SSFFT: Energy-Efficient Selective Scaling for Fast Fourier Transform in Embedded GPUs

Dongwon Yang
Korea University
Seoul, Republic of Korea
yang2919@korea.ac.kr

Jaebeom Jeon
Korea University
Seoul, Republic of Korea
414dragon@korea.ac.kr

Minseong Gil
Korea University
Seoul, Republic of Korea
ms7859@korea.ac.kr

Junsu Kim
Korea University
Seoul, Republic of Korea
j0807s@korea.ac.kr

Seondeok Kim
Korea University
Seoul, Republic of Korea
seondeok0312@korea.ac.kr

Gunjae Koo
Korea University
Seoul, Republic of Korea
gunjaekoo@korea.ac.kr

Myung Kuk Yoon
Ewha Womans University
seoul, Republic of Korea
myungkuk.yoon@ewha.ac.kr

Yunho Oh
Korea University
seoul, Republic of Korea
yunho_oh@korea.ac.kr

Abstract

Fast Fourier Transform (FFT) is critical in applications such as signal processing, communications, and AI. Embedded GPUs are often used to accelerate FFT due to their computational efficiency, but energy efficiency remains a key challenge due to power constraints. Existing solutions, such as the cuFFT library provided by NVIDIA, employ static configurations for the number of thread blocks and threads per block. This static approach often results in ineffective threads that consume power without contributing to performance, particularly if the FFT length or batch size varies. Furthermore, for large FFT lengths, cuFFT internally splits the computation into multiple kernel invocations. This decomposition can lead to L2 cache thrashing, resulting in redundant global memory accesses and degraded efficiency. To address these challenges, this paper proposes SSFFT, a software technique for embedded GPUs. The key idea of SSFFT is to maximize the number of useful threads that contribute to performance while minimizing ineffective threads. SSFFT is implemented based on a novel theoretical model that determines how many thread blocks and threads per block are effective for a given FFT length, batch size, and hardware resource availability. SSFFT statically determines these configurations and adaptively launches either a GPU kernel for regular FFT operations or a newly implemented kernel that integrates multiple FFT steps. By tailoring thread allocation to workload characteristics and minimizing inter-kernel memory interference, SSFFT improves energy efficiency without compromising performance. In our evaluation, SSFFT achieves a 1.29× speedup and a 1.26× improvement in throughput per watt compared to cuFFT.

LCTES '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1921-9/25/06
https://doi.org/10.1145/3735452.3735529

CCS Concepts: • **Computer systems organization → Embedded software**; • **Computing methodologies → Parallel algorithms**.

Keywords: FFT, Dynamic Thread Scaling, Energy Efficiency

ACM Reference Format:
Dongwon Yang, Jaebeom Jeon, Minseong Gil, Junsu Kim, Seondeok Kim, Gunjae Koo, Myung Kuk Yoon, and Yunho Oh. 2025. SSFFT: Energy-Efficient Selective Scaling for Fast Fourier Transform in Embedded GPUs. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735452.3735529

1 Introduction

Fast Fourier Transform (FFT) is a widely used algorithm for efficiently computing the Discrete Fourier Transform (DFT), with applications in signal processing, communications, and artificial intelligence (AI) [3, 5, 7, 9, 12, 21, 25, 33]. FFTs are essential to systems such as real-time signal processing and AI tasks at the edge, where embedded GPUs are often employed for their computational efficiency [1]. However, the power constraints inherent to embedded GPUs make energy-efficient FFT computation a critical requirement. Therefore, enhancing energy efficiency while maintaining performance in FFT operations is critical for embedded GPUs.

We find the following key challenges in GPU-accelerated FFT computations. First, current solutions, such as cuFFT library developed by NVIDIA [27], generate GPU kernels for FFT operations with fixed grid and block dimensions based on FFT lengths and batch sizes. Such a thread allocation policy often results in GPU kernels with too few threads or many ineffective threads, which consume power without contributing to throughput improvement. This misconfiguration of the number of thread blocks (also called grid dimension) and the number of threads per block (also called thread block size or block dimension) incurs energy inefficiency [4, 26].

Second, if the FFT length is large, cuFFT internally decomposes the computation into two separate kernels to process

the input in stages. While this decomposition reduces per-kernel memory usage, it often leads to poor data locality between kernels. In particular, intermediate results written by the first kernel may be evicted from the L2 cache before the second kernel can reuse them, resulting in frequent off-chip memory accesses and L2 cache thrashing. This inefficiency not only increases memory bandwidth pressure but also affects the performance in large-scale FFT workloads.

To address these challenges, we propose Selective Scaling FFT (SSFFT), a novel software technique for energy-efficient FFT computations in embedded GPUs. The key idea of SSFFT is to selectively scale thread blocks and threads to maximize the number of useful threads, which contribute to performance improvement, while minimizing the occurrence of ineffective threads. By adapting thread allocation to workload characteristics, SSFFT improves both performance and energy consumption in embedded GPU environments compared to cuFFT. We design a novel mathematical model of grid and block dimensions, based on not only FFT length and batch size but also critical hardware resource utilization, such as shared memory or L2 cache. Then, we implement SSFFT in software to predetermine the grid and block dimensions and launch a GPU kernel for FFT operations. In addition to adaptive thread allocation, SSFFT dynamically selects one of multiple FFT algorithm implementations depending on the input size and batch configuration. This algorithm-level adaptation allows SSFFT to better exploit data locality and hardware resources under varying workload conditions, further improving performance and energy efficiency.

Our evaluation on the NVIDIA Jetson Orin platform shows that SSFFT outperforms cuFFT. Across various FFT lengths and batch sizes, SSFFT achieves a 1.29× speedup and a 1.26× improvement in throughput per watt compared to cuFFT. In particular, SSFFT achieves up to a 1.8× speedup and a 2× improvement in throughput per watt for short FFT lengths and small batch sizes.

In this paper, we make the following contributions:

- We thoroughly analyze the effect of useful threads and ineffective threads while performing FFT operations in embedded GPUs.
- We present a mathematical model for calculating thread block and grid dimensions based on FFT lengths and batch sizes to improve the energy efficiency of FFT computations in embedded GPUs.
- We design an adaptive FFT algorithm selection to better exploit data locality and hardware resources in software.

The rest of this paper consists of the following sections. Section 2 explains why FFT operations are required to reduce ineffective threads in embedded GPUs. Section 5 explains related work. Section 3 introduces the SSFFT mechanism and implementation. Section 4 explains the experimental results. Section 6 concludes this paper.

2 Why SSFFT?

FFT is a crucial algorithm for efficiently computing the discrete fourier transform, and it has broad applications in communications, signal processing, and AI. The computational requirements of the FFT are heavily influenced by both the length of the input vector (also called FFT length) and the number of vectors being processed [7, 12]. The total number of computations required for an FFT operation increases logarithmically with the length of the input vector, typically scaling as $O(n \log n)$, where n is an FFT length. The overall computational load scales proportionally with the number of vectors (we call it batch size) if multiple vectors are involved, such as in multi-channel or multi-dimensional signal processing tasks. In such cases, the total computation count becomes $O(m \cdot n \log n)$, where m represents batch size.

Unlike traditional edge devices that are equipped with dedicated FFT processors [14, 22, 23, 32, 36], embedded GPUs are viewed as key processors for performing FFT operations for many emerging applications. Embedded GPUs execute thousands of threads in parallel under strict power constraints, making energy efficiency a critical factor in their design and implementation [13, 24, 29]. As such, improving the energy efficiency of FFT operations is essential for advancing these technologies in embedded GPUs.

While running GPU applications, one of the most critical factors is utilizing an appropriate number of threads. Considering that GPUs execute programs by running thousands of threads in parallel, misallocation of threads incurs unnecessary energy consumption without contributing to performance improvements [6, 15, 28, 30, 31]. Although prior work has significantly advanced energy efficiency in running FFT operations on GPUs [1, 2], a fundamental inefficiency persists due to suboptimal thread allocation. To further analyze the effect of the thread misallocation challenge, we define a useful thread as one that actively contributes to performance improvements, whereas an ineffective thread is one that consumes resources but fails to enhance computational throughput. Ineffective threads reduce energy efficiency because they cause contention among threads for critical hardware resources, such as the memory system [15, 30].

A typical approach to improving efficiency in GPU FFT operations is to maximize the use of shared memory, as it provides fast, low-latency access compared to global memory. However, since shared memory shares resources with L1 cache and is limited in size, its effectiveness is constrained, especially when handling large FFT workloads. In GPUs, L2 cache is shared across all SMs. This shared nature allows better data reuse and reduces redundant memory accesses, making it a crucial component in improving energy efficiency for GPU-based FFT operations. Improving L2 cache hit ratio emerges as the most effective approach to mitigating

Table 1. Evaluation configurations

Platform	NVIDIA Jetson Orin AGX
GPU architecture	Ampere architecture
GPU cores	1792 CUDA cores, 56 Tensor cores
CPU	8-core Arm Cortex-A78AE v8.2
Memory	32GB LPDDR5, Bandwidth: 204.8GB/s
Storage	64GB eMMC 5.1
cuFFT	Version 11.0.8 (in Jetpack 6)

Table 2. Grid (gridDim) and Block Dimension (blockDim) Settings for cuFFT and Oracle Based on Vector Length and Batch Size. The Oracle configurations are derived from the results of Figure 1, 2, and 3.

FFT Length	Batch Size	cuFFT		Oracle	
		gridDim	blockDim	gridDim	blockDim
1K	1	1	128	8	32
1K	16	1	128	4	128
1K	128	1	128	1	256
16K	1	1	1024	32	256
16K	16	1	1024	4	256
16K	128	1	1024	1	256
256K	1	64	512	64	256
256K	16	64	512	16	256
256K	128	64	512	4	512

memory stalls and enhancing both performance and energy efficiency.

CUDA Fast Fourier Transform (cuFFT) is a library developed by NVIDIA for performing FFT computations on NVIDIA GPUs [27]. cuFFT offers APIs that allow developers to perform complex-to-complex, real-to-complex, and complex-to-real transforms, making it flexible for various use cases. While cuFFT simplifies the use of GPUs for FFT operations, it predetermines the grid and block dimensions based on the FFT length and the batch size. This static setup may incur inefficiencies, especially in energy consumption, due to the creation of ineffective threads.

We investigate the performance impact of thread configurations on FFT operations using the NVIDIA Jetson Orin system for our experiments [8]. The detailed system configurations are described in Table 1. We manually implement an FFT algorithm with CUDA and optimized it to achieve the same performance as cuFFT 11.0.8 [27]. After that, we configure FFT operations with three FFT lengths (1K, 16K, and 256K) and three batch sizes (1, 16, and 128). We measure throughput per watt of those FFT operations by varying grid and block dimensions. We measure the power consumption of the FFT operations with tegrastats to estimate the energy efficiency [10].

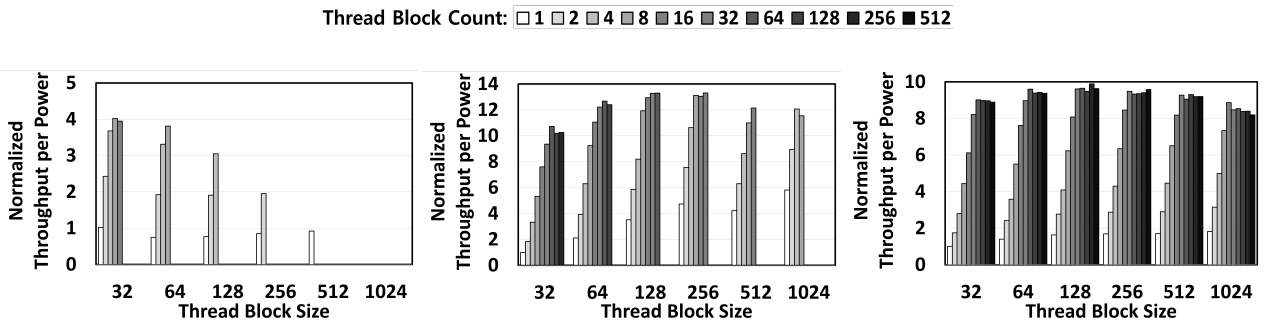
Figures 1, 2, and 3 show the experimental results. We analyze the relationship between thread block size and energy efficiency. With an FFT length of 1k and a batch size of 1, only 32 threads in a single thread block are useful. In this case, it is more efficient to distribute the work across multiple thread blocks, each with 32 useful threads. Similarly, with an FFT length of 16k and a batch size of 128, 256 threads in a thread block are useful. Energy efficiency gains are significant for an FFT length of 16k with smaller thread block sizes, but beyond 256 threads per block, the additional threads become ineffective, resulting in diminishing returns. This behavior occurs because the hardware resources in an SM, such as shared memory, become saturated, causing the remaining threads to be ineffective.

With the experimental results in Figures 1, 2, and 3, we heuristically determine the optimal thread configurations for each scenario, called the Oracle configuration. Based on empirical testing, the Oracle configuration selects the best-performing and minimum thread configuration for every case. Table 2 shows the thread and thread block count in cuFFT and the Oracle configuration for various FFT configurations. In our analysis, cuFFT and the Oracle configuration differ in their grid dimension and block dimension configurations. For example, with an FFT length of 1k and a batch size of 1, cuFFT allocates a single thread block with 128 threads. However, we find that allocating eight thread blocks with 32 threads each is more energy efficient (as shown in Figure 1c). Also, with an FFT length of 256k and a batch size of 128, only 4 out of 64 thread blocks (with a thread block size of 512) are useful. Our analysis shows that a more advanced thread allocation technique than cuFFT is necessary to achieve the optimal balance between speedup and energy efficiency in embedded GPUs.

Starting from cuFFT 11.1, L2-cache awareness has been introduced, utilizing L2 cache for GPUs [27]. However, this optimization is limited to specific single-GPU 3D C2C FFT cases. As a result, cuFFT does not fully leverage L2 cache benefits across a wider range of FFT workloads, such as 1D and 2D FFTs or batched processing. Consequently, inefficiencies in memory access and thread allocation persist, leaving significant room for further improvements in performance and energy efficiency.

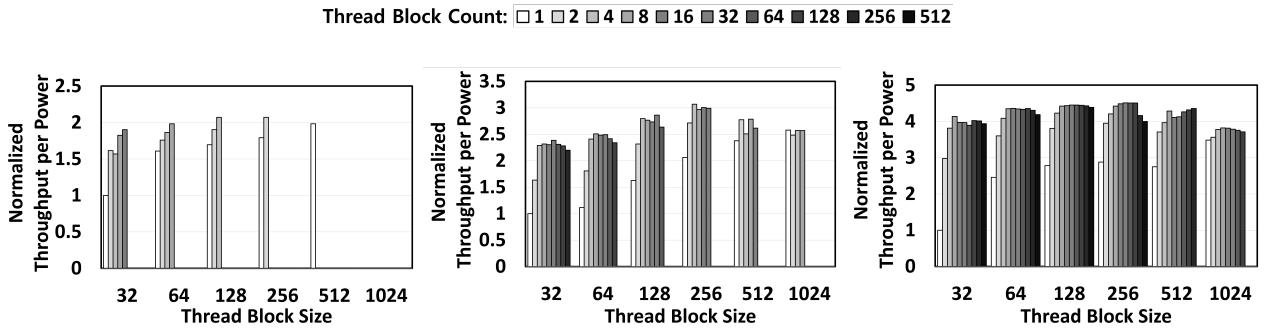
Despite these updates, we find that the L2 cache hit ratio during cuFFT execution is close to zero, indicating that L2 cache utilization remains minimal. Detailed experimental results supporting this observation are presented in Section 4.2. This observation suggests that cuFFT does not effectively utilize L2 cache in most FFT scenarios, further highlighting the need for improved cache-aware execution strategies to enhance performance and energy efficiency.

One possible factor contributing to this inefficiency is the way cuFFT handles large FFT workloads. To process large FFTs efficiently, cuFFT decomposes the computation into multiple stages. For instance, 2D or 3D FFTs, which are stored in row-major order, are typically split into separate row-wise and column-wise FFT computations. Even in the case of large 1D FFTs, cuFFT internally processes the data in block-wise fashion, often requiring multiple CUDA kernels



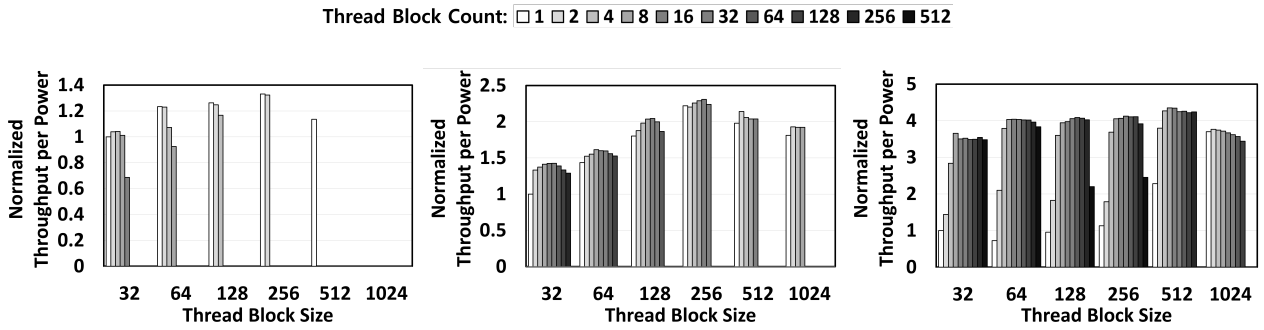
(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 1. Throughput per power comparison varying thread block counts and thread block sizes, Batch size is 1. The areas without bars indicate regions where threads cannot be assigned because the corresponding GPU kernel does not use them.



(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 2. Throughput per power comparison varying thread block counts and thread block sizes, Batch size is 16. The areas without bars indicate regions where threads cannot be assigned because the corresponding GPU kernel does not use them.



(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 3. Throughput per power comparison varying thread block counts and thread block sizes, Batch size is 128. The areas without bars indicate regions where threads cannot be assigned because the corresponding GPU kernel does not use them.

for execution. For example, in 2D FFTs, row FFTs and column FFTs are executed as separate kernel invocations. This decomposition can impact L2 cache utilization, as intermediate data may be evicted from the cache between kernel executions, leading to redundant memory accesses. These structural inefficiencies reinforce the need for a more effective cache-aware execution strategy that better retains data in L2 cache, reducing memory traffic and improving both performance and energy efficiency.

Despite significant advancements in optimizing FFT execution on GPUs, such as those seen in tcFFT [20], TurboFFT [34], and energy-efficient methods for edge computing [14, 22, 32], a fundamental inefficiency remains in the

lack of optimal thread allocation, which directly impacts energy efficiency. Many GPU-accelerated FFT implementations still utilize suboptimal thread allocation strategies, where the thread block sizes and grid dimensions are not dynamically adjusted to match the FFT problem size or the underlying GPU architecture. This misallocation results in underutilization of GPU threads, leaving many threads consuming energy without contributing to performance improvements. Even in high-performance and fault-tolerant implementations, energy consumption can remain unnecessarily high if thread resources are not fully exploited. Thus, without addressing the thread allocation issue, it becomes difficult to achieve optimal energy efficiency, particularly in large-scale or real-time FFT applications on GPUs. A more dynamic and workload-specific approach to thread allocation is required to fully leverage the capabilities of modern GPUs.

3 SSFFT

To address the challenges explained in Section 2, we propose a new software technique called SSFFT. The goal of SSFFT is to properly determine and allocate useful threads on GPUs while minimizing the occurrence of ineffective threads, thereby avoiding resource waste. To achieve this objective, SSFFT consists of two ideas. First, SSFFT determines the grid and block dimensions based on a new theoretical modeling. Figure 4 depicts the key idea of SSFFT. SSFFT addresses the risk of performance degradation that arises if too many FFT operations run simultaneously or if a single FFT operation involves an excessive number of thread blocks or threads in a thread block. Second, depending on FFT length, SSFFT adaptively selects either of three algorithms: regular FFT, vector FFT, and merged FFT. While regular FFT and vector FFT are based on existing algorithms, we newly implement a merged FFT algorithm by simply extending a regular FFT algorithm. With these ideas, SSFFT utilizes GPU hardware resources to minimize ineffective thread creation, while handling varying FFT sizes and batch workloads.

3.1 SSFFT Thread Allocation

We implement the key technique of SSFFT by mathematically modeling the performance and thread block allocation. To improve data locality and maintain a high L2 cache hit ratio, SSFFT first determines the amount of data processed in FFT with the following equation.

$$A = 2 << \lceil \log_2(\text{L2\_cache\_size}) \rceil - 2 \tag{1}$$

where A denotes the amount of data (in bytes) processed in a single iteration. The exponent subtracts 2 from the logarithm of the L2 cache size to compute one-fourth of the cache capacity. This equation ensures that each iteration operates on a dataset that likely fits within the cache, reducing cache misses during FFT computation.

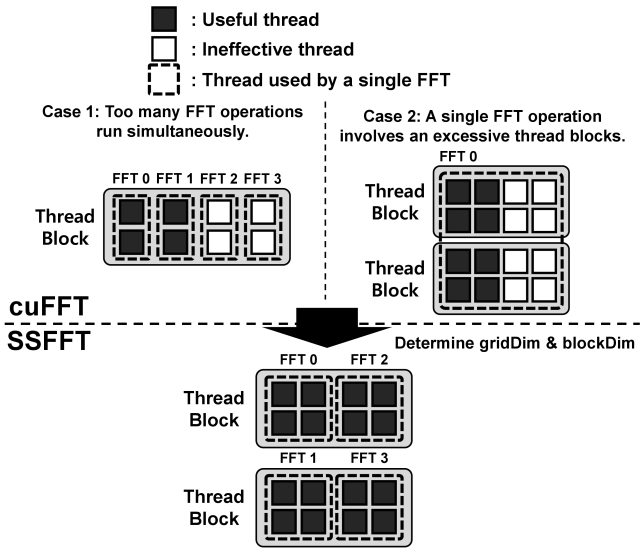


Figure 4. Workload-aware thread allocation of SSFFT.

Given an FFT of length L applied to a batch of N input vectors, the total input size is $L \times N \times IN$ bytes, assuming each input is IN bytes. For example, if an input is a double-precision complex number, IN is 8 bytes. With those numbers, the number of required iterations K is calculated as:

$$K = \frac{L \times N \times IN}{A} \tag{2}$$

This equation determines the number of iterations needed to process the entire input dataset in chunks of size A. By limiting per-iteration memory access to a fraction of the L2 cache, SSFFT achieves better cache utilization and overall performance. After calculating Equation 2, SSFFT calculates two types of maximum grid dimension.

$$M_s = \frac{(\text{shared memory per SM})}{(\text{shared memory per TB})} \times (\#\text{SMs}) \tag{3}$$

$$M_f = \frac{(\text{FFT length})}{64} \tag{4}$$

Equation 3 calculates a maximum grid dimension that can be allocated on a GPU based on the available shared memory size (M_s). Equation 4 calculates a maximum grid dimension based on the FFT length (M_f). As each warp consists of 32 threads and the butterfly operation in an FFT operation involves two operands, the FFT length is divided by 64 to determine the appropriate number of thread blocks.

With M_s, M_f, and FFT batch size N, SSFFT determines grid and block dimensions as follows. If a single FFT is executed ($N = 1$), SSFFT considers the following two scenarios. If the total number of threads required exceeds the FFT length ($2 \times M_s > M_f$), the number of grid dimension is limited by M_f and the block dimension is fixed at 32. Otherwise, if the number of threads is constrained by the available shared memory,

Algorithm 1 Pseudocode of FFT functions for SSFFT

Input: FFT length L, batch size N,
#Iterations K, Input data I
Parameters: k-th sub unit of input I_k

```
1:  function REGULAR_FFT(float I, int L, ...)
2:      DoFFT(I, √L, ...)
3:      // √L-point FFT operation for input I
4:  end function
5:
6:  function VECTOR_FFT(float I, int L, ...)
7:      DoFFT(I, L, ...)
8:      // L-point FFT operation for input I
9:  end function
10:
11: function MERGED_FFT(float I, int L, int N, int K, ...)
12:     for k = 0 → K do
13:         DoFFT(I_k, √L, ...)
14:         // √L-point FFT operation for sub-unit I
15:         // Transpose Matrix
16:         DoFFT(I_k, √L, ...)
17:         // √L-point FFT operation for sub-unit I
18:     end for
19: end function
```

the grid dimension and block dimensions are determined as follows.

$$D_{\text{grid}} = 2 \times M_s \tag{5}$$

$$D_{\text{block}} = \frac{\text{Max. \# of concurrent threads on a GPU}}{M_s}. \tag{6}$$

For batched FFT execution ($N > 1$), the grid and block dimensions are dynamically adjusted to efficiently distribute workloads while considering the batch size. The initial grid size is determined as follows.

$$D_{\text{grid}} = 2 \times M_s \gg \lfloor \log_2(N) \rfloor. \tag{7}$$

The final grid size is set as:

$$D_{\text{grid}} = \min(M_f, D_{\text{grid}}). \tag{8}$$

The block size is then computed as:

$$D_{\text{block}} = \frac{\text{Max. \# of concurrent threads on a GPU}}{D_{\text{grid}}}. \tag{9}$$

3.2 SSFFT Algorithms

SSFFT employs three distinct FFT algorithms to adapt to varying input sizes and batch configurations. SSFFT dynamically selects one of the three FFT algorithms: Regular FFT, vector FFT, and merged FFT. Algorithm 1 describes the FFT algorithms. Note that the regular FFT and vector FFT implementations are identical to the implementations in cuFFT [27] and the merged FFT is a newly added implementation in SSFFT.

Regular FFT, illustrated in `regular_FFT` (lines 1–4), is suitable for short FFT lengths. It decomposes the input into $\sqrt{\mathcal{L}}$-point sub-FFTs, allowing more efficient use of shared memory and enabling finer control over memory access patterns. Regular FFT is selected if the total input size exceeds a cache-friendly threshold, specifically if it cannot fit into one-fourth of the L2 cache.

Table 3. Grid and Block Dimension Settings for SSFFT and Oracle Based on Vector Length and Batch Size

FFT Length	Batch Size	SSFFT		Oracle	
		gridDim	blockDim	gridDim	blockDim
1K	1	16	32	8	32
1K	16	16	32	4	128
1K	128	4	128	1	256
16K	1	32	256	32	256
16K	16	4	256	4	256
16K	128	1	256	1	256
256K	1	64	256	64	256
256K	16	16	256	16	256
256K	128	16	256	4	256

Vector FFT is used if the FFT length is small and the batch size is moderate or large. As shown in `vector_FFT` (lines 6–9), the algorithm performs a standard $\mathcal{L}$-point FFT directly on the input without decomposition. This approach minimizes overhead and achieves high performance if the FFT length is relatively short and the overall workload size is sufficiently large.

Merged FFT, described in `merged_FFT` (lines 11–19), targets large FFT lengths and small batch sizes. It partitions the input into K sub-units and performs two successive $\sqrt{\mathcal{L}}$-point FFTs on each sub-unit, separated by a matrix transpose operation. This transpose step improves memory coalescing and enables better data reuse across row- and column-wise FFTs. The merged FFT reduces kernel launch overhead and increases L2 cache hit rates by preserving temporal locality.

3.3 SSFFT Implementation

Algorithm 2 presents the full implementation flow of SSFFT. SSFFT begins by computing the cache-aware memory threshold A and the number of iterations K based on the FFT length L, batch size N, and the L2 cache size. The value A corresponds to one-fourth of the L2 cache capacity, and determines whether the input fits within a single iteration without causing significant cache pressure.

SSFFT first evaluates whether the total input size ($L \times IN$ bytes) exceeds A. If this condition holds, SSFFT invokes the `regular_FFT` routine twice (once for the column-wise FFT and once for the row-wise FFT) separated by a matrix transpose. Prior to kernel invocation, the `SSFFT_scaling` function is called to determine the optimal `gridDim` and `blockDim` based on shared memory capacity and the FFT problem size.

If the input size fits within the cache threshold, SSFFT further checks whether the FFT length is small (i.e., $L \leq$ 16K) and the total workload size is sufficiently large (i.e., $L \times N/8 \geq$ 4KB). If both conditions are satisfied, SSFFT

Algorithm 2 SSFFT function pseudocode

Input: FFT length: L, batch size: N
Parameters: Maximum #Thread Blocks (TBs): M_s,
Maximum #TBs based on fft length: M_f,
shared memory size per TB: S
Limit of input size per iteration: A bytes
Size of a single input: IN bytes
Output: gridDim: D_{grid}, blockDim: D_{block}

function SSFFT_SCALING(int L, int N)
  // Calculate M_s, M_f
  M_s = (shared memory per SM)$/S \times \#SMs$
  $M_f = L/64$
  if N == 1 **then**
    // Single operation
    if $2 \times M_s > M_f$ **then** // #TB limited by fft length
      $D_{grid} = M_f$
      $D_{block} = 32$
    else// #TB limited by shared memory size
      $D_{grid} = 2 \times M_s$
      D_{block} =(Total maximum threads)$/M_s$
    end if
  else
    // Batch operation
    $D_{grid} = 2 \times M_s >> floor(log_2(N))$
    $D_{grid} = min(M_f, D_{grid})$
    D_{block} =(Total maximum threads)$/D_{grid}$
  end if
  return D_{grid}, D_{block}
end function
function SSFFT(int L, int N, ...)
  $A = 2 << (ceiling(log_2(\text{L2 cache size})) - 2)$
  $K = L \times N \times IN/S_i$
  // Calculate A, K
  if $L \times 8 > A$ **then**
    D_{grid}, D_{block} = SSFFT_scaling(L, N)
    // Calculate gridDim and blockDim
    regular_FFT <<<D_{grid}, D_{block}, S>>>(...)
    // 1-D column dimension FFT
    // Transpose Matrix
    regular_FFT <<<D_{grid}, D_{block}, S>>>(...)
    // 1-D row dimension FFT
  else
    if L <= $16k$ and $L \times N/8$ >= $4k$ **then**
      D_{grid}, D_{block} = SSFFT_scaling(L, N)
      // Calculate gridDim and blockDim
      vector_FFT <<<D_{grid}, D_{block}, S>>>(...)
      // 1-D vector FFT
    else
      D_{grid}, D_{block} = SSFFT_scaling(L, N/K)
      // Calculate gridDim and blockDim
      merged_FFT <<<D_{grid}, D_{block}, S>>>(...)
      // merged FFT
    end if
  end if
end function

executes the `vector_FFT` kernel, which performs a 1-D FFT over each input vector without decomposition, using the dimensions computed by `SSFFT_scaling`.

In all other cases (typically large FFTs with small batch sizes), SSFFT opts for the `merged_FFT` strategy. This kernel processes smaller sub-units of the input sequentially and applies two $\sqrt{L}$-point FFTs for each unit, improving L2 cache locality and reducing redundant global memory accesses. For this case, `SSFFT_scaling` is invoked with a reduced batch size (N/K) to adapt the thread configuration to the smaller sub-problems. By integrating this multi-branch execution model with dynamic thread scaling, SSFFT ensures that each kernel launch is tailored to the workload and hardware constraints, minimizing the creation of ineffective threads while maximizing GPU resource efficiency.

4 Evaluation

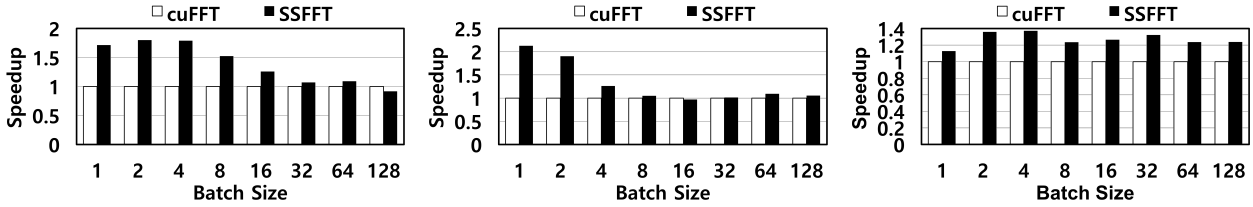
4.1 Evaluation Environments

As we explained in Section 2, we use an NVIDIA Jetson Orin platform whose specification is described in Table 1 for evaluating FFT performance. We set the FFT lengths to 1K, 16K, and 256K, with the batch sizes ranging from 1 to 128. We measure and compare the total execution time to complete all the FFT operations in a batch. Also, we evaluate throughput per watt of SSFFT. We compare SSFFT with cuFFT. For a fair comparison, as we mentioned in Section 2, we implement an FFT algorithm with CUDA and optimized it to achieve the same performance as cuFFT. After that, we implement SSFFT to scale gridDim and blockDim as we modeled in Section 3.

4.2 Experimental Results

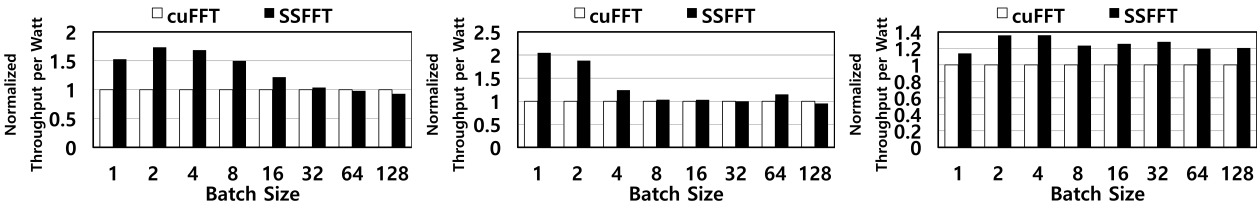
Table 3 shows the thread block and thread count set by SSFFT and the Oracle configuration. These results show that SSFFT closely aligns with the Oracle configuration regarding gridDim and blockDim settings across various FFT lengths and batch sizes. SSFFT determines these parameters to maximize performance and energy efficiency, approaching the optimal Oracle configuration. The ability of SSFFT to achieve such precise thread allocation is rooted in the mathematical modeling that calculates the optimal number of thread blocks and threads based on FFT length, batch size, and shared memory constraints. This dynamic scaling allows SSFFT to operate efficiently under different workloads, minimizing ineffective threads similar to the Oracle. cuFFT uses fixed grid and block dimension configurations based on the FFT length and batch size, which are not adjusted to fully utilize GPU resources. SSFFT, in contrast, dynamically adjusts the gridDim and blockDim values based on the FFT length and batch size, leading to efficient GPU resource utilization.

For an FFT length of 16k, SSFFT determines the optimal gridDim and blockDim, all are equivalent to the Oracle configurations. For an FFT length of 256k and a batch size of 1, SSFFT sets gridDim to 64 and blockDim to 256, optimizing GPU resource allocation. As the batch size increases to



(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 5. Speedup comparison between cuFFT and SSFFT across varying batch sizes and FFT lengths. Each subfigure reports the normalized execution time of SSFFT relative to cuFFT. SSFFT consistently outperforms cuFFT in scenarios with small batch sizes, demonstrating the benefit of selective thread scaling and adaptive algorithm selection in resource-constrained environments. With large vectors, SSFFT achieves performance improvement in all batch sizes.



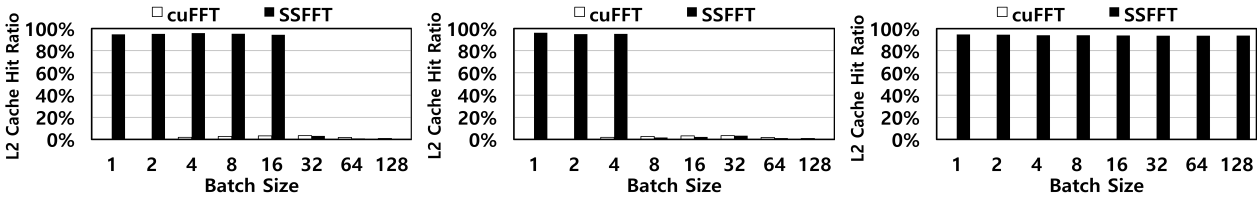
(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 6. Throughput per power comparison between cuFFT and SSFFT. All results are normalized to the baseline, cuFFT. SSFFT consistently outperforms cuFFT in both metrics, with the most significant improvements observed at small batch sizes (e.g., 1–4), where cuFFT suffers from thread underutilization and memory inefficiency. For large FFT lengths such as 256k, SSFFT maintains consistent energy efficiency improvements over cuFFT by reducing L2 cache thrashing and maintaining well-balanced thread allocation across batch sizes.

128, SSFFT further scales the gridDim to 16 and blockDim to 256, Such results are also the same as those of the Oracle configurations. With these results, we observe that SSFFT effectively maximizes the number of useful threads while avoiding the creation of ineffective threads. This dynamic adjustment enables SSFFT to outperform cuFFT in terms of both performance and energy efficiency for larger workloads. For FFT length of 1K and batch sizes 1 and 16, SSFFT sets a higher gridDim of 16 than the Oracle. Also, for batch sizes 128, SSFFT sets a higher gridDim of 4 than the Oracle. Even with those thread allocations, SSFFT still outperforms cuFFT according to the experimental results, which we will explain later.

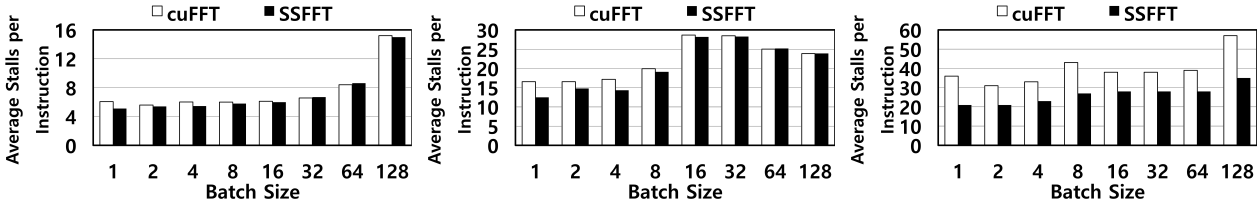
Figure 5 shows the performance of SSFFT compared to cuFFT. For all the configurations, SSFFT achieves a 1.29× (geometric mean) better performance than cuFFT. For small batch sizes (under 16), SSFFT consistently delivers better performance compared to cuFFT. With FFT lengths of 1K and 16K, SSFFT achieves a speedup of up to 2.2× for batch size 1, and for a 256K FFT length, SSFFT achieves up to 1.39× speedup on average, with all batch sizes. This performance advantage stems from the ability of SSFFT to adjust thread blocks based on FFT length and hardware resource utilization, ensuring optimal utilization of useful threads. In contrast, cuFFT with the fixed thread allocation results in underutilization of GPU resources for smaller batch sizes. Small batch sizes are often used in audio or speech signal processing, and in such cases, SSFFT can achieve significant performance improvements.

As the batch size increases, the performance gap between SSFFT and cuFFT narrows. For larger batch sizes, such as 64 and 128, SSFFT performs with marginal performance improvements compared to cuFFT. For an FFT length of 16K and a batch size of 128, SSFFT matches cuFFT in execution time, showing a 1.07× improvement in speedup. Such results are attributed to the fact that both implementations can efficiently handle larger batch operations, with the proposed dynamic scaling providing a slight edge. In 1k FFT vectors and batch sizes from 16 to 64, SSFFT achieves 1.11× speedup compared to cuFFT. SSFFT focuses on optimizing thread block size based on shared memory constraints, allowing it to maintain efficiency even as the batch size grows. With 1k FFT vectors and a batch size of 128, SSFFT shows a 1.05×



(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 7. L2 cache hit ratio comparison between cuFFT and SSFFT across varying FFT lengths and batch sizes. For all FFT lengths of 1k and 16k respectively, SSFFT consistently achieves over 80% L2 cache hit ratio in batch sizes, while cuFFT suffers from poor locality, with hit ratios falling below 10% for most configurations. These results show that SSFFT significantly improves memory locality by minimizing inter-kernel cache interference and tailoring kernel execution to the cache hierarchy.



(a) FFT length = 1k. **(b)** FFT length = 16k. **(c)** FFT length = 256k.

Figure 8. Average stalls per instruction comparison between cuFFT and SSFFT across varying FFT lengths and batch sizes. For FFT lengths of 1k and 16k, SSFFT maintains similar or slightly lower stall counts compared to cuFFT. However, the impact on performance is minimal in these cases, as the execution time per GPU kernel is short. For an FFT length of 256k, SSFFT significantly outperforms cuFFT as the batch size increases, reducing stalls by 34% on average.

speedup compared to cuFFT. Such results show that SSFFT does not sacrifice performance in any case.

SSFFT also demonstrates significant improvements in energy efficiency for small batch sizes. Figure 6 shows the experimental results. For all the configurations, SSFFT achieves a 1.26× (geometric mean) better throughput per watt than cuFFT. With a batch size of 1 and FFT lengths of 1K and 16K, SSFFT shows up to 2.1 × better energy efficiency than cuFFT. This improvement is primarily due to the ability of SSFFT to allocate just enough thread blocks to fully utilize GPU resources without wasting energy on ineffective threads. SSFFT maintains energy efficiency advantages over cuFFT as the batch size increases, though the margin narrows. SSFFT can minimize unnecessary power usage while maintaining performance by dynamically scaling thread blocks based on FFT length and batch size. For the largest batch size (128), SSFFT and cuFFT exhibit similar energy efficiency, with SSFFT maintaining a slight advantage. At this scale, both implementations intensively utilize GPU resources, and the power savings offered by the proposed dynamic thread management technique become less pronounced. Nonetheless, SSFFT continues to show a small edge in power consumption, especially in configurations where it minimizes the creation of ineffective threads. As mentioned in Section 5, too many ineffective threads in a GPU kernel often result in inefficient utilization of critical hardware resources, such as memory systems. To further illustrate this inefficiency, we analyze the L2 cache hit ratio of SSFFT compared to cuFFT. Figure 7 presents the experimental results.

SSFFT effectively reduces ineffective threads, thereby improving memory locality and minimizing unnecessary memory accesses. This idea leads to a significantly higher L2 cache hit ratio compared to cuFFT across different FFT sizes and batch configurations. For instance, in the case of 1K and 16K FFT lengths, SSFFT achieves near 100% L2 cache hit ratio for small batch sizes (under 32), whereas cuFFT exhibits an L2 cache hit ratio close to zero. cuFFT does not effectively utilize the L2 cache, leading to frequent off-chip memory accesses and increased memory bandwidth contention.

For an FFT length of 256K, SSFFT maintains a high L2 cache hit ratio across all batch sizes (94% on average), whereas cuFFT continues to exhibit a 0% L2 cache hit ratio. cuFFT fails to effectively utilize the L2 cache even for large FFT sizes, leading to frequent off-chip memory accesses. This inefficiency primarily stems from the fact that cuFFT executes two separate kernels for computing the FFT along different dimensions. During this process, intermediate data

frequently evicts useful cache-resident data, causing cache thrashing and excessive memory bandwidth consumption.

In contrast, SSFFT applies the merged-FFT approach, as outlined in Algorithm 1, which minimizes the number of kernel launches and improves cache locality. By carefully preserving the working set within the L2 cache across computation phases, SSFFT enables data reuse across column-wise and row-wise FFT computations without unnecessary cache eviction. As a result, memory accesses are more localized, significantly improving the L2 cache hit ratio and reducing off-chip memory traffic. These findings reinforce the need for improved L2 cache-aware execution in cuFFT to mitigate memory inefficiencies and optimize GPU resource utilization, particularly for both small and large FFT sizes in batched processing.

The cache-friendly behavior of SSFFT contributes to improving the performance of FFT operations, especially with large FFT vectors. To show it, we measure the average stalls (cycles) per instruction with 1k, 16k, and 256k vectors using NVIDIA Nsight Compute. Figure 8 shows the experimental results. With 1k vectors (Figure 8a), cuFFT and SSFFT spend 7.3 and 7.5 cycles per instruction, respectively. And, with 16k vectors (Figure 8b), cuFFT and SSFFT show 20.8 and 22.0 cycles per instruction. cuFFT and SSFFT spend similar cycles to complete an instruction. Even though SSFFT achieves a much higher L2 cache hit ratio than cuFFT, a single kernel exhibits a short execution time with any implementation with 1k and 16k vectors. As such, the impact of cache hits does not vary the performance significantly. Instead, SSFFT achieves the performance improvement by allocating threads across SMs in a better way than cuFFT. In case of cuFFT, it often makes each thread iterate a loop, resulting overall execution time depending on such sequential behavior. Unlike cuFFT, SSFFT effectively spreads tasks by allocating more thread than cuFFT, so the average number of instructions per thread reduces. Such behavior of SSFFT results in performance improvement over cuFFT.

With 256k vectors (Figure 8c), SSFFT reduces the average stalls per instruction significantly compared to cuFFT. SSFFT and cuFFT show 26.4 and 39.4 cycles per instructions, so the proposed technique achieves a 34% reduction. As SSFFT achieves more L2 cache hits than cuFFT, it could reduce the stalls caused by memory operations. With this advantage, SSFFT could achieve the performance with large FFT vectors in any batch size. Based on this behavior analysis, we find that SSFFT effectively utilizes GPU hardware resources, thus achieving a better energy efficiency.

5 Related Work

Prior work has proposed various GPU-accelerated FFT operation techniques [20, 34], and energy-efficient methods for edge computing [14, 22, 32]. TurboFFT introduces a high-performance FFT implementation on GPUs, designed with a focus on fault tolerance [34]. TurboFFT employs a two-sided checksum scheme that detects and corrects silent data corruptions during computation, thereby enhancing reliability. TurboFFT focuses on the advancement in fault tolerance for FFT computations, so it is orthogonal to our work. tcFFT presents a half-precision FFT library specifically optimized for NVIDIA Tensor Cores [20]. tcFFT supports a wide range of 1D and 2D FFT sizes, leveraging mixed-precision capabilities of Tensor Cores. Adámek et al. have explored the energy efficiency improvements for FFT computations on GPUs [2]. They have studied the impact of frequency scaling on the cuFFT.

Despite advancements in improving energy efficiency while running FFT operations on GPUs achieved by the prior work, a fundamental inefficiency remains in the lack of optimal thread allocation, which directly impacts energy efficiency. So, we focus on a more dynamic and workload-specific approach to thread allocation. The aforementioned prior work is not directly comparable to our approach, as it optimizes FFT computation using Tensor Cores or frequency scaling.

Prior work has shown that maximum thread-level parallelism (TLP) on GPUs may incur performance degradation due to cache contention or increased stalls in memory systems [11, 15, 18, 30, 35]. To mitigate such a problem, various hardware-level warp throttling techniques [16, 17, 19] have been proposed. Despite the effectiveness of these approaches, there has been a lack of software-based solutions that dynamically adjust thread allocation to avoid performance bottlenecks, particularly in energy-constrained environments such as embedded systems. This gap calls for software techniques that optimize thread management for FFT operations on GPUs.

6 Conclusion

In this paper, we propose SSFFT, a novel software technique that improves the energy efficiency of FFT operations on embedded GPUs. Unlike the static thread allocation used by cuFFT, which leads to underutilized GPU resources and energy inefficiencies, SSFFT selectively scales thread blocks to maximize performance while minimizing the creation of ineffective threads. Also, depending on workload characteristics, SSFFT dynamically selects and runs one of predetermined FFT algorithms. Our evaluation demonstrates the effectiveness of SSFFT in balancing performance and energy efficiency, making it well-suited for power-constrained environments such as edge computing.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) funded by Korea government (MSIT) (NRF-2022R1C1C1011021, NRF-2021R1C1C1012172, and RS-2025-00553645). Yunho Oh is the corresponding author.

References

[1] Tahmid Abtahi, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. 2018. Accelerating convolutional neural network with FFT on embedded hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 9 (2018), 1737–1749. https://doi.org/10.1109/TVLSI.2018.2825145

[2] Karel Adámek, Jan Novotný, Jeyarajan Thiyagalingam, and Wesley Armour. 2021. Efficiency Near the Edge: Increasing the Energy Efficiency of FFTs on GPUs for Real-Time Edge Computing. *IEEE Access* 9 (2021), 18167–18182. https://doi.org/10.1109/ACCESS.2021.3053409

[3] Nasir Ahmed and Kamisetty Ramamohan Rao. 2012. *Orthogonal transforms for digital signal processing*. Springer Science & Business Media. https://doi.org/10.1109/ICASSP.1976.1170121

[4] Tyler Allen and Rong Ge. 2016. Characterizing power and performance of gpu memory access. In *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*. IEEE, 46–53. https://doi.org/10.1109/E2SC.2016.012

[5] Younghoon Byun, Minho Ha, Jeonghun Kim, Sunggu Lee, and Youngjoo Lee. 2019. Low-complexity dynamic channel scaling of noise-resilient CNN for intelligent edge devices. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 114–119. https://doi.org/10.23919/DATE.2019.8715280

[6] Tiffany Connors, Apan Qasem, and Qing Yi. 2015. Modeling the impact of thread configuration on power and performance of GPUs. *Machine Learning: Theory and Applications* (2015), 28.

[7] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301. https://doi.org/10.2307/2003354

[8] Nvidia Corporation. 2022. *NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications*. https://resources.nvidia.com/en-us-jetson-agx-orin-pathfactory-content

[9] Rene de Jesus Romero-Troncoso. 2016. Multirate signal processing to improve FFT-based analysis for detecting faults in induction motors. *IEEE Transactions on industrial informatics* 13, 3 (2016), 1291–1300. https://doi.org/10.1109/TII.2016.2603968

[10] Nvidia Developer. 2023. *Power Optimization with NVIDIA Jetson*. https://developer.nvidia.com/blog/power-optimization-with-nvidia-jetson/

[11] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2019. Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 492–505. https://doi.org/10.1109/HPCA.2019.00061

[12] P. Duhamel and M. Vetterli. 1990. Fast fourier transforms: A tutorial review and a state of the art. *Signal Processing* 19, 4 (1990), 259–299. https://doi.org/10.1016/0165-1684(90)90158-U

[13] Tao Gong, Tiantian Fan, Jizheng Guo, and Zixing Cai. 2017. GPU-based parallel optimization of immune convolutional neural network and embedded system. *Engineering Applications of Artificial Intelligence* 62 (2017), 384–395. https://doi.org/10.1016/j.engappai.2016.08.019

[14] Shijie Jiang, Yi Zou, Hao Wang, and Wanwan Li. 2023. An FFT Accelerator Using Deeply-coupled RISC-V Instruction Set Extension for Arbitrary Number of Points. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 165–171. https://doi.org/10.1109/ASAP57973.2023.00036

[15] Onur Kayıran, Adwait Jog, Mahmut T Kandemir, and Chita R Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 157–166. https://doi.org/10.1109/PACT.2013.6618813

[16] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. 2015. Efficient utilization of gpgpu cache hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*. 36–47. https://doi.org/10.1145/2716282.2716291

[17] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. In *Proceedings of the 44th annual international symposium on computer architecture*. 307–319. https://doi.org/10.1145/3079856.3080239

[18] Hsien-Kai Kuo, Bo-Cheng Charles Lai, and Jing-Yang Jou. 2014. Reducing contention in shared last-level cache for throughput processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 20, 1 (2014), 1–28. https://doi.org/10.1145/2676550

[19] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-consolidation: A novel execution model for gpus. In *Proceedings of the 2018 International Conference on Supercomputing*. 53–64. https://doi.org/10.1145/3205289.3205294

[20] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. https://doi.org/10.1109/Cluster48925.2021.00035

[21] Sheng Lin, Ning Liu, Mahdi Nazemi, Hongjia Li, Caiwen Ding, Yanzhi Wang, and Massoud Pedram. 2018. FFT-based deep learning deployment in embedded systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1045–1050. https://doi.org/10.23919/DATE.2018.8342166

[22] Yiyang Lin, Yi Zou, and Yanfeng Yang. 2024. CSIFA: A Configurable SRAM-based In-Memory FFT Accelerator. In *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 161–162. https://doi.org/10.1109/ASAP61560.2024.00040

[23] Shaohan Liu and Dake Liu. 2018. A high-flexible low-latency memory-based FFT processor for 4G, WLAN, and future 5G. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 3 (2018), 511–523. https://doi.org/10.1109/TVLSI.2018.2879675

[24] Arian Maghazeh, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. 2013. General purpose computing on low-power embedded GPUs: Has it come of age?. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 1–10. https://doi.org/10.1109/SAMOS.2013.6621099

[25] Jason L Mitchell, Marwan Y Ansari, and Evan Hart. 2004. Advanced image processing with directx® 9 pixel shaders. *ShaderX2* (2004), 439–464.

[26] Sparsh Mittal and Jeffrey S Vetter. 2014. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR)* 47, 2 (2014), 1–23. https://doi.org/10.1145/2636342

[27] NVIDIA. 2025. *cuFFT API Reference*. https://docs.nvidia.com/cuda/cufft/index.html

[28] Yunho Oh, Gunjae Koo, Murali Annavaram, and Won Woo Ro. 2019. Linebacker: preserving victim cache lines in idle register files of GPUs. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/3307650.3322222

[29] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1553–1568. https://doi.org/10.1109/TPDS.2021.3119402

[30] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 72–83. https://doi.org/10.1109/MICRO.2012.16

[31] Richard Schoonhoven, Bram Veenboer, Ben Van Werkhoven, and K Joost Batenburg. 2022. Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 48–59. https://doi.org/10.1109/PMBS56514.2022.00010

[32] Peter Schulz and Grigore Sleahtitchi. 2023. FPGA-based Accelerator for FFT-Processing in Edge Computing. In *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 1. IEEE, 590–595. https://doi.org/10.1109/IDAACS58523.2023.10348654

[33] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014). https://doi.org/10.48550/arXiv.1412.7580

[34] Shixun Wu, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Huangliang Dai, Sheng Di, Zizhong Chen, and Franck Cappello. 2024. TurboFFT: A High-Performance Fast Fourier Transform with Fault Tolerance on GPU. *arXiv preprint arXiv:2405.02520* (2024). https://doi.org/10.48550/arXiv.2405.02520

[35] Jun Zhang, Yanxiang He, Fanfan Shen, Qing'an Li, and Hai Tan. 2019. Memory-aware TLP throttling and cache bypassing for GPUs. *Cluster Computing* 22 (2019), 871–883. https://doi.org/10.1007/s10586-017-1396-0

[36] Yupu Zhao, Hong Lv, Jun Li, and Lulu Zhu. 2022. High performance and resource efficient FFT processor based on CORDIC algorithm. *EURASIP Journal on Advances in Signal Processing* 2022, 1 (2022), 23. https://doi.org/10.1186/s13634-022-00855-6

Received 2025-03-21; accepted 2025-04-21