HAL open science

In-Place Multicore SIMD Fast Fourier Transforms

Benoît Dupont de Dinechin, Julien Hascoet, Orégane Desrentes

To cite this version:

Benoît Dupont de Dinechin, Julien Hascoet, Orégane Desrentes. In-Place Multicore SIMD Fast Fourier Transforms. HPEC 2023 - 27th Annual IEEE High Performance Extreme Computing Virtual Conference, Sep 2023, Virtual conference, United States. hal-04240798

HAL Id: hal-04240798

https://inria.hal.science/hal-04240798

Submitted on 13 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

In-Place Multicore SIMD Fast Fourier Transforms

Benoît Dupont de Dinechin
Kalray SA, Montbonnot, France
bddinechin@kalray.eu

Julien Hascoët
Kalray SA, Montbonnot, France
jhascoet@kalray.eu

Orégane Desrentes
Kalray S.A. & INSA Lyon (CITI)
Montbonnot & Villeurbanne, France
odesrentes@kalray.eu

***Abstract*—We revisit 1D Fast Fourier Transforms (FFT) implementation approaches in the context of compute units composed of multiple cores with SIMD ISA extensions and sharing a multi-banked local memory. A main constraint is to spare use of local memory, which motivates us to use in-place FFT implementations and to generate the twiddle factors with trigonometric recurrences. A key objective is to maximize bandwidth of the multi-banked local memory system by ensuring that cores issue maximum-width aligned non-temporal SIMD accesses. We propose combining the SIMD lane-slicing and sample partitioning techniques to derive multicore FFT implementations that do not require matrix transpositions and only involve one stage of bit-reverse unscrambling. This approach is demonstrated on the Kalray MPPA3 processor compute unit, where it outperforms the classic six-step algorithm for multicore FFT implementation.**

***Index Terms*—FFT, DIF, DIT, four-step, six-step, multicore.**

I. Introduction

Application CPUs expose significant processing capabilities by combining multicore processing, multiple instruction issuing per clock cycle, and SIMD instruction sets. When integrated into many-core processors, such CPUs are clustered into "core complexes" that share a slice of the memory hierarchy. GPGPU processors propose a similar organization, where the "stream cores" of a "compute unit" share a multi-banked local memory [1] and a cache coherency domain.

In this work, we focus on the design and implementation of FFT algorithms that exploit compute units composed of multiple cores with SIMD ISA extensions such as those of Kalray MPPA processors [2], [3]. Optimizing applications for multicore SIMD execution requires high memory bandwidth, which is addressed by multi-banking the local memory system (L2 cache or scratch-pad memory) and by providing L1 cache-bypass memory access instructions for SIMD data.

The proposed multicore SIMD FFT algorithms operate in-place in memory, in order to spare use of the shared local memory. As such, they involve bit-reverse scrambling of the samples; thus, we assume that the core ISA supports efficient bit-reversing of transform-sized integers (e.g. 16-bit integers for 2^{16} samples). These multicore FFT algorithms are also compatible with implementation techniques that expose SIMD execution opportunities, in our case SIMD lane-slicing [3].

The presentation is as follows. Section II provides the relevant Cooley-Tukey FFT background. Section III describes the high performance FFT enabling techniques that we use in implementations. Section IV exposes the core of our contribution to in-place multicore FFT algorithms. Section V presents experimental results. Section VI discusses related work.

II. Cooley-Tukey FFT Background

Given a sequence N complex numbers $[x_n]_{0\le n\le N-1}$, its Discrete Fourier Transform (DFT) is the sequence of N complex numbers $[X_k]_{0\le k\le N-1}$ defined as:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \text{ with } W_N^{kn} = e^{\frac{-2i\pi kn}{N}} \text{ and } i^2 = -1$$

Assuming $N = N_1 N_2$, the Cooley-Tukey FFT (Fast Fourier Transform) algorithms rely on a decomposition of the time $n = n_1 N_2 + n_2$ and frequency $k = k_1 + k_2 N_1$ DFT indices in order to expose independent subtransforms which are combined to reduce the algorithmic complexity [4]. Since $W_{N_1 N_2}^{n_1 N_2 k_1 + n_1 N_2 k_2 N_1 + n_2 k_1 + n_2 k_2 N_1} = W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2}$:

$$X_{k_1+k_2N_1} = \sum_{n_2=0}^{N_2-1} \left[\left(\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} W_{N_1}^{n_1k_1} \right) W_N^{n_2k_1} \right] W_{N_2}^{n_2k_2}$$

Textbook Cooley-Tukey FFT implementations take $N_2 = 2$ or $N_1 = 2$, respectively referred to as radix-2 decimation in time (DIT) or radix-2 decimation in frequency (DIF). High-performance Cooley-Tukey FFT implementations use radices larger than two to save computations. A radix-r FFT algorithm is composed of $\log_r N$ stages but requires that N which is a power of two, also be a power of r.

The key feature of the radix-r DIT FFT algorithms is to combine the results of subtransforms that operate on the subset of samples that have the same residue modulo r. In particular, the radix-4 DIT FFT recurrence equations are:

$$\begin{cases} X_l &= (A_l + W_N^{2l} C_l) + (W_N^l B_l + W_N^{3l} D_l) \\ X_{l+\frac{N}{4}} &= (A_l - W_N^{2l} C_l) - i(W_N^l B_l - W_N^{3l} D_l) \\ X_{l+\frac{2N}{4}} &= (A_l + W_N^{2l} C_l) - (W_N^l B_l + W_N^{3l} D_l) \\ X_{l+\frac{3N}{4}} &= (A_l - W_N^{2l} C_l) + i(W_N^l B_l - W_N^{3l} D_l) \end{cases}$$

With $A_l = \sum_{n=0}^{\frac{N}{4}-1} x_{4n} W_{\frac{N}{4}}^{nl}$, $B_l = \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} W_{\frac{N}{4}}^{nl}$, $C_l = \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} W_{\frac{N}{4}}^{nl}$ and $D_l = \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} W_{\frac{N}{4}}^{nl}$.

Similarly, the key feature of the radix-r DIF algorithms is to combine the results of subtransforms that operate on the subset of samples resulting from partitioning into r sections. In particular, the radix-4 DIF FFT recurrence equations are:

$$\begin{cases} a_n &= ((x_n + x_{n+\frac{2N}{4}}) + (x_{n+\frac{N}{4}} + x_{n+\frac{3N}{4}})) \\ b_n &= ((x_n - x_{n+\frac{2N}{4}}) - i(x_{n+\frac{N}{4}} - x_{n+\frac{3N}{4}}))W_N^n \\ c_n &= ((x_n + x_{n+\frac{2N}{4}}) - (x_{n+\frac{N}{4}} + x_{n+\frac{3N}{4}}))W_N^{2n} \\ d_n &= ((x_n - x_{n+\frac{2N}{4}}) + i(x_{n+\frac{N}{4}} - x_{n+\frac{3N}{4}}))W_N^{3n} \end{cases}$$

With $X_{4l} = \sum_{n=0}^{\frac{N}{4}-1} a_n W_{\frac{N}{4}}^{nl}$, $X_{4l+1} = \sum_{n=0}^{\frac{N}{4}-1} b_n W_{\frac{N}{4}}^{nl}$, $X_{4l+2} = \sum_{n=0}^{\frac{N}{4}-1} c_n W_{\frac{N}{4}}^{nl}$, $X_{4l+3} = \sum_{n=0}^{\frac{N}{4}-1} d_n W_{\frac{N}{4}}^{nl}$.

Another application of the two-index decomposition of the time and frequency indices is the four-step FFT algorithm [5], which interprets the input samples as a $N_2 \times N_1$ complex matrix and performs the following steps [6]:

1) N_2 simultaneous N_1-point FFTs on the rows:
   $y_1(n_2, k_1) = \sum_{n_1=0}^{N_1-1} x_{n_1 N_2 + n_2} W_{N_1}^{n_1 k_1}$
2) Twiddle factor multiplication (twisting):
   $y_2(n_2, k_1) = y_1(n_2, k_1) W_N^{n_2 k_1}$
3) Transposition to a $N_1 \times N_2$ complex matrix:
   $y_3(k_1, n_2) = y_2(n_2, k_1)$
4) N_1 simultaneous N_2-point FFTs on the rows:
   $y_4(k_1, k_2) = \sum_{n_2=0}^{N_2-1} y_3(k_1, n_2) W_{N_2}^{n_2 k_2}$

The four-step algorithms expose N_2 independent FFT computations, a matrix transposition, a point-wise multiplication, and N_1 independent FFT computations, which makes it a starting point for multicore FFT implementations [7]. However, in step 1 the input samples $x_n = x_{n_1 N_2 + n_2}$ are accessed with stride N_2, while after step 4 sequentially accessing the output samples $X_k = X_{k_1 + k_2 N_1}$ involves reading $y_4(k_1, k_2)$ with stride N_1 (assuming row-major array layout).

The six-step FFT algorithm [6] addresses the spatial locality issues of the four-step FFT algorithm by wrapping two matrix transpositions around it:

0) Transpose the input samples x into y_1:
   $y_0(n_2, n_1) = x_{n_1 N_2 + n_2}$
1)..4) Four-step FFT algorithm using $y_0(n_2, k_1)$ in 1)
5) Transpose y_4 into the output samples X:
   $X_{k_1 + k_2 N_1} = y_4(k_1, k_2)$

As the six-step FFT algorithm localizes the accesses of each FFT computation into its corresponding row, it is well suited to hierarchical memory systems where the local memory (scratch-pad or cache) can store N_1 then N_2 samples.

III. FFT Implementation Techniques

A. In-Place Data Processing

When computing FFTs using the Cooley-Tukey DIT or DIF algorithms, it is possible to organize memory accesses so that each stage overwrites the results of the previous stage. After the last FFT stage, the output samples are located in-place of the input samples. This approach is especially interesting in the case of limited local memory capacity, but entails data "scrambling". For a DIT algorithm, scrambling the input samples yields the output samples in natural order, while for the DIF algorithm, the naturally ordered input samples yield the output samples in scrambled order.

For the radix-2 algorithms, scrambling an array of samples means applying the self-inverse permutation corresponding to bit-reversing $\log_2 N$ bits in the binary representation of every sample index [8]. Let us denote **bit-reverse**(i, b) the function that takes an integer i whose representation fits into b bits and returns the corresponding bit-reversed value. Efficient bit-reversing is supported in several instruction set architectures (ISA), including the one we use for experiments (§ V). When ISA support for bit-reversing is not available, several approaches are available to scramble samples [9].

Algorithm 1 Scramble an array $x[]$ of n elements

```
for i = 0 to n − 1 do
    j ← bit-reverse(i, log2 n)
    if j > i then
        t = x[i]
        x[i] = x[j]
        x[j] = t
    end if
end for
```

The bit-reverse scrambling of an array may be implemented in-place (Algorithm 1). The loop of this algorithm has independent iterations, as the memory locations written by iteration i cannot overlap with those read by iteration $i' \neq i$:

- Assume that iteration i writes to $x[i]$ and $x[j]$. This implies $i < j =$ **bit-reverse**$(i, l2n)$.
- Assume that iteration i' reads from $x[i']$ and $x[j']$. This implies $i' < j' =$ **bit-reverse**$(i', l2n)$.
- If the writes from iteration i overlap with the reads of iteration i', then $i = i' \vee i = j' \vee j = i' \vee j = j'$.
- As **bit-reverse** is injective (bijective), $i \neq i' \Rightarrow j \neq j'$.
- Then $i = j' \vee j = i'$, which contradicts $i < j \wedge i' < j'$.

Therefore, the loop of Algorithm 1 can be parallelized between cores and vectorized or software pipelined on each core.

When implementing radix-r DIT or DIF algorithms with $r > 2$ a power of two, natural scrambling corresponds to the base-r permutation of the binary representation of the sample index, also known as "digit-reversal" [10]. For example, if $r = 8$, then the permutation reverses the order of triples of bits in the binary representation of the sample index. Digit-reversal does not have the same support as bit-reversal in existing ISAs, which could downplay the implementation of high-radix Cooley-Tukey FFT algorithms.

However, in [11] Burrus explains how to modify a radix-r FFT algorithm, $r = 2^m$, so that it only involves bit-reverse scrambling. For a DIF algorithm, the outputs of each radix-r butterfly should be bit-reversed. Likewise in case of a DIT algorithm, the inputs of each radix-r butterfly should be bit-reversed. In practice, for a radix-4 FFT one only needs to exchange the inputs (DIT) or the outputs (DIF) 1 and 2 of the butterfly to keep bit-reverse scrambling [10].

B. Enabling Loop Software Pipelining

The defining property of a parallel loop is that there are no loop-carried dependencies between iterations. The defining property of a vector loop is that it is innermost with lexically forward (or from RHS to LHS at the statement level) loop-carried dependencies if any [12]. Thus, a parallel loop is a vector loop, and conversely a vector loop may be converted into a sequence of parallel loops at the expense of introducing temporary vectors to carry values between these loops.

Software pipelining is the key technique to take advantage of vector loops on statically scheduled application cores, that

is, in-order superscalar or VLIW. This technique restructures the loop body so that long-latency dependencies become loop-carried with an iteration distance large enough to avoid latency stalls (memory and compute) inside the new loop body.

In order to benefit from software pipelining in iterative Cooley-Tukey FFT implementations, Texas Instruments collapsed the innermost loops in their TI C6x DSP_fft32x32 code for the C6x VLIW-DSP cores. As illustrated in Algorithm 2 for a radix-2 DIF, an iterative Cooley-Tukey FFT is implemented with three nested loops [4]: the innermost l loop iterates over butterflies; the middle loop j iterates over groups of butterflies that share the same twiddle factors; and the outer k loop iterates over the stages of the FFT. This is apparent on Burrus-style [4] Algorithm 2 and Algorithm 3 (radix-4 DIF) that operate in-place, and produce bit-reversed output.

Algorithm 2 Burrus-style implementation of radix-2 DIF FFT

```
for k = 0; k < log_2 n; k += 1 do
  n_1 ← n ≫ k
  n_2 ← n_1 ≫ 1
  for j = 0; j < n_2; j += 1 do
    w_j ← {cos(-2jπ/n_1), sin(-2jπ/n_1)}
    for l = j; l < n; l += n_1 do
      l_1 ← l + n_2
      t_1 ← x[l] − x[l_1]
      x[l] ← x[l] + x[l_1]
      x[l_1] ← t_1 w_j
    end for
  end for
end for
```

The challenge of collapsing the inner loops is to reconstruct the l, j indices from the counter r of the collapsed loop. In case of radix-2, the collapsed loop iterates $\frac{n}{2}$ times and:

$$r \in [0, \frac{n}{2} - 1] : \begin{cases} j \leftarrow r \gg k \\ l \leftarrow j + (r \ll (\log_2 n - k))\%n \end{cases}$$

In case of radix-4, the collapsed loop iterates $\frac{n}{4}$ times and:

$$r \in [0, \frac{n}{4} - 1] : \begin{cases} j \leftarrow r \gg 2k \\ l \leftarrow j + (r \ll (\log_2 n - 2k))\%n \end{cases}$$

The disadvantage of collapsing the inner loops is that accessing the twiddle factors from a table or recomputing them is now performed at each iteration of the new inner loop. This can be mitigated by conditionally accessing or computing the twiddle factors whenever j changes, under one of the conditions $j \neq (r \pm 1) \gg k$ for the Burrus radix-2 DIF and $j \neq (r \pm 1) \gg 2k$ for the Burrus radix-4 DIF.

C. Twiddle Factor Recurrences

The twiddle factors of a N-point DFT are defined as $W_P^{kl} = e^{\frac{-2i\pi kl}{P}}$, with $i^2 = -1$ and P a factor of N. Although one may tabulate the twiddle factors in a way that ensures the spatial locality of accesses, this requires storage of size comparable to the FFT samples. The alternative to spare local memory capacity or cache footprint is to compute twiddle factors using one of the proposed trigonometric recurrences [13], each with different cost and worst-case accuracy bounds [14].

Algorithm 3 Burrus-style implementation of radix-4 DIF FFT

```
for k = 0; k < log_4 n; k += 1 do
  n_1 ← n ≫ 2k
  n_2 ← n_1 ≫ 2
  for j = 0; j < n_2; j += 1 do
    w_j ← {cos(-2jπ/n_1), sin(-2jπ/n_1)}
    w_2j ← {cos(-4jπ/n_1), sin(-4jπ/n_1)}
    w_3j ← {cos(-6jπ/n_1), sin(-6jπ/n_1)}
    for l = j; l < n; l += n_1 do
      l_1 ← l + n_2
      l_2 ← l_1 + n_2
      l_3 ← l_2 + n_2
      t_1 ← x[l] + x[l_2]
      t_2 ← x[l_1] + x[l_3]
      t_3 ← x[l] − x[l_2]
      t_4 ← x[l_1] − x[l_3]
      x[l] ← t_1 + t_2
      t_2 ← t_1 − t_2
      t_1 ← t_3 + it_4
      t_3 = t_3 − it_4
      x[l_2] ← t_3 w_j        /* l_2 instead of l_1 [11] */
      x[l_1] ← t_2 w_2j       /* l_1 instead of l_2 [11] */
      x[l_3] ← t_1 w_3j
    end for
  end for
end for
if log_2 n mod 2 ≠ 0 then
  Do a radix-2 stage with twiddle factor 1.
end if
```

With the availability of fused multiply-add floating-point operators in application cores, the trigonometric recurrences to consider compute $e^{i(k+1)\theta}$ directly from $e^{ik\theta}$ and $e^{i\theta}$:

$$e^{i(k+1)\theta} = e^{ik\theta}e^{i\theta} \qquad \text{(multiply recurrence)}$$

$$e^{i(k+1)\theta} = e^{ik\theta} + e^{ik\theta}(e^{i\theta} - 1) \qquad \text{(multiply-add recurrence)}$$

For the multiply-add recurrence, the real part of $e^{i\theta} - 1$ is calculated as $-2\sin^2\frac{\theta}{2}$ to avoid cancellation in $cos\theta - 1$ [15] since $\theta = -\frac{2\pi}{P}$ may be small for large-N FFTs.

In case of radix-2 FFT algorithms, there is one recurrence computation per iteration of the twiddle group loop, with absolute angle values spanning $[0, \pi)$. In case of the radix-4 FFT algorithms, there are three recurrence computations per twiddle group loop iteration, with angle absolute values spanning respectively $[0, \frac{\pi}{2})$, $[0, \pi)$ and $[0, \frac{3\pi}{2})$.

Algorithm 4 illustrates a Burrus-style radix-4 DIF FFT with collapsed inner loops and multiply-add recurrences for twiddle factors. Precisely, **steps**$[m]$ returns a 3-element array with the complex values $-2\sin^2\frac{\theta_k}{2} + i\sin\theta_k$ for $\theta_k = \frac{-2\pi k}{2^m}, k \in \{1, 2, 3\}$. The **cfma** operation computes a complex fused multiply add where the first two arguments are multiplicands and the third is the addend. In an actual implementation, conditional updates of w_j, w_{2j}, w_{3j} are optimized into unconditional **cfma** operations that write to temporary variables, which are then conditionally moved.

Algorithm 4 Burrus-style implementation of radix-4 DIF FFT with collapsed inner loop and twiddle recurrences

```
for k = 0; k < log_4 n; k += 1 do
  n_1 ← n ≫ 2k
  n_2 ← n_1 ≫ 2
  s[0..2] ← steps[log_2 n − 2k]
  w_j ← w_2j ← w_3j ← {1.0, 0.0}
  for r = 0; r < n/4; r += 1 do
    j ← r ≫ 2k
    l ← j + (r ≪ (log_2 n − 2k))%n
    l_1 ← l + n_2
    l_2 ← l_1 + n_2
    l_3 ← l_2 + n_2
    t_1 ← x[l] + x[l_2]
    t_2 ← x[l_1] + x[l_3]
    t_3 ← x[l] − x[l_2]
    t_4 ← x[l_1] − x[l_3]
    x[l] ← t_1 + t_2
    t_2 ← t_1 − t_2
    t_1 ← t_3 + it_4
    t_3 = t_3 − it_4
    x[l_2] ← t_3 w_j        /* l_2 instead of l_1 [11] */
    x[l_1] ← t_2 w_2j       /* l_1 instead of l_2 [11] */
    x[l_3] ← t_1 w_3j
    if j ≠ (r + 1) ≫ 2k then
      w_j ← cfma(w_j, s[0], w_j)
      w_2j ← cfma(w_2j, s[1], w_2j)
      w_3j ← cfma(w_3j, s[2], w_3j)
    end if
  end for
end for
if log_2 n mod 2 ≠ 0 then
  Do a radix-2 stage with twiddle factor 1.
end if
```

IV. Multicore FFT Implementations

A. Multicore SIMD Lane-Slicing

The first approach we propose to implement in-place FFTs on multiple cores with SIMD ISA extensions is to extend the single-core SIMD lane-slicing of [3] to parallel execution. SIMD lane-slicing can be implemented in-place in two steps with normally ordered input and scrambled output:

- Step 1 computes s independent DFTs of size $\frac{n}{s}$ using a DIF FFT algorithm whose scalar data types are replaced by the corresponding SIMD data types. The output of this step is a bit-reversed array of s-sized vectors.
- Step 2 combines the s subtransforms (one per SIMD lane) with one stage of $\frac{n}{s}$ radix-s DIT butterflies. The output of this step is the bit-reversed array of output samples.

SIMD lane-slicing exploits the fact that the distribution of input samples across the SIMD lanes decimates them in time with a subtransform in each SIMD lane. Furthermore, both steps operate into an array $v[]$ of $\frac{n}{s}$ vectors of s samples each, which aliases the array $x[]$ of n samples. As a result, they only perform aligned SIMD memory accesses to vectors of s samples, which fully exploit local memory bandwidth. This is obvious for Step 1 and apparent for Step 2 (Algorithm 5).

In this work, we use the complex multiply-add recurrence to compute the twiddle factors in the implementation of Step 1 and Step 2. As the output of Step 1 is scrambled on s-sized vectors, the inner loop of Step 2 (Algorithm 5) is iterated in bit-reverse order to ensure that the twiddle factors follow a geometric progression as required by the recurrences.

Please note that Algorithm 5 outputs the elements of vector $v[l]$ in bit-reverse order to ensure that the output array $x[]$ is scrambled at the granularity of complex samples. Indeed, Step 2 is based on the last stage of a radix-s DIT algorithm, which normally scatters the output samples with stride $\frac{n}{s}$. As the Step 2 output has to be unscrambled, bit-reversing the produced vector elements prepares this scattering.

Algorithm 5 Implementation of SIMD lane-slicing Step 2 with $s = 4$ (based on a radix-4 DIT last stage)

```
s[0..2] ← steps[log_2 n]
w_k ← w_2k ← w_3k ← {1.0, 0.0}
for k = 0; k < n/4; k += 1 do
  l ← bit-reverse(k, log_2 n/4)
  z_l[4] ← v[l]
  a_l ← z_l[0]
  b_l ← w_k z_l[1]
  c_l ← w_2k z_l[2]
  d_l ← w_3k z_l[3]
  z_l[0] ← (a_l + c_l) + (b_l + d_l)
  z_l[1] ← (a_l − c_l) − i(b_l − d_l)
  z_l[2] ← (a_l + c_l) − (b_l + d_l)
  z_l[3] ← (a_l − c_l) + i(b_l − d_l)
  v[l] ← {z_l[0], z_l[2], z_l[1], z_l[3]}   /* bit-reversed */
  w_k ← cfma(w_k, s[0], w_k)
  w_2k ← cfma(w_2k, s[1], w_2k)
  w_3k ← cfma(w_3k, s[2], w_3k)
end for
```

In order to derive a multicore FFT implementation from the SIMD lane-slicing approach, we execute in parallel the inner loops of Step 1 (the collapsed loop), Step 2 (Algorithm 5) and of the bit-reverse (un)scrambling Algorithm 1. The inner loops of Step 1 and Step 2 involve trigonometric recurrences so to execute these loops in parallel by chunks, each chunk starts its recurrences from precomputed twiddle factors.

B. Multicore Sample Sectioning

The second approach we propose to implement in-place FFTs on multiple cores is to partition input samples into c contiguous sections to expose c independent subtransforms. To be precise, the sample sectioning decimates the input in frequency; therefore, an initial radix-c DIF stage is required before completing the computations with the subtransforms.

An in-place implementation for the first step of this sample sectioning approach is proposed in Algorithm 6 for $c = 4$. It is derived from the inner loop of Algorithm 3 that produces the output scrambled in bit-reverse, here at the section granularity. We also compute the twiddle factors with three complex multiply-add recurrences like in Algorithm 4.

Algorithm 6 Burrus-style implementation of sample sectioning Step 1 with $c = 4$ (based on a radix-4 DIF first stage)

$x_0 \leftarrow x$
$x_1 \leftarrow x + \frac{n}{4}$
$x_2 \leftarrow x + \frac{n}{2}$
$x_3 \leftarrow x + \frac{3n}{4}$
$s[0..2] \leftarrow$ **steps**$[\log_2 n]$
$w_k \leftarrow w_{2k} \leftarrow w_{3k} \leftarrow \{1.0, 0.0\}$
for $k = 0; k < \frac{n}{4}; k += 1$ **do**
  $t_1 \leftarrow x_0[k] + x_2[k]$
  $t_2 \leftarrow x_1[k] + x_3[k]$
  $t_3 \leftarrow x_0[k] - x_2[k]$
  $t_4 \leftarrow x_1[k] - x_3[k]$
  $x_0[k] \leftarrow t_1 + t_2$
  $t_2 \leftarrow t_1 - t_2$
  $t_1 \leftarrow t_3 + it_4$
  $t_3 \leftarrow t_3 - it_4$
  $x_2[k] \leftarrow t_3 w_k$ /* x_2 instead of x_1 [11] */
  $x_1[k] \leftarrow t_2 w_{2k}$ /* x_1 instead of x_2 [11] */
  $x_3[k] \leftarrow t_1 w_{3k}$
  $w_k \leftarrow$ **cfma**$(w_k, s[0], w_k)$
  $w_{2k} \leftarrow$ **cfma**$(w_{2k}, s[1], w_{2k})$
  $w_{3k} \leftarrow$ **cfma**$(w_{3k}, s[2], w_{3k})$
end for

The second step of the sample sectioning approach then amounts to calling the subtransforms, which independently process their own section of the samples. Given the bit-reverse scrambling of the Step 1 output, if each subtransform produces its output samples scrambled in bit-reverse, then a single (un)scrambling step recovers the entire output normal in normal order (Algorithm 7).

Algorithm 7 Implementation of sample sectioning with $c = 4$

sectioning-step1(n, x) /* Algorithm 6 */
dif-fft$(\frac{n}{4}, x + 0)$ /* normal input, bit-reversed output */
dif-fft$(\frac{n}{4}, x + \frac{n}{4})$ /* normal input, bit-reversed output */
dif-fft$(\frac{n}{4}, x + \frac{n}{2})$ /* normal input, bit-reversed output */
dif-fft$(\frac{n}{4}, x + \frac{3n}{4})$ /* normal input, bit-reversed output */
bit-reverse-scramble(n, x) /* Algorithm 1*/

The sample sectioning approach can be combined with techniques that exploit SIMD ISA extensions in FFT implementations, in particular the SIMD lane-slicing of [3]. Algorithm 7 is adapted for SIMD lane-slicing so that each call to the **dif-fft** subtransforms is replaced by the sequence of two calls to the SIMD lane-slicing Step 1 and Step 2. SIMD lane-slicing may also be applied to Algorithm 6.

In order to derive a multicore FFT implementation with the sample sectioning approach, we execute in parallel by chunks with recurrence seeds the inner loop of Step 1 (Algorithm 6) and of the bit-reverse (un)scrambling Algorithm 1). We also execute concurrently the calls to the c **dif-fft** subtransforms.

V. Experimental Results

A. Comparing multicore Implementations

To compare the multicore FFT approaches, we implement them in C with builtins for the SIMD complex arithmetic. We compile and execute on four cores of the MPPA3 processor running at 1 GHz with a local memory system configured for 3 MB of shared local memory and 1 MB of level-2 cache [2]. Parallel execution is explicit based on POSIX threads, but we synchronize the cores with a fast hardware barrier.

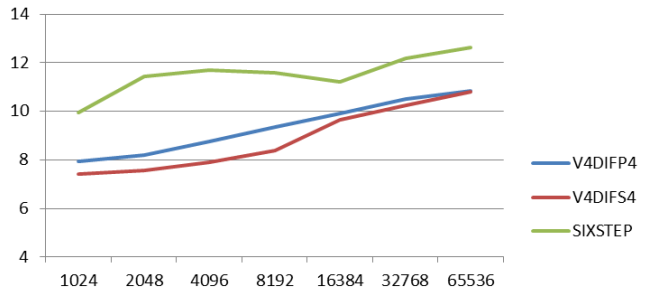

Fig. 1. Cycles divided by transform size of the three multicore FFTs.

Fig. 1 shows the performances in cycles divided by the transform size for three multicore implementations: SIMD lane-slicing (V4DIFP4), sample sectioning (V4DIFS4), and six-step (SIXSTEP). It can be observed that V4DIFP4 and V4DIFS4 both outperform SIXSTEP. Furthermore, V4DIFS4 performs slightly better than V4DIFP4, although we did not (but could) use SIMD lane-slicing for implementing V4DIFS4 Step 1.

B. Accuracy of Trigonometric Recurrences

We evaluate the accuracy of trigonometric recurrences to compute the twiddle factors in the context of fused floating-point arithmetic. Let us consider implementations of the complex multiply $r = xy$ and complex multiply-add $s = xy + z$ operations using real floating-point arithmetic:

$$r = r_{\text{re}} + ir_{\text{im}} \text{ with } \begin{cases} r_{\text{re}} = (x_{\text{re}} y_{\text{re}} - x_{\text{im}} y_{\text{im}}) \\ r_{\text{im}} = (x_{\text{re}} y_{\text{im}} + x_{\text{im}} y_{\text{re}}) \end{cases}$$

$$s = s_{\text{re}} + is_{\text{im}} \text{ with } \begin{cases} s_{\text{re}} = (x_{\text{re}} y_{\text{re}} - x_{\text{im}} y_{\text{im}} + z_{\text{re}}) \\ s_{\text{im}} = (x_{\text{re}} y_{\text{im}} + x_{\text{im}} y_{\text{re}} + z_{\text{im}}) \end{cases}$$

First, consider fused multiply-add (FMA) operators, which compute $a \times b \pm c$ with one rounding step. Also, consider the more advanced FDP2A operators, which compute $a \times b \pm c \times d \pm e$ followed by a single rounding step [16]. The FMA and FDP2A operators, respectively, allow us to implement the complex multiply operation $r = xy$ and the complex multiply-add operation $s = zy + z$ with two (FMA) operations or one (FDP2A) operation per real or imaginary component.

We focus on the comparison between the complex multiply recurrence implemented using FP32 FMA operators and the complex multiply-add recurrence using FP32 FDP2A operators, as both are available in scalar and SIMD variants on our target application cores. When a FP32 FDP2A operator is not available, we observed that a similar accuracy was achieved by using FP64 FMA operations to implement the FP32 complex multiply-add recurrence.

The baseline for comparison computes each twiddle factor using the `libm` standard `cos` and `sin` function, rounded to FP32 from FP64. For each recurrence term, we compute the error as the modulus of the difference between its value and

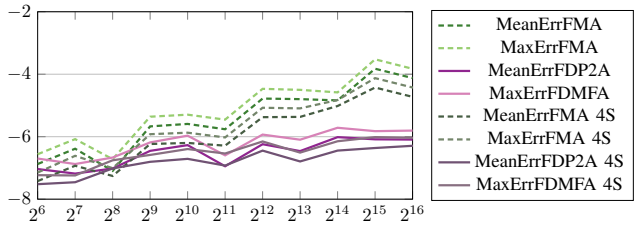

Fig. 2. Base-10 logarithm of errors for the multiply and the multiply-add radix-2 twiddle factor FP32 recurrences depending on the FFT size.

the baseline using FP64 arithmetic. Since twiddle factors are roots of unity, these errors are absolute and relative.

Fig. 2 shows the maximum and average errors for the multiply and multiply-add twiddle factor recurrences in the $\log_{10}$ scale, for N successive powers of two and recurrence angles spanning $[0, \pi)$. When parallelizing loops with twiddle recurrences, each chunk of iterations executing on a core starts from a tabulated twiddle factor. This helps to reduce errors, also shown in Fig. 2 with the 4S curves for four cores.

We observe that errors of multiply recurrences implemented with FMA operators grow up to two orders of magnitude larger than errors of multiply-add recurrences implemented with FDP2A operators. In a radix-2 Cooley-Tukey FFT of size N, there is a twiddle factor recurrence per stage so that all terms of all recurrences up to length $\frac{N}{2}$ are actually used.

VI. Related Work

As we optimize for multi-banked memory systems that are accessed without traversing the L1 caches, the most closely related work is optimizing FFT algorithms for implementations on parallel-vector processors [6], [17], [18]. None of the algorithms proposed collapse the inner loops, but this is possibly useful on classic vector units, provided that the twiddle factors are conditionally accessed from a table.

In the area of multicore FFT algorithms, the main contributions are variants of the four-step and six-step methods [7], [19]. Their focus is to fit the subtransform working sets into the L1 caches and to address the complexities of transposing nonsquare matrices. These topics are irrelevant in our case.

The SIMD lane-slicing implementation of [3] accesses the twiddle factors from tables that are kept in L1 cache as they expose spatial locality, while samples are accessed in L1 cache-bypass mode. This leads to a different condition for updating the twiddle factors in the collapsed loop.

VII. Conclusions

We present two new approaches to the implementation of multicore FFTs, which are compared to the classic six-step FFT algorithm implemented for multicore processors.

The first approach exposes suitably collapsed twiddle loops as the main source of parallel execution. The second approach exposes concurrency across subtransforms following the partitioning of input samples into a number of sections that match the number of cores.

We demonstrate how both approaches leverage FFT implementation techniques that fully exploit the SIMD ISA extensions of the target application cores. Moreover, these approaches operate in-place, producing bit-reversed output samples that may be reordered in-place using a single loop over the samples, which we also show as being parallel.

An intriguing outcome is that our approaches to multicore FFT implementation do not involve any matrix transposition or sample twisting between steps, unlike the four-step method, the six-step method, and other FFT algorithms designed for classic parallel-vector processors.

Acknowledgment

This work has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2).

References

[1] J. R. Diamond, D. S. Fussell, and S. W. Keckler, "Arbitrary modulus indexing," in *2014 47th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2014, pp. 140–152.

[2] B. D. de Dinechin, J. Hascoët, J. L. Maire, and N. Brunie, "Deep Learning Inference on the MPPA3 Manycore Processor," in *Proc. of the Embedded World Conf. 2020, EWC 2020, Nuremberg, Germany*, February 2020.

[3] B. D. de Dinechin, "Computing In-Place FFTs with SIMD Lane Slicing," in *26th Annual IEEE High Performance Extreme Computing Virtual Conference (HPEC)*, 9 2022.

[4] C. S. Burrus, M. Frigo, S. G. Johnson, M. Pueschel, and I. Selesnick, *Fast Fourier Transforms*. Houston, Texas, USA: Connexions, Rice University, 2008.

[5] W. M. Gentleman and G. Sande, "Fast fourier transform for fun and profit," in *Proc. AFIPS, Joint Computer Conference, vol.29*, 1966, pp. 563–578.

[6] D. H. Bailey, "Ffts in external or hierarchical memory," in *Supercomputing '89: Proc. of the 1989 ACM/IEEE Conf. on Supercomputing*, 1989, pp. 234–242.

[7] D. Takahashi, *Fast Fourier Transform Algorithms for Parallel Computers*, ser. High-Performance Computing Series. Springer, 2019, vol. 2.

[8] R. Polge, B. Bhagavan, and J. Carswell, "Fast computational algorithms for bit reversal," *IEEE Transactions on Computers*, vol. C-23, no. 1, pp. 1–9, 1974.

[9] A. H. Karp, "Bit reversal on uniprocessors," *SIAM Review*, vol. 38, no. 1, pp. 1–26, 1996.

[10] P. Papamichalis and C. Burrus, "Conversion of digit-reversed to bit-reversed order in fft algorithms," in *Int. Conf. on Acoustics, Speech, and Signal Processing,*, 1989, pp. 984–987 vol.2.

[11] C. Burrus, "Unscrambling for fast dft algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 7, pp. 1086–1087, 1988.

[12] M. Wolfe and U. Banerjee, "Data dependence and its application to parallel processing," *Int. Journal of Parallel Programming*, vol. 16, pp. 137–178, 1987.

[13] K. Dobeš, "Algorithm fast fourier transforms with recursively generated trigonometric functions," *Computing*, vol. 29, pp. 263–276, 1982.

[14] M. Tasche and H. Zeuner, "Improved roundoff error analysis for precomputed twiddle factors," *J. of Computational Analysis and Applications*, vol. 4, pp. 1–18, 01 2002.

[15] R. C. Singleton, "On computing the fast fourier transform," *J. of the ACM*, vol. 10, no. 10, 1967.

[16] O. Desrentes, B. Dupont de Dinechin, and F. de Dinechin, "Exact fused dot product add operators," in *30th IEEE Int. Symp. on Computer Arithmetic (ARITH), Portland, USA*, 9 2023.

[17] D. H. Bailey, "A high-performance fft algorithm for vector supercomputers," *The Int. J. of Supercomputing Applications*, vol. 2, no. 1, pp. 82–87, 1988.

[18] P. N. Swarztrauber, "Fft algorithms for vector computers," *Parallel Comput.*, vol. 1, no. 1, p. 45–63, aug 1984.

[19] R. Crandall and J. Klivington, "Supercomputer-style fft library for apple g4," Tech. Rep., 01 2000.