sample $x[uN+i]$ and a different filter coefficient $h[l-i]$.

The second case ($0 \le l \le i-1$) is similar,

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{17}$$

$$y[(u+1)N+l] = \sum_{k=0}^{N-1} h[k]x[(u+1)N+l-k] \tag{18}$$

$$= \underbrace{\sum_{k=N+l-(i-1)}^{N-1} h[k]x[(u+1)N+l-k]}_{\text{(past inputs)}} + \underbrace{h[l-i+N]x[uN+i]}_{\text{(current input)}} + \underbrace{\sum_{k=0}^{N+l-i-1} h[k]x[(u+1)N+l-k]}_{\text{(future inputs)}}. \tag{19}$$

The accumulator recursion in this case is

$$a_l^i = a_l^{i-1} + h[l-i+N]x[uN+i]. \tag{20}$$

Compared to (16) the only difference is the argument of $h[l-i+N]$. Because $l < i$ the added N keeps the sum in the valid range $0 \le l-i+N \le N-1$. Both recursions can be combined into a single recursion that is valid for all l,

$$a_l^i = a_l^{i-1} + h[(l-i)\%N]x[uN+i], \quad l = 0, 1, \cdots, N-1. \tag{21}$$

We will shortly give a code parallel code listing that exploits the accumulator recursion, but the code listing below shows how (21) may be used in a single-threaded filtering routine.

```
#include <stdio.h>

#define N  4 /* impulse response length */
#define M 10 /* input signal length */
#define L N+M-1 /* output signal length */

int main(int argc, char* argv[]) {
  float h[N] = {0.25, 0.25, 0.25, 0.25};
  float x[L] = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 0, 0, 0};
  float y[L];
  float a[N] = {0.0 , 0.0 , 0.0 , 0.0 };
  int i=0; // current data index
  int l; // accumulator index
  int n; // output time index
  int m; // working variable

  for(n=0; n<L; n++) {
    for(l=0; l<N; l++) {
      m = (l-i+N) % N; // compute coefficient index
      a[l] += h[m] * x[n]; // multiply-accumulate
      if(m==0) {
        y[n] = a[l]; // output completed accumulation
        a[l] = 0.0;  // reset accumulator
      }
    }
    i = (i+1+N) % N; // update the current sample index
  }

  return 0;
}
```