Memory Efficiency Oriented Fine-Grain Representation and Optimization of FFT

Salvatore Servodio
University of Delaware
Newark, DE, United States
servogod@udel.edu

Xiaoming Li
University of Delaware
Newark, DE, United States
xli@udel.edu

ABSTRACT

Fast Fourier Transform (FFT) is the computation kernel of many scientific and machine learning applications. Its state-of-the-art implementation in current practices normally involves searching for the best performing variant in a pre-defined implementation space. The expressiveness of the space and the capability to manipulate the variant expressions largely determine the final performance. It is known that large size FFT problems are memory bound. However, existing FFT libraries such as FFTW adopt the search space representations that are oriented for tuning operational efficiency. Such representations make it almost impossible to specifically tune FFT for memory performance. In this paper, we introduce REFFT, a software library capable of computing Discrete Fourier Transforms (DFT) by applying a novel approach to exploring the decomposition space of memory accesses in FFT. This is achieved by creating a new model for representing a Fast Fourier Transform (FFT) by means of a binary tree structure to represent its decomposition and by organizing the subspace of valid solutions into two ordered and directed sets: tree rotations and codelet permutations. Furthermore, in this new model, the nodes of a given FFT binary tree provide computationally meaningful attributes which are derived from its edges. This paper demonstrates that it is possible to easily navigate with fine granularity what would initially seem to be an exceptionally vast space of solutions in a practical manner and find best performing plans by tuning specifically for memory efficiency through mathematical structure manipulations. We compare REFFT with FFTW, one of the most broadly used FFT libraries, for problem sizes ranging from 2^{15} to 2^{25} on three recent Intel and AMD processors. REFFT achieves the average speedup of 14.3% across all sizes with max speedup 43.1%

1 INTRODUCTION

Fast Fourier Transform (FFT) is one of the most broadly used computation kernels in science and engineering. FFT calculates the spectrum representation of time-domain input signals $\vec{x}$ with the formula:

$$F[k] = \sum_{j=0}^{N-1} x[j] \left(e^{-2\pi ik/N}\right)^j, 0 \leq k < N$$

If the input size is N, the FFT operates in $O(NlogN)$ steps. The performance of FFT algorithms is known to be determined only by input size, and not affected by the value of input.

By using different decomposition schemes such as Cooley-Tukey [3] or Winograd [16], FFT can be implemented in many different ways. The performance of an implementation is theoretically determined by both the operational efficiency and the memory access efficiency. However, FFT's memory accesses are intrinsically strided and it conducts data shuffling on input, intermediate results and twiddle factors, all being the type of memory access patterns that current computer memory systems are hard to optimize for. Therefore, it is widely realized that FFT, especially for large size FFT problems, is fundamentally a memory bound problem.

Because the number of possible ways to implement FFT is huge even for relatively small problem sizes, the state-of-the-art FFT libraries such as FFTW [7] almost all use empirically search to tune FFT for a particular computer architecture and for a particular problem size. Generally speaking, the libraries will try different combination of decomposition methods, empirically measure their performance and use the best performing decomposition scheme as the final implementation. Essentially, they define a internal representation of the search space. Therefore, the expressiveness of the space and the capability to manipulate the variant expressions largely determine the final performance.

This work is driven by a crucial observation: existing search space representations in libraries like FFTW often hinder tuning for FFT's memory performance. This issue stems from two main factors. Firstly, current representations prioritize operational efficiency tuning. FFTW, for instance, defines specific FFT implementations as plans that apply decomposing methods top-down, with each method clearly defining operations but overlooking memory accesses. Consequently, tuning for memory performance becomes nearly impossible. Secondly, these high-level decomposition decisions can significantly impact memory access patterns. A small change in the operational decomposition can lead to vastly different memory access behaviors. Thus, in the current setup, fine-tuning memory performance is unattainable.

Our key idea and the key contribution of this work is a new representation of FFT's decomposition space of memory accesses. The representation not only straightforwardly denotes the memory access patterns, but also support fine-grain tuning of memory accesses in FFT. This new representation, when combined with empirical searching, enables direct tuning of memory efficiency in FFT implementations, a capability lacking in existing libraries like FFTW. Specifically, we introduce the Rotational Enumerated FFT Library (REFFT), capable of computing Discrete Fourier Transforms (DFT) by exploring the decomposition space of memory accesses in FFT. This is achieved through a new model representing FFT via a binary tree structure, organizing the subspace of valid solutions into two ordered and directed sets: tree rotations and codelet permutations. In this model, the nodes of an FFT binary tree offer computationally meaningful attributes derived from its edges. Our paper demonstrates the feasibility of navigating what initially appears to be an exceptionally vast solution space with fine granularity and practically tuning memory efficiency by manipulating

mathematical structures. Regarding empirical searching, akin to FFTW, REFFT employs a planner to explore the space of all possible solutions and seeks the best candidate based on a combination of fixed and experimental criteria for a given machine. Crucially, the REFFT planner navigates the abstracted subspace by manipulating the mathematical structures of plans, guided by performance on today's computer memory systems.

2 BACKGROUND

FFT algorithms recursively decompose a N-point DFT into several smaller DFTs [5], and the divide-and-conquer approach reduces the operational complexity of a Discrete Fourier Transform (DFT) from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems. Our REFFT library is based on the general Cooley-Tukey factorization FFT algorithm [4]. In this section we briefly introduce the FFT algorithms and overview how they are incorporated into our approach. The DFT transform of an input series $x(n), n = 0, 1, ..., N-1$ of size N is presented as $Y(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$. We can map the one dimensional input into two dimensions indexed by l in L dimension and m in M dimension, respectively. The Cooley-Tukey FFT decomposes the original DFT into three sub-steps: (1) Perform M DFTs of size L, $A(p,m) = \sum_{l=0}^{L-1} x(l,m)W_L^{lp}$; (2) Multiply twiddle factors, $B(p,m) = A(p,m)W_N^{pm}$; and (3) Perform L DFTs of size M, $Y(p,q) = \sum_{m=0}^{M-1} B(p,m)W_M^{mq}$. Therefore, $Y(k) = Y(pM+q)$.

In essence, Cooley-Tukey introduces a decomposition approach that divides one dimensional computation into two. Moreover, the Radix algorithm [5] is a special case of the Cooley-Tukey algorithm for power-of-two FFT problems. Furthermore, the Rader algorithm [13] and the Bluestein algorithm [1] can be used to calculate prime-sized Fourier transform problems.

In this paper, REFFT's internal representation extends the tensor representation introduced in FFTW [7] to represent the transformations of FFT. An I/O tensor $d(C, Si, So, I, O)$ denotes FFTs along a data dimension where C is the size of one dimensional FFT, Si and So represent the stride of input and output, and I and O are the addresses of input and output array. t_M^L represents multiplication of twiddle factors with size $L \times M$. The I/O tensor representation captures the two most important factors that determine FFT's performance, i.e., data access patterns and computation load. As an example, the Cooley-Tukey FFT decomposition can be precisely denoted as an extended I/O tensor representation $u = \{d(L, M, M, I, O), t_M^L d(M, 1, 1, O, O)\}$.

FFTW is one of the most broadly used FFT libraries and it provides a solution for general FFT problems. Beside it, there are other FFT libraries for different problem domains such as Nukada's work on 3D FFT [12, 11], and Govindaraju's [8] and Gu's work on 2D and 3D FFT [9]. Recently, Gu et.al. [10] demonstrated a FFT library that can solve FFT problems larger than the device memory. For even larger FFT problems, Chen et.al. presented a GPU cluster based FFT implementation [2]. However, since computation contributes only a trivial part to the overall execution time, the work has been almost exclusively focused on the optimization of communication over inter-node channels.

3 FOUNDATIONAL CONCEPTS

The theoretical foundation of REFFT is the representation of FFT as a binary tree with n internal nodes and n+1 leaf nodes. The motivation for this representaion is three-fold: (1) the representation must either explicitly encode or support straightforward deduction of primitive operations in FFT that are most consequential to FFT's memory efficiency. These primitive operations include decimation either in the time domain or in the frequency domain, the shuffling of data (on input or intermediate results), and twiddle factor multiplication. (2) The representation must be able to be indexed so that we can define the performance neighbors of a plan. This addressibility is necessary so that we can fine-tune a plan for its memory performance. Surprisingly existing libraries such as FFTW don't provide this capability, so that tuning in those has to be done as repetitive trials of different high-level decomposition schemes. In some cases, software libraries implement a custom stride feature that would allow users to fine tune memory efficiency to some degree[14]. In such instances, the memory performance can be measured or even used as an optimization goal. However, there is no way to refine a plan to solve a particular memory performance issue. (3) The capability to enumerate plans in the search space. Our representation is based on a binary tree structure derived from the radix decomposition. However, due to the intrinsic property of FFT, not all binary trees represent valid FFT execution plans. Therefore, our representation must support enumeration of all valid trees.

In this section, we will introduce three foundational concepts upon which the REFFT architecture functions: binary tree ranking, leaf node permutations, and the data shuffling algorithm. Then, we will introduce the operation- and memory-oriented interpretations of the nodes and edges. For instance, internal nodes will represent twiddle nodes, while the leaf nodes will map to a direct solution called codelets, similar to their counterparts in FFTW.

3.1 Notations

A binary tree representation of FFT has two main defining components: the arrangement of the edges that connect the internal nodes and the permutation of the leaf nodes. We will denote $\boldsymbol{T}$ as a binary tree and $\boldsymbol{B_n}$ as the set of all binary trees with $\boldsymbol{n}$ internal nodes. It is possible to derive different variants of binary trees simply by altering the arrangement of the internal nodes and the edges that connect them. One way to accomplish this is through a series of binary tree rotations, in which the internal nodes are swapped to obtain different topographical structures while preserving the order of the leaf nodes. By 'leaf node order,' we define it as if one were to 'read the leaf nodes from left to right.

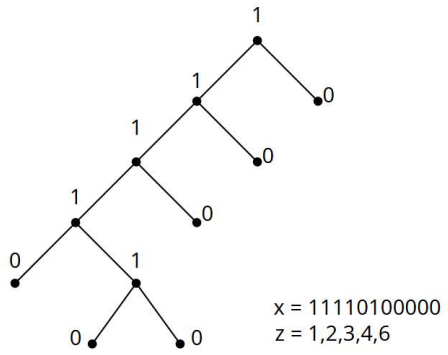
3.2 Binary tree ranking and unranking

In order to efficiently apply tree rotations and generate all possible tree variants, we implement a ranking mechanism based on lexicographic order proposed by Zaks et al. [17]. We first define a lexicographic representation of binary tree using a sequence of 1's and 0's, where a 1 represents an internal node and a 0 represents a leaf node. The order of the 1's and 0's is obtained by traversing a binary tree $\boldsymbol{T}$ in preorder. This will generate a string denoted $\boldsymbol{x}$ with n 1's and n+1 0's. However, given that not all binary sequences of length $\boldsymbol{2n+1}$ will necessarily generate a binary tree, the labeling

algorithm has a validation rule which is based on the "dominating property". As stated in Zaks' paper, if we consider a 0,1-sequence $\boldsymbol{x}$ and omit the final element which is always a 0 (given the nature of binary trees), such that the length of the sequence is $\boldsymbol{2n}$, then $\boldsymbol{x}$ is said to have the dominating property if, in any given prefix, the number of 1's is at least as the number of 0's. Furthermore, if in the entire sequence the number of 1's and the number or 0's are the same then that sequence maps to a unique binary tree with $\boldsymbol{n}$ internal nodes and $\boldsymbol{n+1}$ leaf nodes and is said to be "feasible". Feasible sequences will translate to valid and executable REFFT plans.

One way to imagine valid binary sequences that can be successfully mapped to a FFT solution is by considering the box-office problem: $\boldsymbol{2n}$ people are in line at the box office and $\boldsymbol{n}$ have 1 dollar bills, while the others have 2 dollar bills. Tickets cost 1 dollar each. When the box-office opens, it has no money in their drawer to disburse change. Given that each ticket costs 1 dollar, in how many ways can all the customers stand in line such that none will have to wait for change? All the possible solutions represent a feasible 1,0 binary sequence that REFFT can map to an FFT solution. The binary sequence can also be labeled by the integer sequence $\boldsymbol{z}$ where $\boldsymbol{z_i}$ is the ith position of the 1's in the binary sequence $\boldsymbol{x}$. Figure1 shows the sequence $\boldsymbol{x = 11110100000}$ which represents a valid binary tree and has $\boldsymbol{z = (1, 2, 3, 4, 6)}$.

Figure 1: A binary tree and its x and z sequences.



The set of binary tree $\boldsymbol{B_n}$, the 0,1-sequence $\boldsymbol{x}$ and the integer sequences $\boldsymbol{z}$ are in a 1-1 correspondence with one another. This enables us to index each tree and thus enumerate all binary trees, simply by toggling different index values from the range of 1 up to the Catalan value [15] $\boldsymbol{b_n}$ of the number of internal nodes $\boldsymbol{n}$:

$$\boldsymbol{b_n = \frac{1}{n+1}\binom{2n}{n}}$$

$$\boldsymbol{Index(z) \in B_n \rightarrow [1, b_n]}$$

The ranking function will map a binary tree to a specific index, while the unranking function does the inverse, meaning from a given index value and the number of internal nodes $\boldsymbol{n}$, it will retrieve the associated binary tree. REFFT utilizes the unranking function to easily explore the FFT plan space. To obtain the $\boldsymbol{z}$ sequence for the index $\boldsymbol{I}$, we make use of an auxiliary doubly indexed data structure $\boldsymbol{a_{i,j}}$ such that $\boldsymbol{0 \le j \le i-1}$:

$$\boldsymbol{a_{i,i-1} = 1} \text{ for all i,}$$

$$\boldsymbol{a_{i,0} = b_i = \frac{1}{i+1}\binom{2i}{i}} \text{ for all i,}$$

$$\boldsymbol{a_{i,j} = a_{i,j} + a_{i-1,j-1}} \text{ otherwise}$$

For example, if we consider a binary tree $\boldsymbol{T}$ with $\boldsymbol{n}$ = 7 internal nodes, we would have the doubly indexed auxiliary sequence as shown in table 1.

Table 1: The sequence $a_{i,j}$ for n = 7.

i / j	0	1	2	3	4	5	6
1	1						
2	2	1					
3	5	3	1				
4	14	9	4	1			
5	42	28	14	5	1		
6	132	90	48	20	6	1	
7	429	297	165	75	27	7	1

At this point, the lexicographic representation gives us the capability to numerically index or address binary trees. The next step is to do the inverse mapping, that is, deducing FFT implementation plans from their addresses. Unranking is the process of mapping an "index" or "address" in the search space to a valid binary tree for FFT implementation. We highlight here the algorithm for generating the $\boldsymbol{z}$ sequence of a binary tree $\boldsymbol{T}$ with $\boldsymbol{n}$ internal nodes, whose index is $\boldsymbol{Index(z) = t}$. We indicate with $\boldsymbol{a_{n,l}}$ the value indicated by row $\boldsymbol{n}$ and column $\boldsymbol{l}$ in Table 1 and $\boldsymbol{Terms}$ as the array that contains the tuples $\boldsymbol{(n, l)}$ that are utilized to generate the $\boldsymbol{z}$ sequence.

To better understand how the algorithm works, let's consider the following example: we want to know the $\boldsymbol{z}$ sequence of a binary tree with $\boldsymbol{n}$ = 5 internal nodes and with rank $\boldsymbol{t}$ = 38:

Given $\boldsymbol{t}$ = 38 and $\boldsymbol{n}$ = 5, we generate the $\boldsymbol{a_{i,j}}$ sequence:
Step 1 (Line 2): $\boldsymbol{zi}$ = 42 - 38 + 1 = 5
Step 2 (Lines 3 - 8): While $\boldsymbol{zi} > 0$
Iteration 1: $\boldsymbol{zi}$ = 5, $\boldsymbol{n}$ = 5, $\boldsymbol{l}$ = 4, so push $\boldsymbol{(5, 4)}$ in $\boldsymbol{Terms}$, $\boldsymbol{zi \leftarrow 5 - a_{5,4} = 5 - 1 = 4}$, $\boldsymbol{n \leftarrow n - 1}$
Iteration 2: $\boldsymbol{zi}$ = 4, $\boldsymbol{n}$ = 4, $\boldsymbol{l}$ = 3, so push $\boldsymbol{(4, 3)}$ in $\boldsymbol{Terms}$, $\boldsymbol{zi \leftarrow 4 - a_{4,3} = 4 - 1 = 3}$, $\boldsymbol{n \leftarrow n - 1}$,

Algorithm 1 Unranking algorithm

1: initialize $Terms$ and $rTerms$ as empty arrays
2: $zi \leftarrow a_{n,0} - t + 1$
3: **while** $zi > 0$ **do**
4: find l s.t. $a_{n,l} < zi$ or $a_{n,l} == 1$
5: push tuple (n, l) into $Terms$
6: $zi \leftarrow zi - a_{n,l}$
7: $n \leftarrow n - 1$
8: **end while**
9: $ls \leftarrow l$
10: $rTerms \leftarrow$ tuples of $Terms$ in reverse order
11: initialize $z[i] \leftarrow i + 1$ **for** $i = 0$ to ls ▷ the l value of the first (n, l) tuple of $rTerms$
12: $ax \leftarrow 1$
13: **while** $ax < len(rTerms)$ **do**
14: $a \leftarrow rTerms[ax]$
15: initialize y as array of 0s of length a_n ▷ a_n is the n value of the tuple $a_i = (n_i, l_i)$
16: **for** i = 0 **to** a_l: $y[i] \leftarrow i + 1$ ▷ a_l is the l value of the tuple $a_i = (n_i, l_i)$
17: **for** i = $(a_l - 1)$ **to** $len(z)$: $y[i + 1] \leftarrow z[i + 1] + 2$
18: $z \leftarrow y$
19: $ax \leftarrow ax + 1$
20: **end while**
21: **return** z

Iteration 3: zi = 3, n = 3, l = 2, so push $(3, 2)$ in $Terms$, $zi \leftarrow 3 - a_{3,2} = 3 - 1 = 2$, $n \leftarrow n - 1$
Iteration 4: zi = 2, n = 2, l = 1, so push $(2, 1)$ in $Terms$, $zi \leftarrow 2 - a_{2,1} = 2 - 1 = 1$, $n \leftarrow n - 1$
Iteration 5: zi = 1, n = 1, l = 0, so push $(1, 0)$ in $Terms$, $zi \leftarrow 1 - a_{1,0} = 1 - 1 = 0$, $n \leftarrow n - 1$
End of While $zi = 0$
Step 3 (Line 9): Reversing the sequence we have the following terms (n, l): [(1, 0), (2, 1), (3, 2), (4, 3), (5, 4)]
▷ Now we build the z sequence from $rTerms$:
Step 4 (Line 11): for $a_{i,j} = a_{1,0}$, set z = [i] = [1]
Step 5 (Line 12): $ax \leftarrow 1$
Step 6 (Lines 14 - 20): While $ax <$ $len(rTerms)$
Iteration 1: $y \leftarrow [1, z[0] + 2]$, $z \leftarrow y$, $ax \leftarrow ax + 1$
Iteration 2: $y \leftarrow [1, 2, z[1] + 2]$, $z \leftarrow y$, $ax \leftarrow ax + 1$
Iteration 3: $y \leftarrow [1, 2, 3, z[2] + 2]$, $z \leftarrow y$, $ax \leftarrow ax + 1$
Iteration 4: $y \leftarrow [1, 2, 3, 4, z[3] + 2]$, $z \leftarrow y$, $ax \leftarrow ax + 1$
Step 6 (Line 20): The final z sequence is $z = [1, 2, 3, 4, 9]$

Utilizing the unranking algorithm, REFFT can do fine-tuning of FFT plans that no other FFT libraries can do. Essentially it can generate all potential binary trees through incremental adjustments to an integer value—the address of a plan in the proposed representation. This incremental adjustment is effectively a rotation of the tree corresponding to the tree's address. Consequently, this approach enables efficient exploration for optimal FFT plans.

3.3 Leaf node set permutation

The node set permutation is denoted by an ordered set $l = (l_0, l_1, ...l_n)$, where l_i represents the i-th leaf node in the binary tree. Each li value ranges between 1 and 4, serving as the exponent of the codelet size expressed as 2^{l_i}. These components correspond to distinct DFT problems that REFFT can directly solve using its pre-built codelets. When seeking the optimal execution plan, it becomes crucial to explore all permutations of l to identify the most suitable candidate. Within a given DFT size, various codelet permutations may exist, differing in both the number of leaves $|l|$ and the magnitude of each leaf component l_i. While REFFT does not implement a ranking mechanism for leaf permutations similar to the binary tree rotation problem, it computationally generates all valid codelet permutations for the planner to utilize. The algorithm functions by employing an ordering mechanism, recursively generating permutations. More specifically, given two permutation sets l, l' we state that $l < l'$ if:

(1) $|l| < |l'|$,
(2) $l_i = l'_i$ for $i = 1, 2, m - 1$ and $l_m < l'_m$ for $i = m$.

For instance, considering a DFT of size $N = 16$: following the aforementioned criteria, the set of all valid leaf permutations L would begin with the set $l_0 = (16)$ and conclude with the final set $l'_m = (2, 2, 2, 2)$. There are two types of permutation sets: generator sets and derived sets. Generator sets serve as the cornerstone for the algorithm to derive other sets, hence the term 'derived'. These generator sets are either derived from other generator sets or originate from the root set, which consists of a single element $l_{root} = (N)$, where N is the size of the DFT size (in our example (16) is the root set). If N is less than 32, the root set will also serve as a potential solution; otherwise, it will function as an interim structure for deriving permutation sets that can be mapped to a feasible REFFT plan. A complete list of all the codelet permutations are illustrated in figure 2. The root set $l_{root} = (16)$ generates two additional sets, which also serve as generator sets: $l_0 = (2, 8)$ and $l_1 = (4, 4)$. Subsequently, from l_0 the algorithm generates the derived permutation $l_2 = (8, 2)$, while from l_1 another generator set is obtained $l_3 = (2, 2, 4)$. Next, from l_3, two additional derived permutations are derived: $l_4 = (2, 4, 2)$ and $l_5 = (4, 2, 2)$. Finally, from l_5,the algorithm derives the final set $l_6 = (2, 2, 2, 2)$, which does not generate other permutations, thus leading to the termination of the algorithm.

Each leaf permutation corresponds to a distinct subspace of all potential REFFT plans. The quantity of plans is directly proportional to the number of tree rotations achievable by each permutation. Equation 1 provides the total number of plans generated by all permutations:

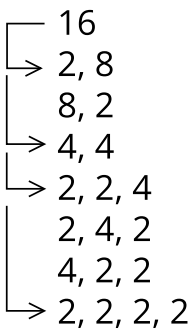
$$TotalPlans_N = \sum_i |l_i| * CatalanNumber(|l_i - 1|) \quad (1)$$

Given that $|l_i - 1|$ is the number of internal nodes n of a REFFT binary tree, we can rewrite equation 1 as follows:

$$TotalPlans_N = \sum_i |l_i| * \frac{1}{n+1}\binom{2n}{n} \quad (2)$$

In table 2, we illustrate the total number of plans for $N = 16$ along

Figure 2: Leaf permutations for N = 16.



with the number generated by each leaf permutation.

Table 2: Number of Possible plans for $N = 16$.

Codelet Permutation	Possible Plans (Catalan Number)
16	1
(2, 8)	1
(8, 2)	1
(4, 4)	1
(2, 2, 4)	2
(2, 4, 2)	2
(4, 2, 2)	2
(2, 2, 2, 2)	5
Total REFFT Plans	15

3.4 Node edges and Data shuffling algorithm

In FFT algorithms, data shuffling is an operation that is particularly consequential to the memory performance. As we traverse down the binary tree and execute the codelets associated with leaf nodes, data shuffling becomes inevitable. The operation entails that the input data sequence does not align with the expected order (referred to as natural order). For radix-2 FFT problems, shuffling is characterized by the bit-reversal order. In other words, if we represent the index of a data input in binary notation, the original input sequence and the resulting shuffled sequence are in reverse order. For instance, in a three-stage radix-2 Cooley-Tukey FFT of size $\boldsymbol{N = 8}$, the input sequence for index $\boldsymbol{n = 3}$ in binary notation is **011**, while the resulting shuffled sequence would actually present the reverse bit sequence **110**, equivalent to 6 in decimal notation. Hence, where we anticipate finding the input at location $\boldsymbol{n = 3}$, it will instead appear at location $\boldsymbol{n = 6}$. Consequently, explicit handling and tuning of all shuffling effects becomes crucial to guarantee the correct output sequence upon the completion of a REFFT plan's execution. To address this, we devised a mathematical model and a corresponding algorithm to systematically track and compute the cumulative shuffling effects at any node within the REFFT binary tree structure.

Each node in the binary tree is connected to the rest of the structure by up to three directed edges. By examining an internal node and one of its edges, we can derive another tree structure by condensing the entire subtree structure connected to the opposite side of the edge into a single node. This node's value would equal the sum of all its leaf nodes. This value holds computational significance as it determines the behavior of data shuffling resulting from decomposing a large DFT into numerous smaller DFT problems. Given that each node has four potential directed edges, we refer to these values as *super-right, super-left, sub-right* and *sub-left*. Figure 3 illustrates a standard node alongside its labeled edges: $\boldsymbol{supr}$, $\boldsymbol{supl}$, $\boldsymbol{subr}$ and $\boldsymbol{subl}$. These edge labels are purely conventional and denote the direction of the edge relative to the internal node it connects. Indeed, when considering two connected nodes, the edge can be either super-right/sub-left or super-left/sub-right, depending on the node in question.

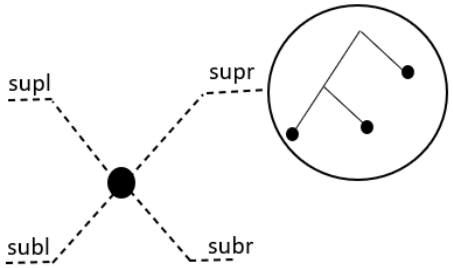
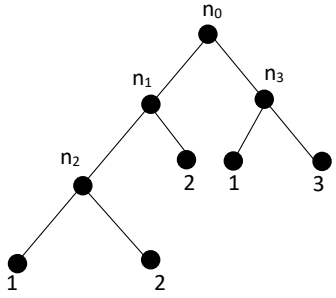


Figure 3: Edge naming conventions for REFFT nodes.

To further illustrate the concept, let us consider the tree in figure 4. Within this structure, we can observe the internal nodes represented by $\boldsymbol{n_i}$ along with the various leaf nodes and their respective values expressed as power-of-two exponents, denoted as $\boldsymbol{exp_i}$ of the direct DFT size, such that $\boldsymbol{exp_i = log_2}$(DFT problem size of leaf node i).

Figure 4: Edge naming conventions for REFFT nodes.



For the internal node n_1 the sub-right value is equal to 2, the super-right is 4 and the sub-left value is 3. Table 3 provides a comprehensive overview of all the edge values associated with each internal node.

Table 3: Edge values for each internal node of the binary tree shown in Figure 4.

internal node	supr	subr	supl	subl
n_0	0	4	0	5
n_1	4	2	0	3
n_2	6	2	0	1
n_3	0	3	5	1

The data shuffling algorithm leverages the edge values to ascertain the data shuffling effects on the input sequence. This algorithm takes as input a twiddle node of the REFFT binary tree and produces an array of tuple t. each containing three components $t_i = (a_i, b_i, c_i)$. The resulting shuffled position n' of a data input in position n is computed based on the summation outlined in equation 3.

$$n' = \sum_{k} \frac{n \mod a_k}{c_k} * b_k \tag{3}$$

The data shuffling algorithm encompasses two variants tailored to distinct scenarios within the REFFT execution life-cycle: traversing a left subtree structure and reaching the final leaf node of the binary tree. In the former scenario, the execution transitions into a left subtree structure when an internal node's left child is an internal node. This requires continued descent through the left subtree until a left leaf node is encountered. Upon reaching this condition, the REFFT planner understands that data shuffling has taken place within the input sequence, thereby invoking the data shuffling algorithm. Conversely, the latter scenario invariably materializes, as during the concluding phase of REFFT execution, the output sequence inevitably assumes some degree of disorder due to the topography of the tree structure. The sole scenario where the output sequence naturally maintains order is when the REFFT executes a binary tree decomposition comprising solely left internal nodes and right leaf nodes, with the exception of the final left node, which serves as a leaf. In this configuration, the binary tree mirrors a complete decimation in time setup, wherein the input initially lacks order and, as the execution advances, the resulting output sequence gradually aligns in order. In all other cases, REFFT is required to "unshuffle" the output sequence, rearranging all out-of-order elements into their anticipated positions.

Algorithm 2 illustrates the process of deriving data shuffling parameters based on the tree structure. The algorithm accepts a reference to the starting internal node as input and produces an array of tuples comprising metadata necessary for calculating shuffled positions. It begins by processing the initial internal node provided as input, computing tuples for both its left and right children. Depending on specific conditions, the initial value of the b component may vary, subsequent to which tuples are derived and appended to the output array. Upon completion, the algorithm proceeds upwards within the binary tree structure, transitioning from the current internal node in execution to its parent. The lv and rv variable function as flags to keep track of whether we have accounted for the left child and right child nodes respectively, as we ascend up the binary tree. The termination condition arises immediately after generating tuples associated with the tree's root node. At this point, the while loop ceases execution, and the array T array containing all the tuples is returned as the output.

Let us illustrate the functionality of Algorithm 2 through a practical example, focusing on the REFFT decomposition as depicted in Figure 4. Specifically, we examine the shuffling effect within the internal node n_2, serving as the input parameter $startNode$ for the algorithm.

Initially, the algorithm initializes core variables and the while loop entry condition, then enters the main loop. Since the lv flag is false, it branches into the code block and initializes the value of b to $supr$. None of the conditions trigger the addition of $subr$, thus the algorithm proceeds to generate tuples for the left child node. As the value of the left leaf node is 1, it generates a single tuple **[2, 256, 1]**, which is pushed into the T array. The execution flow continues, checking if the right leaf node has been visited, indicated by the value of the rv variable.Upon entering the code block in this case, it performs similarly to the left leaf node, setting b to the $supr$ value, and then proceeds to generate the tuples **[4, 64, 2]**, **[8, 128, 4]**, which are also pushed to the array T. Before ascending the binary tree, it determines whether $currNode$ itself is a left or right child tree with respect to its parent node, and sets the lv and rv flags accordingly. The execution continues, and $currNode$ now references n_1. In this case, only rv is false, so only the right child of n_1 generates tuples. Again, the lv and rv flags are set accordingly and the execution continues to ascend the binary tree. Now, $currNode$ so it generates tuples for its right child node and then sets the $finished$ variable to true. This serves as the exit condition for the while loop, and the T array containing all the tuples is returned as the result:

T = [[2, 256, 1], [4, 64, 2], [8, 128, 4], [16, 16, 8], [32, 32, 16], [64, 1, 32], [128, 2, 64], [256, 4, 128], [512, 8, 256]]

Table 4 illustrates the tuples generated for each internal node visited, starting from n_2, the starting node. To determine the shuffled position, we employ Equation 3 for each physical position p of the input array, utilizing the components a_k, b_k, c_k of our generated T array to compute the shuffled position p'.

Table 4: Tuples Generated by Algorithmn 2 and binary tree from Figure 4.

currNode	lv, rv	(supr, subr, supl, subl)
n_2	False, False	(6, 2, 0, 1)
left tuples	[2, 256, 1]	
right tuples	[4, 64, 2], [8, 128, 4]	
n_1	True, False	(4, 2, 0, 3)
right tuples	[4, 64, 2], [8, 128, 4]	
n_0	True, False	(0, 4, 0, 5)
right tuples	[4, 64, 2], [8, 128, 4]	

Algorithm 2 Data Shuffling algorithm

Input: **StartNode**
initialize empty array T
$a \leftarrow 1$
$c \leftarrow 0$
$lv \leftarrow False$
$rv \leftarrow False$
$finished \leftarrow False$
$currNode \leftarrow StartNode$
while $finished \neq True$ **do** ▷ Main Loop:
  if $lv = False$ **then** ▷ Visit the left child node :
    $lv \leftarrow True$
    $b \leftarrow supr$
    if right child ≠ TWIDDLE **or** right child = TWIDDLE
    and (left child = TWIDDLE **or** currNode ≠ startNode
    or currNode = ROOT)
    then
      $b \leftarrow b + subr$
    for i = 0 **to** left child value **do** ▷ Tuples for left child:
      $tuple_i \leftarrow (2^{a_i}, 2^{b_i}, 2^{c_i})$
      $b \leftarrow b + i$
      $c \leftarrow a$
      $a \leftarrow a + 1$
      push $tuple_i$ into **T**
    end for

  if $rv = False$ **then** ▷ Visit the right child node :
    $rv \leftarrow True$
    $b \leftarrow supr$
    if right child = TWIDDLE **and** left child ≠ TWIDDLE
    and right child value > 1 **and** currNode ≠ ROOT
    Then
      $b \leftarrow b + subl$
    for i = 0 **to** right child value **do** ▷ Tuples for right child:
      $tuple_i \leftarrow (2^{a_i}, 2^{b_i}, 2^{c_i})$
      $b \leftarrow b + i$
      $c \leftarrow a$
      $a \leftarrow a + 1$
      push $tuple_i$ into **T**
    end for

  if $currNode$ = LEFT CHILD **then** ▷ Traverse up:
    $lv \leftarrow True, rv \leftarrow False$
  else if $currNode$ = RIGHT CHILD **then**
    $lv \leftarrow False, rv \leftarrow True$

  if $currNode$ = ROOT **then**
    $finished \leftarrow True$
  else
    $currNode \leftarrow currNode.parent$
end while
Output: **T**

4 WORKFLOW, PLANNER AND EXECUTION IN REFFT

In this section, we provide an overview of the structure of the proposed Rotational Enumerated FFT Library (REFFT) and explain how its internal mechanisms facilitate the execution of Discrete Fourier Transform (DFT) by exploring an extensive set of potential solutions and subsequently selecting those that exhibit best performance. Specifically, we delve into the selection of plans, the contents of their internal details, the translation of plan directives by the importer into an executable FFT plan, and the execution process utilizing the REFFT library.

4.1 Plan Structure

Similar to FFTW, REFFT employs tensors to represent Discrete Fourier Transform (DFT) problems. These tensors contain elements for a single DFT execution, organized within a nested tensor structure to delineate core components: *plan*, *plan passes*, *plan pass data* and *plan slices*. A REFFT Plan P is a tensor that contains the core data and is defined as $P = (N, PA, I, O, T)$, where N is and integer value and contains the size of the DFT. The nested tensor PA embodies the plan passes, I and O represent the input array pointer output array respectively and T denotes a pointer to the internal buffer array. Each plan pass, denoted as $PA = (PD, PA*)$ and contains details regarding a single DFT problem, alongside a reference to the subsequent plan pass for execution. The sequence of plan passes mirrors the order of the codelet permutation set. The tensor $PD = (I, O, si, so, SL, n)$ stores all the granular data needed to execute the codelet itself, including pointers to the input and output buffers, as well as stride information. SL, also called *plan slice*, is another tensor containing information for a single codelet instance. In particular, SL is defined as $SL = (F, TWD)$, where F is an array comprising the data positions of both the inputs and the outputs, and TWD is an array serving as a double buffer for twiddle operations. Both arrays are generated during the compilation phase and are essential for minimizing overhead during the execution of the plan. Hence, REFFT plans exhibit a nested tensor structure, characterized by the format: *plan(PA(PD(PS,...),...),...)*. This design adopts a decoupled and modular approach, emphasizing the retention of comprehensive information. Such caching strategies aim to facilitate notable performance enhancements.

Similar to FFTW, REFFT employs plans to retain metadata needed for executing particular FFT implementations. These plans contain precise directives and auxiliary data comprehensible to the REFFT importer, facilitating the translation into executable C-based FFT solutions. Additionally, REFFT extends this capability by allowing users to formulate plans using an algebraic notation, which directly corresponds to a feasible FFT decomposition. The $*$ symbolizes an internal node, while decimal integers to its left and right signify leaf nodes. Parentheses are employed to group binary operators with their operands and other elements. Following algebraic principles, expressions within inner parentheses are executed first, with execution proceeding from left to right and from inner to outer parentheses in the expression. A notable difference arises in the representation of problem sizes between the user-friendly expression and the binary tree representations. While the user-friendly expression denotes DFT problem sizes in both base 10 and powers of 2 (e.g., 2, 4, 8, or 16), the binary tree representations express the problem size using the base logarithm notation log_2(problem size). The latter notation is employed in plan implementation due to its efficiency in reducing overall planning time. Conversely, the former notation enhances user experience by offering a more intuitive approach to expressing FFT decompositions.

4.2 Emperical Performance Tuning

REFFT employs empirical tuning to explore and determine the best performing plan. This exploration is facilitated by its utilization of a regression-based search strategy. We need to point out that the best-performing plan found does not necessarily represent the best achievable plan within the search space. The search strategy yields two key outputs: the leaf node permutation $\boldsymbol{l}$ and the binary encoding of the tree rotation $\boldsymbol{X}$. These outputs are then translated into a standardized plan that REFFT utilizes for the actual low-level implementation. The following presents an example of an REFFT plan compared side by side with an FFTW plan for a problem size of 2^{17}.

```
 ( fftw-3.3.8 fftw_wisdom #x4be12fff ...
  (fftw_dft_vrank_geq1_register 0 #x11bdd #x11bdd #x0 ...)
  (fftw_codelet_t1_16 0 #x11bdd #x11bdd #x0 #x45ebf64b ...)
  (fftw_codelet_n1_64 0 #x11bdd #x11bdd #x0 #xbaa459d1 ...)
  (fftw_dft_vrank_geq1_register 0 #x11bdd #x11bdd #x0 ...)
  (fftw_codelet_t1_16 0 #x11bdd #x11bdd #x0 #x2dbefd0a ...)
  (fftw_codelet_t1_8 0 #x11bdd #x11bdd #x0 #x66101a7a ...)
  )
```

```
( refft-1.0.0 size 131072
refft_info  (((4*8)*16)*16)*16 111100000 2_3_4_4_4
refft_pass dtfsz 4 strin 32768 strout 1 n1 4 n2 1
refft_twd n1 8 n2 4
refft_pass dtfsz 8 strin 4 strout 4 n1 8 n2 4
refft_twd n1 16 n2 32
refft_pass dtfsz 16 strin 32 strout 32 n1 16 n2 32
refft_twd n1 16 n2 512
refft_pass dtfsz 16 strin 512 strout 512 n1 16 n2 512
refft_twd n1 16 n2 8192
refft_pass dtfsz 16 strin 8192 strout 8192 n1 16 n2 8192
)
```

A major distinction between between FFTW and REFFT lies in their approach to solution space exploration. While FFTW excels in providing rapid solutions, it operates without assumptions regarding the extent or granularity of the solution space it explores. In contrast, REFFT meticulously maps all exploitable solutions by considering permutation sets of a given problem size and ranking their corresponding tree rotations. This capability proves crucial, as different plans exhibit distinct stride distributions during DFT execution, significantly impacting memory performance. Therefore, REFFT evaluates plans to ensure tailored solutions optimized for the memory architecture of a specific machine.

4.3 The Planner

The REFFT planner comprises two primary components: the plan explorer and the importer. The former is responsible for searching for the best-performing plan of a given size, while the latter imports the plan directives as input and constructs the executable plan as output. Currently, the plan explorer implements a regression-driven strategy to explore and identify the most efficient plans. It accomplishes this by navigating the solution space in two steps: first, identifying the optimal codelet permutation, and then determining the optimal tree rotation for the given codelet permutation.

Currently, the REFFT architecture only supports codelet sizes of **2**, **4**, **8** or **16**. Given that these sizes are preconstructed, future efforts will include expanding the supported sizes to include **32** and **64** among the possible codelet sizes.

To elaborate on the first step, the codelet permutation is a tensor composed of an ordered set of codelets, each potentially sized as **2**, **4**, **8**, or **16**. For a DFT problem of size $\boldsymbol{N}$, this permutation entails a frequency count for each codelet and a specific order in which they appear. Table 5 illustrates an example of a DFT of size 1,024 and some of the possible codelet permutations.

Table 5: Example of possible codelet permutations for $N = 1024$.

N = 1024
1, 3, 3, 3
2, 2, 2, 4
1, 1, 1, 3, 4
1, 1, 4, 4
1, 3, 3, 3
2, 2, 2, 4
2, 4, 4

To expedite the searching, we conducted a regression analysis to estimate the correlation between codelet permutations and overall execution time. Specifically, we profiled and benchmarked all plans ranging from size $\boldsymbol{N} = 2^3$ up to $\boldsymbol{N} = 2^{12}$. Next,we expanded our data acquisition efforts by sampling plans for DFT sizes $\boldsymbol{N} = 2^{13}$ up to $\boldsymbol{N} = 2^{20}$, considering the exceptionally vast theoretical plan space (for size $\boldsymbol{N} = 2^{20}$ there are potentially more than 168 billion possible plans). For the remaining values ranging from $\boldsymbol{N} = 2^{21}$ up to $\boldsymbol{N} = 2^{30}$, we derived them through linear projections based on coefficients from known sizes. Beginning with extrapolated projections, the plan explorer addresses the permutation set as follows: for a DFT of size $\boldsymbol{N} = 2^{\boldsymbol{x}}$, it selects two codelet sizes $\boldsymbol{l_1} = 2^{\boldsymbol{a}}$ with a frequency $\boldsymbol{f_1}$ and score $\boldsymbol{s_1}$ and $\boldsymbol{l_2} = 2^{\boldsymbol{b}}$ with frequency $\boldsymbol{f_2}$ and score $\boldsymbol{s_2}$ such that $\boldsymbol{s_1} < \boldsymbol{s_2}$. At this point, we have can have one of two possible scenarios:

(1) $\boldsymbol{x} = \boldsymbol{f_1} * \boldsymbol{a}$,
(2) $\boldsymbol{x} = \boldsymbol{f_1} * \boldsymbol{a} + \boldsymbol{f_2} * \boldsymbol{b}$.

In the first case, the frequency $\boldsymbol{f_1}$ of $\boldsymbol{a}$ can be maximized to allow the DFT size to be completely decomposed into $\boldsymbol{f}$ smaller problems of size $2^{\boldsymbol{a}}$. For example, considering a DFT of size $\boldsymbol{N} = 2^{12}$ and assuming that $\boldsymbol{l_1} = 2^3 = 8$ has the lowest score in the hash table for the given DFT problem size, it becomes possible to decompose $\boldsymbol{N}$ with only codelets of size $\boldsymbol{l_1}$ with a frequency $\boldsymbol{f_1} = 4$.

In the second scenario, solving the DFT problem with only one codelet size type is not feasible. In this case, the plan explorer incorporates another codelet size with the second-lowest score to complete the decomposition. For example, for a DFT size $\boldsymbol{N} = 2^{19}$, the two lowest-scoring codelet sizes are $\boldsymbol{l_1} = 2^3 = 8$ and $\boldsymbol{l_2} = 2^2 = 4$, with exponents $\boldsymbol{a} = 3$ and $\boldsymbol{b} = 2$ respectively. The planner resolves the permutation by assigning frequencies $\boldsymbol{f_1} = 5$ and $\boldsymbol{f_2} = 2$, satisfying the equation $19 = 5 * 3 + 2 * 2$.

After generating the permutation, the plan explorer proceeds to search for the optimal-performing REFFT binary tree. It's worth

noting that the number of possible trees equals the Catalan number of the size of the leaf permutation set. To illustrate, considering $N = 2^{19}$ and the permutation set has a size $|l| = f_1 + f_2 = 5 + 2 = 7$, we potentially have 429 binary tree decompositions to choose from. Each binary tree can be assigned an ordered integer value, enabling the plan explorer to start ranking from 1 and benchmark a small percentage of the binary tree decompositions to select the optimal-performing one, within the limits of the naive strategy implementation and capable of outperforming FFTW. Empirical evidence indicates that the optimal-performing binary tree typically lies within the top 10 percent of the ranking space. Using the earlier example, the optimal-performing binary tree would have a ranking value r such that $429*.09 \leq r \leq 429$. At this stage, the plan explorer has a set of promising implementation candidates. Subsequently, the plan explorer selects one of three possible techniques:

- For small N values (less than $N \leq 2^{19}$), the plan explorer evaluates all tree decompositions to find the best-performing one among them.
- For larger N values ($2^{20} \leq N \leq 2^{24}$), the plan explorer evaluates the top ten percent of the available plans, one by one, to identify the best-performing decomposition.
- If N is very large ($N \geq 2^{25}$), the plan explorer evaluates the top ten percent of the available plans, sampling every jth rank, to find the best-performing decomposition.

Once the plan explorer has identified a plan solution, it generates the final REFFT plan comprehensible for the plan importer. The primary strategy of REFFT is to generate as much as possible at compile time to maximize memory access efficiency, leading to faster execution. To achieve this goal, the plan importer leverages preconstructed codelets that serve as a scaffold, replicable and populated with the necessary data for a single codelet instance. This approach is particularly effective in the twiddle computation space, where twiddle operands can be determined during the compile phase by knowing the size N of the DFT and the problem decomposition structure. In the case of REFFT, such parameters are expressed via the binary tree generated from the codelet permutation set and the tree edge layout x. As a result, it becomes possible to calculate all possible twiddle data during the plan compilation phase. These twiddle values can then be cached in a global space where each codelet instance can reference them, or they can be double-buffered so that each codelet has direct access to its own copy of the twiddle operands. This approach reduces the number of required address-related instructions during the execution phase of the REFFT plan. However, the trade-off is a somewhat increased plan memory footprint. Additionally, input data for all codelet instances is aligned to reduce the need for pointer address calculations. Expressions are grouped to favor cache locality and exploit the instruction-level parallelism of modern architectures. As previously mentioned, REFFT leverages pre-built codelets for sizes **2**, **4**, **8**, and **16**. Any DFT problem will be solved with a mix of these codelet sizes.

4.4 Plan Execution

The execution of REFFT plans is conceptually much simpler than other components of REFFT's architecture. During this phase, REFFT aims to group or coalesce memory accesses with similar patterns, with the expectation that batch processing of memory accesses will yield the best overall performance. Any form of irregular memory access is considered overhead that REFFT strives to avoid. The design of the plan executor in REFFT is similar to that of FFTW, facilitating easy swapping of one with the other. Once invoked, execution begins, and the executor iterates through all plan passes, calling the function pointer that references one of the prebuilt codelets and providing the plan pass data as an input parameter. The codelet interface is generalized with the goal of reducing procedural calling overhead, while embedding the complexities of FFT execution logic within the data itself. Furthermore, REFFT plans can be reused with other data input sets as long as they are of the same DFT size. Users simply need to swap out the reference of the old input array with the new one and provide an output array reference for storing the results.

5 EVALUATION

We will first examine REFFT from both performance and correctness perspectives. Following this, we will conduct a detailed analysis of its interactions with various architecture features, utilizing hardware counters to aid in our assessment.

5.1 Experimental Setup

We selected three current CPU architectures for experimentation: the Intel i7 1300F, AMD Ryzen 9 5900, and AMD Ryzen 7845HX. All experimental computers are equipped with Ubuntu 20 and GCC, and include the FFTW library version 3.3.8, which serves as the baseline implementation of FFT against which REFFT is compared. Table 6 provides further information regarding each CPU, including clock frequency and various memory capacities.

Table 6: Experimental Platform Specification.

Model	Frequency	L1 Cache	L2 Cache	L3 Cache	SYSTEM RAM
1300F	2.1 GHz	80KB	2MB	30MB	64GB
5900	3.7 GHz	64KB	6MB	64MB	64GB
7845HX	3 GHz	64KB	1MB	64MB	64GB

Testing was conducted on FFT problem sizes ranging from 64 (2^6) to 32M (2^{25}). To mitigate noise during testing, each size was executed multiple times. Specifically, for each size, the number of executions was normalized so that the total accumulated time was approximately equal to a threshold ranging from 100 milliseconds for smaller sizes up to 3,000 milliseconds for the larger sizes. This approach ensures a consistent relative error measurement when evaluating performance. Additionally, after each execution, a cache flushing routine was performed to clear the cache for the next iteration.

5.2 Correctness

To verify the correctness of REFFT, we compare its outputs with those generated by FFTW, which serves as the source of truth.

Since the outputs of both libraries are complex number arrays, we compare the differences among the respective values of the

arrays. Therefore, we utilize a well-established metric for error analysis called the Root Mean Square Error (RMSE). The formula for RMSE is shown in Equation 4, where N represents the problem size, while $\vec{FW}$ and $\vec{RF}$ represent the output arrays of FFTW and REFFT, respectively.

$$RMSE(\vec{FW}, \vec{RF}) = \sqrt{\frac{\sum_{i=0}^{N-1} [\Re(FW_i) - \Re(RF_i))^2 + (\Im(FW_i) - \Im(RF_i))^2]}{2N}} \quad (4)$$

Figure 5 shows the RMSE for all the problem sizes. As shown, the RMSE values are between $1E^{-7}$ and $1E^{-9}$, average in the order of $1E^{-8}$. These errors are notably small for the double-precision data type, indicating that the REFFT library effectively implements the DFT/FFT algorithms.

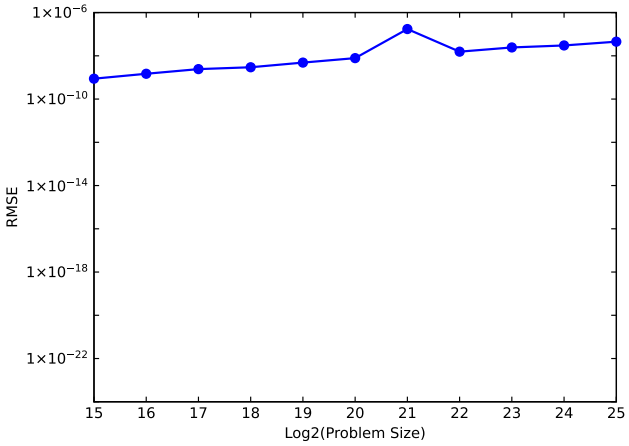
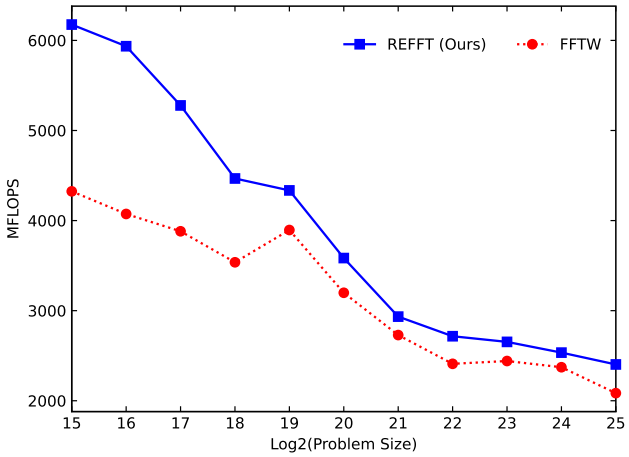


Figure 5: RMSE of REFFT vs. FFTW.

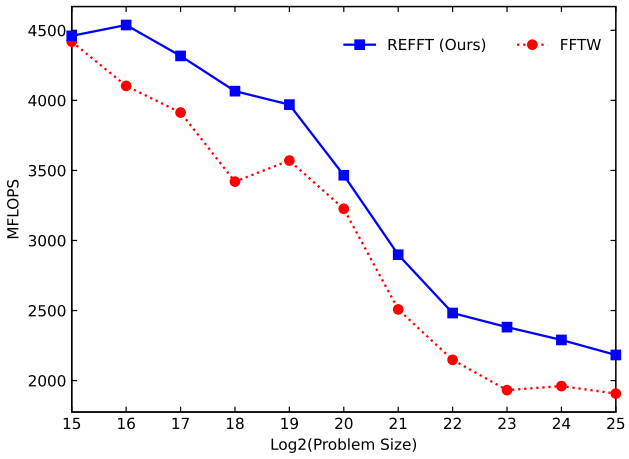
5.3 Performance Evaluation

Performance for both REFFT and FFTW is measured using the number of processor cycles. This metric is made possible by utilizing hardware counters available on modern CPU architectures, accessed via a library called PAPI. The number of cycles is then converted to absolute time using the processor frequency. To facilitate a better comparison of benchmark results between REFFT and FFTW, we plot performance using a metric denoted as Million Floating Point Operations Per Second (MFLOPS). MFLOPS is calculated as $(5N * log_2(N))/t$ where N is the problem size and t is the execution time in microseconds. This formula has been employed in the FFTW paper [6] and other research papers on FFT.

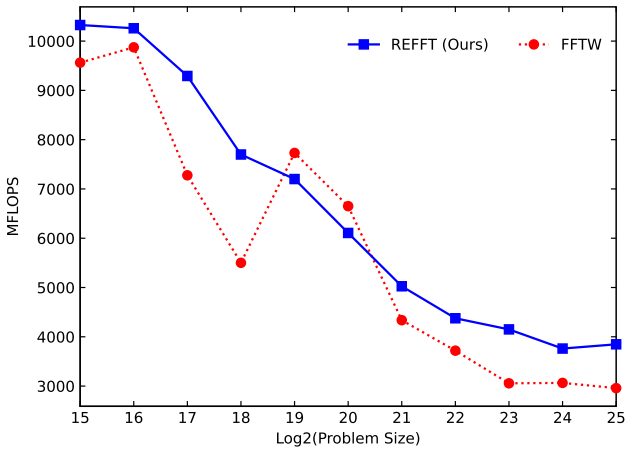
Figure 6 illustrates the performance evaluation across the three platforms. The plots yield two key observations. Firstly, REFFT consistently outperforms FFTW across all platforms. For instance, in the case of the Ryzen 9 7845HX, the average speedup across all sizes is approximately 17.3%, with peaks reaching as high as 45.7%. Similar performance gains have been observed for the other CPU architectures. Secondly, as the problem size increases, the advantage of REFFT over FFTW diminishes to some extent. This phenomenon is likely attributable to the "naive" implementation of our searching module. While it excels in finding better-performing



(a) Performance on Ryzen 9 7848HX speedup: mean 17.3%, max 45.7%

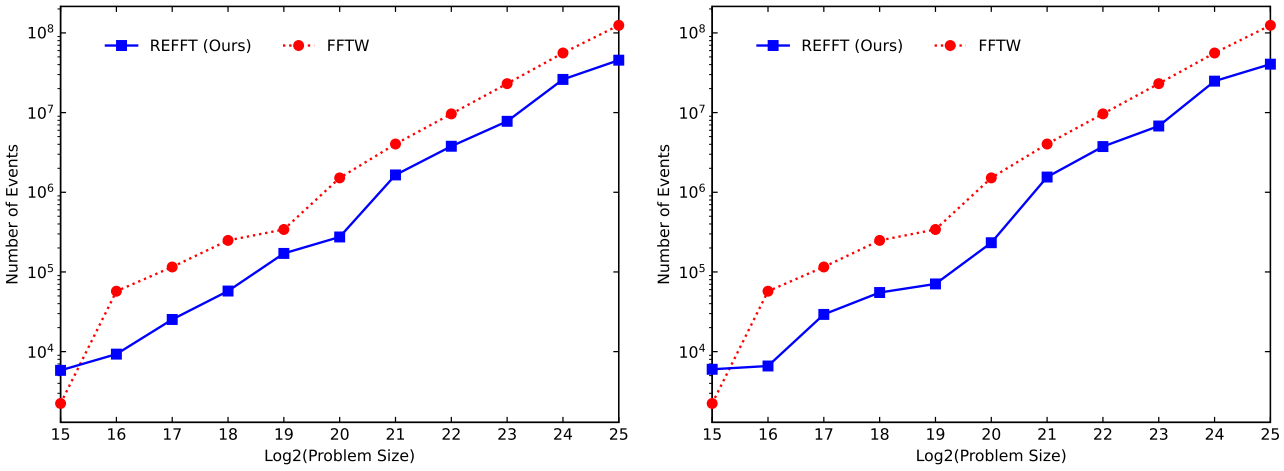


(b) Performance on Ryzen 9 5900 speedup: mean 11%, max 23.3%



(c) Performance on I7 13700F speedup: mean 18%, max 40%

Figure 6: Performance Comparison: REFFT vs. FFTW.



(a) Number of Data Translation Lookaside Buffer Misses

(b) Number of Level 2 Data Cache Misses

Figure 7: Hardware Counter Profiling: REFFT vs. FFTW.

plans for smaller problems, the exponential increase in potential plans for larger sizes poses a challenge. The planner conducts only a superficial search within a short time frame, aiming to identify "sufficiently" performing solutions, which nonetheless represent winning plans.

5.4 Performance Profiling

Next, we analyze the performance profile of REFFT in contrast with FFTW to better understand the source of the performance improvement. Our focus is on understanding why the selected plans indeed offer improved performance, with particular attention given to their impact on memory performance.

The profiling is conducted using hardware counters, which are available on almost all modern processors. Hardware counters allow engineers to monitor counts of specific architectural events, which can reveal software behavior that would otherwise go undetected or be difficult to extract from the noise of other concurrent processes. We focus on two events closely tied to memory performance: TLB_DM, which records the number of data translation lookaside buffer misses, and L2_DCM, which counts the number of level 2 data cache misses. Both of these events are individually supported by the Ryzen architectures that we tested.

However, it's important to note that our profiling analysis encountered a limitation: the recent Intel processor utilized in our experiment is not yet supported by the PAPI counters. Consequently, for this specific Intel CPU, we were unable to utilize hardware counters for further analysis. Instead, we relied on the gettimeofday function to obtain execution time data. Despite this limitation, we were still able to evaluate the contemporary Intel CPU's performance effectively.

Figure 7 illustrates the profiling of TLB_DM and L2_DM events for both FFTW and REFFT on the Ryzen 9 5900. The figures demonstrate that REFFT outperforms FFTW in these two event metrics. The reduced number of translation lookaside buffer (TLB) misses strongly correlates with the efficiency of our library. Conversely, L2_DCM illustrates how REFFT's advantage slightly diminishes as the problem size increases. This observation aligns well with REFFT's lack of larger-size codelets compared to FFTW, resulting in diminishing performance as the FFT problem size grows. Larger-size codelets enable greater symmetry and offer more optimization opportunities for FFT problems, while reducing overhead from managing smaller-size codelet executions. Therefore, as the FFT size increases, having larger prebuilt codelet sizes becomes advantageous. The results affirm the validity of our core hypothesis: REFFT effectively selects plans that lead to more efficient memory management, thereby enhancing overall performance.

6 CONCLUSION

In conclusion, the Rotational Enumerated FFT Library (REFFT) presented in this paper offers a promising approach to computing Discrete Fourier Transforms (DFT) by exploring the decomposition space of memory accesses in FFT. Through the utilization of binary tree structures, REFFT enables effective tuning for memory efficiency, allowing fine-grained navigation of a vast solution space. Our comparative analysis against FFTW, a widely adopted FFT library, demonstrates an average speedup of **14.3%** across three contemporary Intel and AMD processors. These findings underscore the potential of REFFT to significantly enhance computational performance across diverse processor architectures. With its innovative methodologies and tangible performance gains, REFFT represents a valuable contribution to the field of FFT optimization and memory management.

REFERENCES

[1] L. Bluestein. 1970. A linear filtering approach to the computation of discrete Fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18, 4, 451–455.

[2] Y. Chen and X. etc. Cui. 2010. Large-scale FFT on GPU clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 315–324.

[3] J.W. Cooley, P.A.W. Lewis, and P.D. Welch. 1970. The fast fourier transform algorithm: programming considerations in the calculation of sine, cosine and

laplace transforms. *Journal of Sound and Vibration*, 12, 3, 315–337. DOI: https://doi.org/10.1016/0022-460X(70)90075-1.

[4] J.W. Cooley and J.W. Tukey. 1965. An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation*, 19, 90, 297–301.

[5] P. Duhamel and M. Vetterli. 1990. Fast fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19, 4, (Apr. 1990), 259–299.

[6] M. Frigo and SG Johnson. 1997. The Fastest Fourier Transform in the West.

[7] Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of fftw3. *Proceeding of the IEEE*, 93, 2, 216–231.

[8] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. 2008. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 1–12.

[9] Liang Gu, Xiaoming Li, and Jakob Siegel. 2010. An empirically tuned 2d and 3d fft library on cuda gpu. In *Proceedings of the 24th ACM International Conference on Supercomputing* (ICS '10). ACM, Tsukuba, Ibaraki, Japan, 305–314. ISBN: 978-1-4503-0018-6.

[10] Liang Gu, Jakob Siegel, and Xiaoming Li. 2011. Using gpus to compute large out-of-card ffts. In *Proceedings of the international conference on Supercomputing* (ICS '11). ACM, Tucson, Arizona, USA, 255–264. ISBN: 978-1-4503-0102-2.

[11] A. Nukada and S. Matsuoka. 2009. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 1–10.

[12] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. 2008. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 1–11.

[13] CM Rader. 1968. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56, 6, 1107–1108.

[14] Salvatore Servodio and Xiaoming Li. 2021. An efficient shuffle-light fft library. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 1–10. DOI: 10.1109/IPCCC51483.2021.9679431.

[15] Richard P. Stanley. 2015. *Catalan Numbers*. Cambridge University Press.

[16] Shmuel Winograd. 1980. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970364. DOI: 10.1137/1.9781611970364.

[17] S. Zaks. 1980. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10, 1, 63–81. DOI: http://dx.doi.org/10.1016/0304-3975(80)90073-0.