# Concurrent FFT Execution on GPUs in Real-Time

Syed W. Ali, Joseph Goh, Joshua Bakita, Samarjit Chakraborty, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill
{swali, jgoh, jbakita, samarjit, anderson}@cs.unc.edu

***Abstract*—Fourier transforms are vital for a broad range of signal-processing applications. Accelerating FFTs with GPUs offers an orders-of-magnitude improvement vs. CPU-only FFT computation. However, two problems arise when executing FFT tasks with other GPU work. First, concurrent GPU use introduces unpredictability in the form of lengthy response times. Second, it is unclear how to best parameterize and schedule FFT tasks to meet the throughput and timeliness constraints of real-time signal processing. This work investigates how FFT and other GPU-using tasks can concurrently access a GPU while maintaining bounded response-time guarantees without sacrificing throughput. In our experiments, the techniques proposed by this work result in an up to $17\%$ improvement in worst-case FFT response times.**

***Index Terms*—GPU, FFT, signal processing, real-time systems**

## I. Introduction

Fourier analysis, often performed using Fast Fourier Transform (FFT) algorithms, are indispensable in signal processing applications. For instance, methods for spectrum sensing, the detection of transmitters across the wireless spectrum, may utilize FFTs to identify occupied frequencies or trends in the signals therein. However, executing FFTs on large, high-frequency inputs is a computationally arduous task.

GPUs are well suited to address this challenge, as they can perform the matrix-like operations of FFT algorithms in parallel, greatly improving data throughput. NVIDIA GPUs are standard in industry where cuFFT [1], NVIDIA's FFT library, is used to achieve orders-of-magnitude speedup in signal-processing algorithms (compared to CPU-only execution) [2]–[9]. Furthermore, GPUs can be shared among tasks, whether they be signal-processing tasks or a combination of distinct workloads.

Unfortunately, concurrently executing multiple tasks on the same GPU in a naïve manner can cause highly unpredictable response times [10]–[13]. Conversely, while task-exclusive GPU access can mitigate such unpredictability, GPU throughput is severely reduced for systems with tasks that cannot fully utilize the GPU's computational capacity [14]. Furthermore, some signal-processing systems adhere to precise, real-world timing constraints. Such constraints may be inherent to the design and operation of their algorithms [15]–[17] or imposed by components relying on their output for real-time correctness [18], [19]. Without predictable response times for GPU work, signal-processing tasks cannot be guaranteed to meet such deadlines, thus weakening the system's reliability.

Work supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120, and ONR contract N0001424C1127.

In this work, we derive predictable response times for cuFFT in a system with many GPU-using tasks. We build upon existing hardware partitioning techniques for NVIDIA GPUs and explore how such techniques may be best implemented and analyzed for FFT tasks. Our approach successfully mitigates interference between concurrent GPU tasks and provides provable response-time guarantees, thus facilitating verifiable real-time operation. We also show how methodical selection of GPU partitioning and scheduling parameters can improve worst-case response times. The proposed approach is generalizable to any GPU work, including other GPU-based FFT implementations, of which many exist [20]–[24]. Our investigation of cuFFT serves as a representative example for how GPU-driven systems can provide real-time correctness without compromising throughput.

**Related work.** There has been extensive work on improving FFT performance on GPUs, where performance gains are typically achieved by optimizations to FFT algorithms that best leverage the parallel architecture and computational cores found in GPUs [9], [20]–[25]. We instead aim to improve predictability irrespective of the underlying design and implementation of FFT algorithms. Crucially, existing work does not, to our knowledge, consider the existence of other GPU-using workloads in the system. Thus, existing approaches cannot guarantee the timely completion of FFT tasks when shared GPUs incur unpredictable delays in computation.

Prior work has provided insight into GPU hardware provisioning and scheduling [10]–[13], [26] and developed mechanisms to predictably partition and share NVIDIA GPUs [14], [26]–[28]. These works focus primarily on predictability, leaving implementation details on average-case performance or potential pitfalls under-explored. Thus, it is unclear how best to apply such techniques to GPU tasks such as cuFFT. Our work addresses these shortcomings through an implementation-based approach examining the concurrent scheduling of cuFFT tasks. Leveraging recent advancements in spatial partitioning [26] and real-time locking protocols for GPUs [14], we demonstrate how predictable, concurrent execution can be achieved for FFT tasks with minimal overheads.

**Contributions.** Our contribution is threefold:

(i) We demonstrate the scheduling of FFTs on an NVIDIA GPU to enable provable response-time bounds.
(ii) We propose a general GPU-partitioning approach backed by measurements on cuFFT kernels.
(iii) We evaluate our approach via experiments and demonstrate reduced response times and improved throughput.

2377-5750/25/$31.00 ©2025 IEEE
DOI 10.1109/PDP66500.2025.00029

**Organization.** After covering the necessary background information in Sec. II, we outline in Sec. III how FFT tasks on the GPU can be modified and analyzed to allow for predictable, concurrent GPU access. In Sec. IV we apply our method to FFT tasks with representative inputs and experimentally evaluate our approach. Sec. V concludes.

## II. Preliminaries

In this section, we provide the background necessary for our GPU-partitioning approach for FFT tasks. We state our analysis objectives, summarize key prior work, and specify our system model.

### A. Analysis Objectives

**Spectrum-sensing motivations.** This work is motivated by the timeliness requirements of spectrum-sensing systems. Such systems can require signal samples to be collected and processed at strict time intervals to support dynamic signal detection and spectrum-access decisions [15]–[17]. For instance, Shi *et al.* [15] proposed the use of a precise schedule for when and how long to access each frequency channel in order to maximize detection likelihood. However, the work acknowledges that its implementation adheres to its schedule on a "best-effort" basis and is unable to guarantee sensing deadlines are reliably met. By enabling predictable execution times for FFTs on GPUs, we seek to alleviate the challenge of designing and analyzing a spectrum-sensing system to meet stringent real-time requirements.

**Real-time correctness.** A system is said to be *real-time correct* if each task in the system can meet their timing requirements, usually specified in the form of deadlines by which task invocations must finish. Even for applications without hard deadlines, it is often desirable for the system to be soft real-time correct, *i.e.*, ensured bounded latency or only subject to a limited pattern of deadline misses [29], [30]. An application amenable to real-time correctness guarantees may also be allowed to share a hardware platform with other, more safety-critical applications [31].

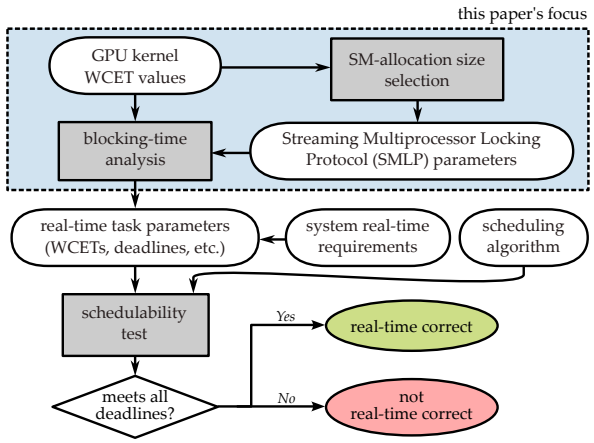


Fig. 1. High-level process of determining real-time correctness of a system. This paper's focus is shown in the dotted box.

We illustrate the process of designing and analyzing a system for real-time correctness in Fig. 1. Typically, a system's real-time correctness is determined by employing a *schedulability test*, which determines whether all tasks of the system under analysis are guaranteed to meet all of their deadlines under a given scheduling algorithm. Schedulability tests require task properties such as their *worst-case execution times* (WCETs) to be known and upper-bounded. In order to derive reliable WCET values of a GPU-using task, one must not only ensure bounded execution time on the GPU, but also bounded *blocking time*, time spent waiting for access to hardware accelerators such as GPUs. Hence, our paper focuses on enabling bounded execution and blocking times for FFT tasks on the GPU, thereby enabling verification of real-time correctness in a signal-processing system.

### B. Partitioning and Sharing of NVIDIA GPUs

Our proposed method leverages prior work on predictable hardware partitioning and sharing of NVIDIA GPUs. Recent work by Bakita and Anderson on `libsmctrl` [26] enables NVIDIA GPUs to be spatially partitioned through the allocation of individual *streaming multiprocessors* (SMs), a basic unit of computing hardware in NVIDIA GPUs, shown in Fig. 2. To obtain provable upper bounds for blocking and execution times for GPU accesses, we employ a real-time locking protocol, as relying on the default OS scheduler typically provides no response-time guarantees. In particular, our method adapts the recently introduced *Streaming Multiprocessor Locking Protocol* (SMLP) [14], which arbitrates exclusive access to SMs, thereby reducing the interference incurred by work co-scheduled on GPUs. The SMLP improves upon coarser-grained locking methods to arbitrate GPU access, providing lower analytical and observed response-time bounds while significantly improving GPU utilization and throughput.

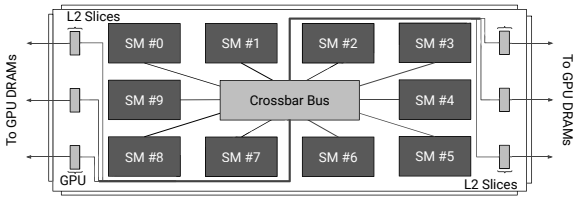


Fig. 2. Simplified NVIDIA GP106 die illustrating how the GPU's computational units are organized into streaming multiprocessors (SMs).

### C. System Model

We now present our system model, focusing on the GPU and work executing on it. We do not focus on the CPU, but discuss interactions with the CPU in Sec III-A.

**GPU-task model.** We consider a set of $\mathcal{N}$ *GPU kernels*, *i.e.*, GPU programs, executing on a single discrete GPU. We denote the $i^{\text{th}}$ GPU kernel by $\tau_i$. An invocation of the $i^{\text{th}}$ GPU kernel by the CPU is referred to as a *job* of $\tau_i$.

We assume that the same GPU kernel does not execute multiple jobs simultaneously. The GPU is composed of $\mathcal{H}$ total SMs, across which jobs execute. In order for GPU access to be arbitrated by the SMLP, we require each job to *request* exclusive access to a subset of the GPU's SMs before it is allowed to execute.

The SMLP can allocate each job a variable number of SMs, provided that no two concurrent jobs' allocations overlap. The set of SM-allocation sizes permitted for each $\tau_i$, denoted by $S_i$, is specified prior to execution. We require $\forall i,\ 1 \in S_i$ such that a job can begin execution as soon as any number of SMs become available. We define $L_{i,k}$ as the WCET of kernel $\tau_i$ when allocated exactly $k$ SMs. We use $L_i^{\max}$ to denote the largest WCET of $\tau_i$ among all permitted values of $k$ such that

$$L_i^{\max} = \max_{k \in S_i} L_{i,k}.$$

We describe later in Sec. III how GPU kernels' $L_{i,k}$ values may be measured, the SMLP configured correspondingly with ideal $S_i$, and, based on analysis from [14], how to upper bound the time each $\tau_i$ spends from request to completion.

**cuFFT.** In this paper, we use cuFFT [1], a closed-source FFT library provided by NVIDIA, for all FFT computation. An FFT task can be specified using cuFFT, which is then executed by SMs as a GPU kernel. We have chosen cuFFT as our exemplar FFT implementation for two reasons. First, cuFFT is widely accessible. It does not require developers to be intimately familiar with GPU programming APIs and comes bundled with NVIDIA's developer toolkit. Second, there exists extensive work on GPU-accelerated signal processing relying on cuFFT [2]–[9]. Thus, our work has immediate applicability to existing work on GPU-accelerated signal processing.

## III. Real-Time FFT Tasks

In this section, we show how cuFFT and other GPU-using tasks can be modified to use the SMLP for predictable GPU access. We adapt and summarize analysis for the SMLP from [14] to provide blocking-time and response-time guarantees. Then, based on extensive profiling of cuFFT kernel response times, we describe performance trends across different SM-allocation sizes. Finally, informed by our measurements and observations, we show how to select permitted SM-allocation sizes for GPU kernels using the SMLP such that blocking and response times are optimized.

### A. Modifications for GPU Partitioning

The modifications to GPU-using tasks needed to support concurrency and to be compatible with the SMLP are straightforward, requiring no changes to the GPU kernel itself.

**Launching GPU kernels.** On NVIDIA GPUs, GPU work is enqueued on *CUDA streams*, where work in one stream can run in parallel with work enqueued on other streams. As such, we require each job to launch its GPU kernels, whether they be cuFFT or user-defined kernels, with a CUDA stream unique to that job. Only asynchronous CUDA functions should be used to allow multiple streams to be launched in parallel. For instance, one should use `cudaMemcpyAsync` instead of `cudaMemcpy`, as the former may be specified with a CUDA stream and is performed asynchronously with any other asynchronous operations. Additionally, `cudaStreamSynchronize` should be used to ensure all prerequisite asynchronous operations are complete before using the GPU's compute capacity.

**Algorithm 1** cuFFT kernel launch procedure

```
1  void cudakernel(stream, h_in, h_out) {
2      cudaMemcpyAsync(d_in, h_in, ..., stream);
3      cufftPlan(plan, ...);
4      cufftSetStream(plan, stream);
5      cudaStreamSynchronize(stream);
6
7      mask = smlp_lock(allowed_sizes);
8      libsmctrl_set_stream_mask(stream, mask);
9      cufftExec(plan, d_in, d_out);
10     cudaStreamSynchronize(stream);
11     smlp_unlock(mask);
12
13     cudaMemcpyAsync(h_out, d_out, ..., stream);
14     cudaStreamSynchronize(stream);
15 }
```

**Applying SM partitioning.** Lines 7–11 of Alg. 1, highlighted in red, demonstrate how one would invoke the SMLP and `libsmctrl` for a cuFFT kernel. Note that all GPU kernels in the system must request GPU access through the SMLP by performing these steps. While Alg. 1 uses cuFFT, the procedure can be generalized to any GPU kernel by replacing `cufftExec` (line 9) with the desired kernel to be executed. GPU requests are necessary as interference is mitigated by arbitrating access to SMs.

When a job of $\tau_i$ is ready to execute on the GPU, it issues a lock request to the SMLP by calling `smlp_lock` (line 7). If no SMs are idle at that moment, the SMLP suspends the requesting job until other GPU work finishes and at least one SM becomes available. For accessibility to the reader, we lightly modify the SMLP to enqueue and satisfy requests in FIFO order instead of taking per-job priorities into consideration. We describe how the blocking-time bound for our adapted SMLP is derived and incorporated into the WCET of each GPU-using task in Sec. III-D.

As soon as any SMs become idle, the SMLP allocates some or all of the available SMs to a requesting job and `smlp_lock` returns a corresponding SM mask. The mask is then applied to the requesting job's stream using `libsmctrl` (line 8), ensuring spatially partitioned GPU access. The exact number of SMs allocated to each request is dependent on the number of SMs available at the moment the request is satisfied, as well the allowed SM-allocation sizes specified for each task in $S_i$ (`allowed_sizes` on line 7). A satisfied request is assigned the largest SM-allocation size permitted by $S_i$ that does not exceed the number of available SMs.

Additionally, partitioning of SMs only benefits the computation portion of a GPU kernel. As such, acquisition of the SMLP lock and application of the corresponding SM mask

should be performed immediately before actual execution begins. For instance, lines 2–5 of Alg. 1 perform functions such as copying memory to the GPU and configuring the kernel to be launched. Because these operations can have significant latency but do not perform computation on the GPU, calling `smlp_lock` too early would result in idle SMs. Similarly, `smlp_unlock` should be performed immediately once kernel computation finishes to allow other kernels to access the now-freed SMs as soon as possible.

### B. Response-Time Analysis

Before stating our proposed method for selecting permitted SM-allocation sizes, we briefly adapt and summarize blocking-time and response-time analysis of the SMLP from [14].

We first introduce additional notation. Let $\mathcal{A}_{i,k} = k \cdot L_{i,k}$, *i.e.*, $\mathcal{A}_{i,k}$ denotes the worst-case cumulative time the GPU kernel $\tau_i$ may consume across $k$ SMs. Similar to $L_i^{\max}$, we use $\mathcal{A}_i^{\max}$ to denote the largest value $\mathcal{A}_{i,k}$ among all permitted SM-allocation sizes $k$, such that

$$\mathcal{A}_i^{\max} = \max_{k \in S_i} \mathcal{A}_{i,k}. \tag{1}$$

The following two lemmas provide blocking-time and response-time bounds for the SMLP.[1] We illustrate the worst-case scenarios accounted for by Lem. 1 and 2 in Fig. 3.

**Lemma 1.** (Adapted from [14]) The duration a job of $\tau_i$ will be suspended by the SMLP waiting for SMs to be allocated to it is no longer than

$$\sum_{j \neq i} \frac{\mathcal{A}_j^{\max}}{\mathcal{H}}. \tag{2}$$

*Proof.* Let $J_i$ denote an arbitrary job of $\tau_i$. Recall that we require each GPU kernel to execute at most one job at once. Thus, in the worst case, $J_i$'s lock request must be queued behind exactly one request from each GPU kernel other than $\tau_i$. By definition of $\mathcal{A}_i^{\max}$ given by (1), the cumulative work completed by SMs for jobs ahead of $\tau_i$'s job is at most

$$\sum_{j \neq i} \mathcal{A}_j^{\max}.$$

Since we require $1 \in S_i$, the job of $\tau_i$ can begin executing as soon as any SMs become available and no longer need to service jobs enqueued earlier. Since each occupied SM completes one unit of work per unit of time, the longest period all $\mathcal{H}$ SMs can be occupied with jobs ahead of $J_i$ is

$$\sum_{j \neq i} \frac{\mathcal{A}_j^{\max}}{\mathcal{H}}$$

as desired. □

From Lem. 1, we also obtain a response-time bound for jobs using the SMLP, expressed in the following lemma.

[1]Blocking-time analysis from [14] is less pessimistic and more appropriate when used in conjunction with the unmodified SMLP and a priority-based real-time scheduling algorithm.

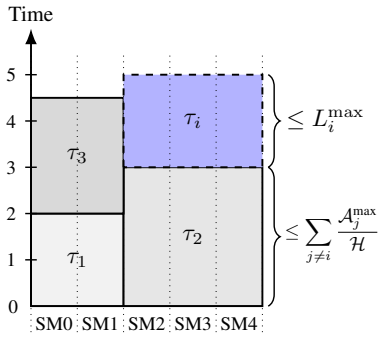


Fig. 3. Example schedule illustrating blocking and response times for a job of GPU kernel $\tau_i$. The job of $\tau_i$ is scheduled after being blocked for no longer than the summation from Lem. 1. Once scheduled, it executes for no longer than $L_i^{\max}$ and its response time is bounded correspondingly by Lem. 2.

**Lemma 2.** The response time of a job of $\tau_i$, *i.e.*, the duration from its SMLP lock request to completion, is no longer than

$$L_i^{\max} + \sum_{j \neq i} \frac{\mathcal{A}_j^{\max}}{\mathcal{H}}. \tag{3}$$

*Proof.* Recall that $L_i^{\max}$ denotes the greatest possible WCET of $\tau_i$ for all permitted SM-allocation sizes. Thus, the response time of a job of $\tau_i$ is at most the sum of its blocking time, bounded by (2) from Lem. 1, and $L_i^{\max}$. □

In analysis of real-time systems, response-time bounds such as those derived using Lem. 2 are directly incorporated into the WCET of CPU tasks which launch GPU kernels. This requires additional hardware capacity to be provisioned for such tasks. In addition to affecting the responsiveness of individual tasks, increased GPU response-time bounds make it more difficult for real-time tasks to meet their deadlines, further reducing the system's effective capacity. Thus, in the remainder of this section, we outline a process for measuring values of $L_{i,k}$ and choosing $S_i$ correspondingly such that response-time bounds are not unnecessarily inflated.

### C. WCET Measurement for GPU Kernels

A prerequisite to determining the ideal $S_i$ for a GPU kernel $\tau_i$ is to estimate $L_{i,k}$, the WCET of $\tau_i$ when allocated $k$ SMs, for each possible value of $k$. These estimates may be obtained by executing GPU kernels for each SM-allocation size and recording the worst-observed execution times. When measuring the execution time of a GPU kernel, any SMs not allocated to the GPU kernel under analysis should execute memory-intensive and compute-intensive dummy kernels to simulate maximal interference from concurrent tasks. Execution time measurements should not include steps before lock or unlock requests (lines 7 and 11 in Alg. 1, respectively).

**Addressing non-uniformity in SMs.** Similar to CPU cores, each SM is not necessarily identical in performance. The specific SMs allocated may impact response times, even for identical allocation sizes. Moreover, other factors such as the physical distance between SMs in the same GPU may also

affect response times. Thus, if one is to account for true worst-case performance, all possible SM allocations should be tested.

Unfortunately, it is often not feasible to test every possible combination of SMs for each GPU kernel. Therefore, to obtain safer, albeit somewhat pessimistic WCETs, we suggest estimating a worst-case slowdown factor to be incorporated into all measured response times. This effect was negligible in our experiments using cuFFT kernels. However, the necessity and magnitude of such a slowdown factor depends on which hardware and GPU kernels are used.

**Performance trends in cuFFT.** To better understand the impact that $L_{i,k}$ values and the selection of $S_i$ can have on response times and throughput, we have collected extensive response-time measurements for cuFFT kernels. We estimated the WCET of cuFFT kernels for each input sample size from $\{2^{10}, 2^{11}, \ldots, 2^{24}\}$, where one sample is a 32-bit floating point value. Our measurements were performed using the random input generation method and hardware platforms detailed in Sec. IV. Each configuration was executed 1,000 times, recording the worst-observed execution times. To simulate maximal GPU saturation, each SM not allocated to the kernel under analysis were configured to execute a cuFFT kernel with $2^{12}$ randomly generated input samples in a loop. From our results, we observe a few key trends.

**Observation III-1.** *The benefit of an increased SM-allocation size diminishes at higher SM counts.* Fig. 4 illustrates this (the $x$-axis represents the SM-allocation size, and the $y$-axis the measured WCET). For cuFFT kernels of input size $2^{24}$, measured WCETs decrease roughly proportional to the number of SMs up to about four, but allocating 10 SMs, for instance, is only approximately 6.15 times faster than a single SM. This trend was originally noted in [14], which recommends avoiding SM-allocation sizes that offer little incremental execution time reduction. Response times of cuFFT kernels with small input sample sizes (around $2^{10}$ to $2^{18}$) scaled poorly with SM-allocation sizes greater than one, implying that they could not fully utilize more than one SM. Fig. 4 only shows results for the RTX 3060 Ti, but these patterns persisted on other GPUs.

cuFFT kernels likely scale poorly to large numbers of SMs due to some combination of 1) intrinsic algorithmic limitations on the amount of parallelism, 2) the high overhead of synchronizing work across the GPU, and 3) a loss of cache locality (each SM contains its own data, instruction, and constant caches).

**Observation III-2.** *Our execution-time graphs deviate somewhat from those attained by [14].* Ali et al. describe their results in [14] as imitating a "step-graph," where, as SM-allocation sizes increased, measured WCETs did not decrease in a smooth, curve-like manner as shown in Fig. 4.

**Observation III-3.** *Single-SM allocations consistently produced the lowest worst-case cumulative computation time across SMs.* That is, for a cuFFT kernel $\tau_i$ and $k > 1$, $\mathcal{A}_{i,1} \lesssim \mathcal{A}_{i,k}$. This implies that, with respect to per-SM throughput, $\mathcal{A}_{i,1}$ can serve as reference for the maximum efficiency by which $\tau_i$ can be completed. SM-allocation sizes

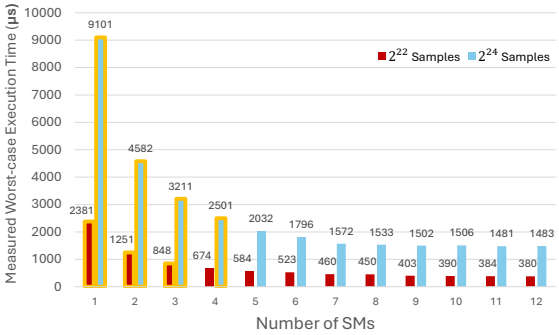


Fig. 4. Largest-observed execution times of cuFFT kernels by number of SMs allocated, measured on an NVIDIA RTX 3060 Ti. Bars highlighted in yellow correspond to permitted SM-allocation sizes we use with the SMLP.

$k$ with noticeably greater $\mathcal{A}_{i,k}$ values can be inferred to be less efficient and should be avoided.

Based on these observations, we have developed a simple heuristic technique for cuFFT kernels to select ideal SM-allocation sizes. While the trends observed in cuFFT may not be universal to all workloads, we believe our technique to be applicable to most FFT tasks and other GPU kernels that exhibit similar trends.

### D. Finding Ideal SM-Allocation Sizes

While it is possible to use a one-size-fits-all set of permitted SM-allocation sizes or apply no restrictions at all, inefficient SM allocations may negatively impact the throughput and responsiveness of GPU tasks. In this subsection, we outline how one may derive $S_i$ for each GPU kernel for optimized performance and analytical guarantees.

Recall that values of $\mathcal{A}_i^{\max}$ contribute directly to the summation in (2) and (3). Intuitively, a greater $\mathcal{A}_i^{\max}$ value means that computation of $\tau_i$ occupies more of the GPU's computational capacity in the worst case, thus leading to increased blocking-time and response-time bounds.

Additionally, our WCET measurements for cuFFT kernels, described in Sec. III-C, showed response times decreasing roughly proportional to SM-allocation size, to a point. *I.e.*, for cuFFT kernel $\tau_i$ and small $k, l \in \mathbb{N}$, we have

$$k \cdot L_{i,k} \approx l \cdot L_{i,l} \quad \Leftrightarrow \quad \mathcal{A}_{i,k} \approx \mathcal{A}_{i,l}.$$

Fig. 5 illustrates the finding that for larger SM-allocation sizes, response times do not always see a commensurate decrease. Thus, large SM allocations may be inefficient.

Based on these intuitions, we propose the following heuristic for selecting allowable SM-allocation sizes for kernel $\tau_i$:

**SM-allocation size selection method.** For each $\tau_i$, set

$$S_i = \{k \in [1, \mathcal{H}] \mid \mathcal{A}_{i,k} \leq \mathcal{A}_{i,1} \cdot \rho_i\} \tag{4}$$

where $\rho_i \geq 1$ is a real-valued parameter.

As noted in Observation III-3, our method assumes $\mathcal{A}_{i,1}$ corresponds to a maximally efficient SM-allocation size of one. From Observation III-1, we see that, for some kernels, efficiency is maintained for increased SM-allocation sizes, up

TABLE I
RECOMMENDED SM PARTITION SIZES FOR CUFFT BY INPUT SIZE

| GPU model | $2^{10} \sim 2^{13}$ | $2^{14} \sim 2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ | $2^{21}$ | $2^{22}$ | $2^{23}$ | $2^{24}$ |
|---|---|---|---|---|---|---|---|---|---|
| NVIDIA GTX 1070 | $\{1\}$ | $\{1,2\}$ | $\{1,2\}$ | $\{1,2\}$ | $\{1,2\}$ | $\{1,2,3\}$ | $\{1,2,3,4\}$ | $\{1,2,3,4\}$ | $\{1,2,3,4\}$ |
| NVIDIA RTX 2080 Ti | $\{1\}$ | $\{1\}$ | $\{1,2\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3,4,5\}$ | $\{1,2,3,4,5,6\}$ | $\{1,\ldots,7\}$ | $\{1,\ldots,8\}$ |
| NVIDIA RTX 3060 Ti | $\{1\}$ | $\{1\}$ | $\{1,2\}$ | $\{1,2\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3,4\}$ |

to a point. As such, our heuristic allows $k$ SMs to be a permitted allocation size if $\mathcal{A}_{i,k}$ is within a tolerated degree of inefficiency specified by $\rho_i$. Such tolerance is necessary, as overly restrictive $S_i$ may cause difficulty in the SMLP's allocation of SMs, causing some to be idle. Our proposed strategy selects $S_i$ such that $\mathcal{A}_i^{\max}$ is not too large relative to $\mathcal{A}_{i,1}$ while not overly restricting GPU kernels' access to SMs. The exact value of $\rho_i$ may be customized for each GPU kernel. A higher $\rho_i$ allows for more inefficiency in the system, increasing the blocking-time bound given by (2), but can occasionally improve $\tau_i$'s responsiveness, especially when SMs would be idle otherwise.

### E. Application to cuFFT

We illustrate our SM-allocation size selection and response-time analysis process by example. We also provide recommendations for $S_i$ by target hardware and cuFFT input size.

**Example 1.** Let $\tau_i$ be a cuFFT kernel whose jobs each process input samples of size $2^{24}$. Suppose that $\tau_i$'s WCETs at different SM-allocation sizes, *i.e.*, values of $L_{i,k}$, are estimated equal to the measurements plotted in Fig. 4.

Using the heuristic proposed in (4) to select SM-allocation sizes for $\tau_i$ disqualifies allocation sizes $k$ such that

$$\mathcal{A}_{i,k} > \mathcal{A}_{i,1} \cdot \rho_i = 9101 \cdot \rho_i.$$

We use a value of $\rho_i = 1.1$, allowing SM-allocation sizes that are at most 10% more inefficient compared to executing $\tau_i$ on a single SM. Applying (4) to $k = 4$ SMs gives us

$$\begin{aligned}\mathcal{A}_{i,4} = 4 \cdot 2501 = 10004 &\leq 10011.1 \\ &= 9101 \cdot 1.1 = \mathcal{A}_{i,5} \cdot \rho_i,\end{aligned}$$

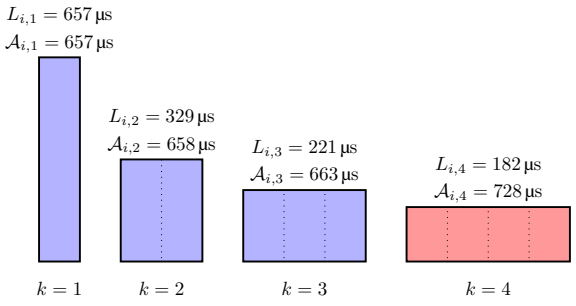


Fig. 5. Boxes representing cuFFT kernel execution at SM-allocation sizes $k \in \{1,2,3,4\}$. The width, height, and area correspond to $k$, $L_{i,k}$, and $\mathcal{A}_{i,k}$ respectively. While an increase from three SMs to four reduces execution time somewhat, the area, *i.e.,* the total computation time across used SMs increases excessively such that we set $4 \notin S_i$.

whereas $k = 5$ SMs gives us

$$\mathcal{A}_{i,5} = 5 \cdot 2032 = 10160 > 10011.1 = \mathcal{A}_{i,5} \cdot \rho_i.$$

Applying the heuristic to all possible SM-allocation sizes gives us $S_i = \{1,2,3,4\}$.

As seen in Ex. 1, our proposed method, alongside a reasonable value of $\rho_i$, can easily winnow out inefficient SM-allocation sizes for cuFFT kernels. We have applied this process to our comprehensive measurements of cuFFT described in Sec. III-C using $\rho_i = 1.1$. The recommended permitted SM-allocation sizes we obtain are summarized by target GPU and by cuFFT input sample size in Tab. I.

Note that our heuristic can be improved upon by testing for specific GPU task sets and by solving for specific optimization goals. However, we show in Sec. IV that the SMLP with SM-allocation sizes configured by our method fares well compared to default scheduling of cuFFT kernels.

## IV. EXPERIMENTS

In this section, we evaluate our approach with experiments on a comprehensive FFT task set. We compare the response times and throughput with and without the SMLP. We use BE-cuFFT and SMLP-cuFFT to refer to results from best-effort scheduling and from using the SMLP, respectively.

### A. Experimental Design

**Hardware platform.** We performed our tests using three different NVIDIA GPUs across three product generations: the GTX 1070, RTX 2080 Ti, and RTX 3060 Ti. Each GPU contains 15, 34, and 19 SM partitions, respectively.

**Input configurations.** Our FFT input size (*i.e.*, number of 32-bit floating point samples) for each trial was configured as one of $z \in \{2^{10}, 2^{11}, \ldots, 2^{24}\}$, with the number of concurrent cuFFT-using tasks as one of $\mathcal{N} \in \{1,2,\ldots,21\}$. We performed 1,000 trials for each combination of input size $z$ and task count $\mathcal{N}$.

**Input generation.** To obtain representative samples for FFT input, each trial, we generated $k$ signals, with $k$ sampled uniformly from $[5,50]$. Each generated signal "transmitted" random data using phase-shift keying (PSK), carrier amplitude and phase (QAM), or amplitude keying (ASK). Specifically, each signal used one of the following modulations (bits per symbol) chosen uniformly at random: 4-PSK, 8-PSK, 16-QAM, 256-QAM, 2-ASK, or 4-ASK. Lastly, the input was mixed with additive white Gaussian noise (AWGN) using a signal-to-noise (SNR) ratio selected uniformly at random from the interval $[10,30]$.

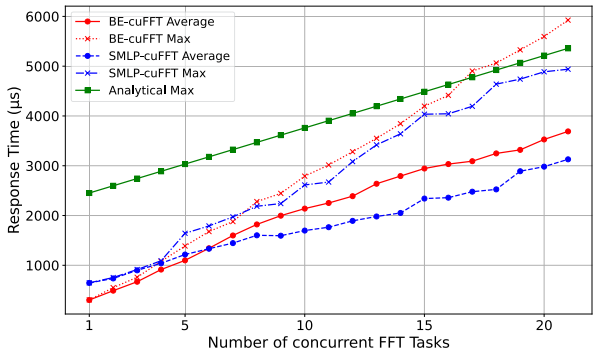


Fig. 6. Largest observed and average response times of cuFFT kernels executed concurrently on an NVIDIA RTX 3060 Ti, each with $2^{22}$ input samples. Analytical SMLP response-time bounds are shown in green.

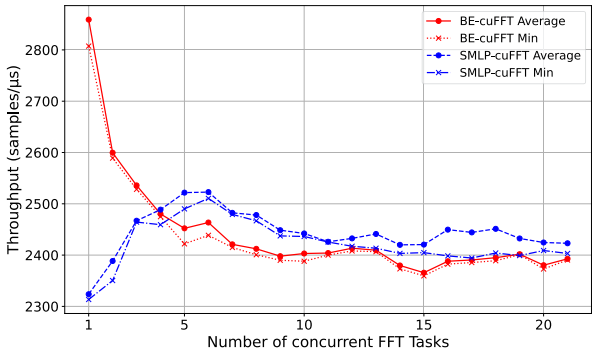


Fig. 7. Total FFT throughput across all concurrently executing cuFFT kernels on a NVIDIA RTX 3060 Ti, each with $2^{22}$ 32-bit floating point input samples.

**Methodology.** In our experiments, we recorded the response times of cuFFT kernels under BE-cuFFT and SMLP-cuFFT. Each trial synchronously launched $\mathcal{N}$ threads, with each thread executing 100 consecutive cuFFT jobs. The response time of each job was recorded, noting the average and worst-observed times across all trials. We additionally computed analytical response-time bounds under the SMLP as described in Sec. III-B and using measurements from Sec. III-C. To configure permitted SM-allocation sizes and calculate response-time bounds, we used the measurements and $S_i$ recommendations determined in Sec. III-E (using $\rho_i = 1.1$). We also recorded the throughput of cuFFT kernels, calculated per-trial as the total number of signal samples processed by the trial divided by its aggregate response time.

### B. Results

Fig. 6 shows the average, worst-case, and analytical bounds for response times and Fig. 7 shows throughput when using an NVIDIA RTX 3060 Ti and $z = 2^{22}$. These figures are representative of trends seen across all configurations. Additional graphs may be found in the full version of this paper [32]. From our results, we make the following observations.

**Observation IV-1.** *The benefit of the SMLP grows as the number of concurrent tasks increases.* This applies to worst- and average-case times (Fig. 6), and throughput (Fig. 7). For a low number of concurrent tasks, SMLP-cuFFT constrains the number of SMs a kernel may access, leaving capacity for other kernels. When few kernels run, extra SMs are left idle, leaving GPU compute capacity unused—a problem that BE-cuFFT does not suffer—explaining the under-performance of SMLP-cuFFT for a small number of concurrent FFTs.

However, despite the significant under-utilization of GPU compute cores, the slowdown is not proportionate. For example, consider the worst-case response times when running only two concurrent FFT tasks (the leftmost points in Fig. 6). Under BE-cuFFT, the max response time is 305 $\mu s$. For SMLP-cuFFT, the max is 647 $\mu s$—a 2.1× slowdown. However, SMLP-cuFFT only uses a SM partition size of up to 3 per FFT (see $S_i$ values in Tab. I); up to 6 for two FFTs. The RTX 3060 Ti hardware supports an SM partition size of up to 19, meaning that SMLP-cuFFT uses only 32% of what it could. As BE-cuFFT uses every SM, one would expect SMLP-cuFFT to run 3× slower, rather than the observed 2.1×. This milder slowdown is because cuFFT is not able to effectively utilize the large number of SMs BE-cuFFT provides (see Obs. III-1).

The point at which SMLP-cuFFT can utilize as many SMs as BE-cuFFT (≈ 6 concurrent FFT tasks) is where SMLP-cuFFT matches the BE-cuFFT response times in Fig. 6.

**Observation IV-2.** *The SMLP analytical bounds hold.* In no cases have we observed the response time of a SMLP-cuFFT task exceed the analytically guaranteed response-time bound. This is visible in Fig. 6, as the "Max" line approaches, but does not exceed, the straight "Analytical Max" line.

**Observation IV-3.** *Applying the SMLP can increase overall system throughput.* This is visible in Fig. 7, where SMLP-cuFFT's average throughput exceeds BE-cuFFT's for five or more concurrent FFT tasks. As in Obs. IV-1, this throughput benefit occurs when all SMs are utilized and can be configured with $\rho_i$. A small value of $\rho_i$ constrains each kernel to a small number of SMs (*i.e.*, small $S_i$), leaving room for more concurrent FFT tasks. For systems containing few concurrent FFT tasks, a higher value of $\rho_i$ allows access to more SMs per task. Thus, throughput improvements are obtained with few concurrent FFT tasks and a large $\rho_i$ at the cost of reducing the total possible number of concurrently executing FFTs.

**Why a throughput benefit?** With the SMLP, different FFTs are guaranteed to run on mutually exclusive sets of SMs. This can yield a throughput benefit against the default scheduler (which will run different tasks on the same SM) because several caches exist on a per-SM level, and having more homogeneous work on a single SM can allow for greater cache locality and less time spent in memory stalls.

Across Obs. IV-1 through IV-3, we see that SMLP-cuFFT beats BE-cuFFT on all metrics when SMLP-cuFFT is able to access the entire GPU. The weak performance of SMLP-cuFFT for low levels of concurrency could be addressed in deployed systems by tuning $\rho_i$ to suit the expected amount of

concurrent work. In our experiments, we intentionally fixed $\rho_i$, no matter the concurrent FFT task count, to target systems with arbitrary numbers of concurrent FFT tasks.

In summary, we demonstrated that the SMLP provides bounded response times for FFTs, while matching or exceeding the throughput of the default scheduler under high levels of concurrency. This means that the SMLP is not only real-time correct, but suitable for systems where throughput is critical.

## V. Conclusion

We have demonstrated how FFT execution on GPUs can be improved with respect to predictability and real-time guarantees by incorporating GPU hardware partitioning and real-time locking techniques. We have shown how our approach may be applied without sacrificing, and at times even improving, GPU throughput. Our approach applies to any GPU-based application and our analysis is accessible to existing spectrum sensing systems. In future work, we intend to explore the relationship between real-time scheduling and spectrum sensing accuracy. We also plan to examine ways to improve real-time guarantees and throughput for other signal-processing workloads on the GPU, such as those based on AI/ML techniques.

## References


[1] NVIDIA, "cuFFT." [Online]. Available: https://developer.nvidia.com/cufft

[2] S. Marshall, G. Vanhoy, A. Akoglu, T. Bose, and B. Ryu, "GPGPU Based Parallel Implementation of Spectral Correlation Density Function," *Journal of Signal Processing Systems*, vol. 92, p. 71–93, 2020.

[3] X. Zhao, P. Liu, B. Wang, and Y. Jin, "GPU-Accelerated Signal Processing for Passive Bistatic Radar," *Remote Sensing*, vol. 15, no. 22, 2023.

[4] D. Xu, Y. Huang, and J. U. Kang, "GPU-accelerated non-uniform fast Fourier transform-based compressive sensing spectral domain optical coherence tomography," *Opt. Express*, vol. 22, no. 12, pp. 14 871–14 884, Jun 2014.

[5] Á. Ordóñez, F. Argüello, and D. B. Heras, "GPU Accelerated FFT-Based Registration of Hyperspectral Scenes," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 10, no. 11, pp. 4869–4878, 2017.

[6] P. Nejedly, F. Plesinger, J. Halamek, and P. Jurak, "CudaFilters: A SignalPlant library for GPU-accelerated FFT and FIR filtering," *Software: Practice and Experience*, vol. 48, no. 1, pp. 3–9, 2018.

[7] S. Dimoudi, K. Adamek, P. Thiagaraj, S. M. Ransom, A. Karastergiou, and W. Armour, "A GPU Implementation of the Correlation Technique for Real-time Fourier Domain Pulsar Acceleration Searches," *The Astrophysical Journal Supplement Series*, vol. 239, no. 2, p. 28, dec 2018.

[8] M. Abdellah, A. Eldeib, and A. Sharawi, "High Performance GPU-Based Fourier Volume Rendering," *International Journal of Biomedical Imaging*, vol. 2015, no. 1, p. 590727, 2015.

[9] R. S. Perdana, B. Sitohang, and A. B. Suksmono, "A survey of graphics processing unit (GPU) utilization for radar signal and data processing system," in *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, 2017, pp. 1–6.

[10] J. Bakita and J. H. Anderson, "Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2024, pp. 294–305.

[11] N. Otterness and J. H. Anderson, "Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, ser. RTNS '21, 2021, p. 24–34.

[12] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, 2018, pp. 20:1–20:21.

[13] G. A. Elliott, "Real-Time Scheduling for GPUs with Applications in Advanced Automotive Systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, August 2015.

[14] S. W. Ali, Z. Tong, J. Goh, and J. H. Anderson, "Predictable GPU Sharing in Component-Based Real-Time Systems," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 298, 2024, pp. 15:1–15:22.

[15] L. Shi, P. Bahl, and D. Katabi, "Beyond Sensing: Multi-GHz Realtime Spectrum Analytics," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, p. 159–172.

[16] V. Kone, L. Yang, X. Yang, B. Y. Zhao, and H. Zheng, "On the Feasibility of Effective Opportunistic Spectrum Access," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10, 2010, p. 151–164.

[17] S. Sarkar, M. Buddhikot, A. Baset, and S. K. Kasera, "DeepRadar: A Deep-Learning-based Environmental Sensing Capability Sensor Design for CBRS," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '21, 2021, p. 56–68.

[18] L. J. Wong, W. H. Clark, B. Flowers, R. M. Buehrer, W. C. Headley, and A. J. Michaels, "An RFML Ecosystem: Considerations for the Application of Deep Learning to Spectrum Situational Awareness," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 2243–2264, 2021.

[19] L. J. Wong, W. C. Headley, and A. J. Michaels, "Specific Emitter Identification Using Convolutional Neural Network-Based IQ Imbalance Estimators," *IEEE Access*, vol. 7, pp. 33 544–33 555, 2019.

[20] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–11.

[21] B. Li, S. Cheng, and J. Lin, "tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 1–11.

[22] L. Gu, X. Li, and J. Siegel, "An empirically tuned 2D and 3D FFT library on CUDA GPU," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10, 2010, p. 305–314.

[23] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009.

[24] X. Cui, Y. Chen, and H. Mei, "Improving Performance of Matrix Multiplication and FFT on GPU," in *2009 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 42–48.

[25] D. B. Lloyd, C. Boyd, and N. Govindaraju, "Fast Computation of General Fourier Transforms on GPUs," in *2008 IEEE International Conference on Multimedia and Expo*, 2008, pp. 5–8.

[26] J. Bakita and J. H. Anderson, "Hardware Compute Partitioning on NVIDIA GPUs," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 54–66.

[27] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 29–41.

[28] N. Otterness, V. Miller, M. Yang, J. H. Anderson, F. D. Smith, and S. Wang, "GPU Sharing for Image Processing in Embedded Real-Time Systems," in *2016 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2016.

[29] C. J. Kenna, J. L. Herman, B. B. Brandenburg, A. F. Mills, and J. H. Anderson, "Soft Real-Time on Multiprocessors: Are Analysis-Based Schedulers Really Worth It?" in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 93–103.

[30] M. Caccamo and G. Buttazzo, "Exploiting skips in periodic tasks for enhancing aperiodic responsiveness," in *Proceedings Real-Time Systems Symposium*, 1997, pp. 330–339.

[31] A. Burns and S. Baruah, "Multi-model workload specifications and their application to cyber-physical systems," *Research Directions: Cyber-Physical Systems*, vol. 2, p. e3, 2024.

[32] S. W. Ali, J. Goh, J. Bakita, S. Chakraborty, and J. H. Anderson, "Concurrent FFT Execution on GPUs in Real-Time," 2025, full version with additional graphs. [Online]. Available: https://jamesanderson.web.unc.edu/papers/