Paella: Low-latency Model Serving with Software-defined GPU Scheduling

Kelvin K.W. Ng
University of Pennsylvania
kelvinng@seas.upenn.edu

Henri Maxime Demoulin
DBOS, Inc.
maxdml@dbos.dev

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

Abstract

Model serving systems play a critical role in multiplexing machine learning inference jobs across shared GPU infrastructure. These systems have traditionally sat at a high level of abstraction—receiving jobs from clients through a narrow API and relying on black-box GPU scheduling mechanisms when dispatching them. Fundamental limitations in the built-in GPU hardware scheduler, in particular, can lead to inefficiency when executing concurrent jobs. The current abstraction level also incurs system overheads that are similarly most significant when the GPU is heavily shared.

In this paper, we argue for co-designing the model compiler, local clients, and the scheduler to bypass the built-in GPU scheduler and enable software control of kernel execution order. Doing so enables the use of arbitrary scheduling algorithms and reduces system overheads throughout the critical path of inference.

CCS Concepts: • **Software and its engineering** → Compilers; **Scheduling**; *Real-time systems software*; **Software performance**.

Keywords: GPUs, low-latency model serving, machine learning inference, scheduling

ACM Reference Format:
Kelvin K.W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613163

1 Introduction

Machine learning (ML) continues to find utility in every aspect of today's software systems. In applications, ML-based predictions are used for everything from computing database indices to generating recommendations for users [43, 70]. In computer networks and systems, they have been proposed as a replacement for heuristics and hand-tuned procedures in scheduling, intrusion detection, and configuration optimization [6, 14, 41, 60, 69, 71]. The proliferation of these techniques—both in the critical path of user requests and multiple points therein—necessitates minimizing the latency and maximizing the throughput of ML inference.

Model serving systems like NVIDIA's Triton [52], Clipper [19], and Clockwork [28] play a key role in this process—enabling multiple users and models to share GPU infrastructure. Typically sitting above the model execution engine (*e.g.*, Tensorflow, PyTorch, JAX, etc.) and separate from the client applications, these systems are responsible for receiving incoming requests, choosing the model/configuration, scheduling, and eventually dispatching model executions to the underlying execution engines.

Unfortunately, despite large strides toward improving the performance of these systems, today's serving platforms continue to struggle to provide low latency, particularly when the GPU is heavily shared and models exhibit high dispersion. We argue that this struggle stems from significant inefficiencies in the interactions of the serving platform with both (a) the clients from which requests arrive and (b) the GPU execution logic to which the requests are dispatched. For (a), we show that the overhead of the serving platform—encompassing serialization, de-serialization, scheduling, dispatch, etc.—can sometimes rival the execution time of the actual models. For (b), we find that inefficient FIFO-based scheduling policies (that have been baked into the CUDA runtimes, GPU drivers, and hardware) are susceptible to frequent Head-of-Line (HoL) blocking as well as mismatches between scheduling objectives and the job-completion-time targets of applications. In both cases, treating the serving platform's surrounding components as black boxes is detrimental to inference latency.

Existing approaches to addressing the above problems, including many introduced by GPU manufacturers, have provided only partial solutions. For example, allowing users to submit CUDA kernels directly to the GPU avoids communication overheads but introduces substantial context-switching overheads when running multiple GPU processes concurrently. Post-Volta MPS [2] support in GPUs can remove some of these overheads and iterative improvements to

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10...$15.00
https://doi.org/10.1145/3600006.3613163

CUDA stream queues/priorities (*e.g.*, in the Fermi and Kepler microarchitectures) expose some control of the built-in GPU scheduler to the application; however, these approaches are still prone to the problem of HoL blocking and the mismatch between scheduling objectives and application performance targets [47, 56]. In the end, the abstraction level of today's serving systems imposes overheads and drops information that is not easily mitigated by more hardware complexity.

In this paper, we present Paella, a lightweight framework for low-latency ML inference on shared GPU accelerators. Paella is a co-design of a request submission interface, model scheduler, and model execution logic that, together, ensure high utilization and minimal latency at all stages of inference request processing: submitting a job, scheduling its constituent CUDA tasks, and returning the results.

Paella's primary design goal is to enable fine-grained control over scheduling and its related tasks. When dispatching jobs to the GPU, control and subsequent careful management of GPU scheduling can circumvent inefficiencies in hardware schedulers and fundamental limitations in their scheduling policies. When receiving jobs from clients and returning results, control over communication and receive-side operation enables Paella to minimize request latency overheads while also limiting resource consumption overheads.

Supporting Paella's fine-grained control are two key innovations. The first is a lightweight and automated TVM compiler instrumentation to explicitly shine a light on currently opaque GPU scheduling mechanisms. With a detailed, real-time view of the GPU and its available resources, Paella enables softwareization of the scheduler, which, among other benefits, allows the Paella serving platform to schedule *every* CUDA kernel individually, precisely, and based on arbitrary performance/fairness metrics, regardless of the behavior of the built-in GPU scheduler. The second is a series of specialized communication channels between each of the above components—the GPU, the client, and the serving framework—that support fine-grained coordination and that borrow ideas from recent and classic work, *e.g.*, the polling and shared-memory communication of low-latency Library-OS services [10, 21, 25, 38, 59], techniques in lock-free data structures, and the low-overhead multitasking capabilities provided by co-routine mechanisms. Both innovations were challenging. Communication channels must be carefully designed to not hinder inference latency. Paella scheduler and kernel instrumentation must be lean to offer superior scheduling capabilities while keeping overheads low.

We implemented and evaluated Paella on a GPU testbed. We show that Paella's scheduling algorithm, on its own, can sustain 1.4× more load and 1.35× lower latency than the baseline. As a system and compared to NVIDIA's Triton, it can sustain up to 58× more load and 11× lower latency. It does all of this while leveraging *only a single CPU core* for scheduling and dispatching.

Beyond improving the performance of model serving, a core incentive for Paella's software-defined GPU scheduling is that, fundamentally, there is no single scheduling policy that is optimal for all workloads [68]. Hence, regardless of the current or future design of GPU schedulers, Paella's approach will benefit inference. In summary, this paper makes the following contributions:

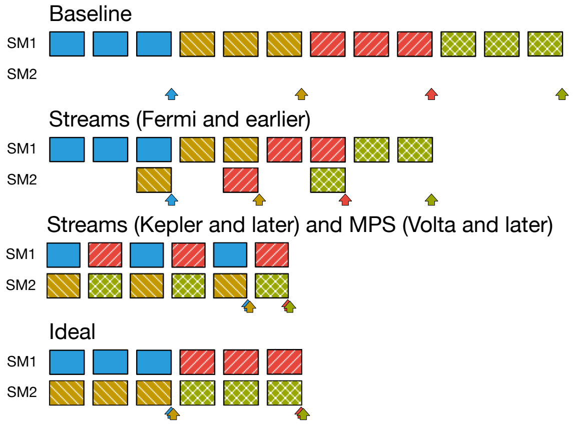
Baseline
SM1
SM2
Streams (Fermi and earlier)
SM1
SM2
Streams (Kepler and later) and MPS (Volta and later)
SM1
SM2
Ideal
SM1
SM2

Figure 1. Simplified illustration of NVIDIA GPU scheduling under different submission methods. For this example, we assume four tasks consisting of 3 kernels each, all submitted at time $t = 0$. All kernels occupy an entire SM out of 2 total SMs. Even for this simple setup, none of the supported methods of submitting the four tasks results in an ideal schedule.

- We show that existing prediction serving frameworks are hamstrung by surrounding components and that the interfaces between these components must be chosen carefully to achieve low latency and high utilization.
- We introduce a compiler-service co-design that instruments kernels to offer visibility into and full control over black-box GPU scheduling decisions.
- We present a service architecture that abstracts scheduling from the GPU. Paella uses this to minimize the latency of the complete inference pipeline: receiving requests, dispatching them to the GPU, and returning results.

2 Background and Motivation

The past few years have seen tremendous interest in integrating an increasing amount of classification, regression, clustering, and reinforcement learning into the critical path of user requests [6, 22, 43, 60, 69]. Compared to other contexts, such as model training or bulk model serving, critical-path inference tasks are distinct in their extreme sensitivity to latency and the use of smaller models (due to quantization and distillation). For example, a real-time NLP application that samples voice at 24 KHz may only have 41.5 µs to process each inference query [55]. More extreme cases also exist, *e.g.*, the ns-scale latency requirements for ML inference of particle selection in the Large Hadron Collider [16, 23] and measurement of gravitational waves [20]. In each case, inference latency determines the allowable complexity (and, thus, the accuracy and utility) of the deployed models.

The community has already made tremendous strides in speeding up the computation itself, *e.g.*, through specialized hardware [37, 45, 62], algorithms [31, 46], and batching [19]. Unfortunately, for applications with stricter latency requirements, computationally shallow models, or faster GPUs, computation is only one piece of the execution time—existing systems sometimes contribute more latency (directly or indirectly) than the computations themselves.

In this paper, we argue that to achieve low end-to-end application latency, serving frameworks need tighter integration with their surrounding components and finer-grained control over their scheduling and operation. In the remainder of this section, we provide background on the inefficiencies of existing model serving platforms and why they are difficult to address with current serving architectures.

2.1 GPUs and Mismatched Scheduling Policies

One major bottleneck to end-to-end application-level performance is the behavior of modern GPU hardware schedulers. In this section, we focus on those of NVIDIA GPUs.

Fundamentally, GPUs are specialized devices capable of high levels of parallelism. Originally intended for the sole purpose of manipulating computer graphics, toolkits like NVIDIA's CUDA have led to the use of GPUs in other fields. For machine learning, in particular, the parallelism and high memory bandwidth of GPUs lend themselves to the acceleration of computations like matrix multiplication and convolution, which are common in today's deep learning models.

At an architectural level, a GPU is a PCIe device that consists of several *streaming multiprocessors* (SMs), each of which contains functional units, a register array, and an L1 cache. An SM can run several user contexts concurrently, with each allocated a static share of the SM's local resources.

The smallest unit of GPU computation in the CUDA framework is called a *thread*. Each thread is a sequential program that users parallelize by grouping many threads into a *block* of threads that execute together on a single SM (to facilitate inter-thread shared memory and synchronization). Once a block is placed in an SM, the required resources, including registers, shared memory, and the slots for blocks and threads, are statically allocated for the duration of the block (in contrast to CPUs, where threads share resources using time-division multiplexing)[1]. Blocks can then be grouped into a *grid* to form a *kernel*; all of the threads in a kernel can execute concurrently. Kernels are the granularity at which applications generally interact with the GPU. A typical application-level task/model will consist of several kernels, often executed sequentially.

Scheduling in a modern GPU. GPUs contain a limited number of hardware queues where they receive kernel launches and consider them for scheduling. One distinguishing feature of these queues is their strict FIFO nature. In the basic case (single process, single stream), the GPU's internal scheduler will examine only the earliest-launched kernel to see if any of its blocks can be placed given the SMs' currently available resources. It does not consider blocks of subsequent kernels regardless of their resource demands.

Instead, today's GPUs provide features that help applications control what is dispatched to the hardware queues and when. For example, kernels can be submitted to different *streams* to indicate that they can be executed concurrently; with MPS, kernels can be submitted from different processes and receive similar benefits; and with Multi-Instance GPU (MIG), users can set up multiple virtual GPUs with strong isolation properties. In the end, however, users must fully trust the hardware to determine the kernel execution schedule.

What is schedulable at a given moment is, thus, heavily dependent on the order of kernel submission, the GPU's current resource utilization, and the features/configuration of the GPU. Figure 1 illustrates GPU scheduling for different NVIDIA chips under different submission methods. For the Fermi microarchitecture or earlier, GPUs only support a single hardware queue, regardless of the number of user-defined streams. So, kernels from all streams are serialized to the hardware queue in "issue order." The primary advantage of streams in these architectures occurs only after all of a kernel's blocks have been placed *and* the new head of the hardware queue has no running dependencies, in which case streams may allow the GPU to execute an independent kernel. With a natural submission order (one model at a time), only the first/last kernels of adjacent models can share the GPU (second row in the figure).

Acknowledging this issue, Kepler chips added multiple hardware queues [1] such that each stream is mapped onto one of those queues. Within each queue, operation proceeds as in earlier microarchitectures, with limited concurrency; however, the GPU can now consider the head of each hardware queue independently, increasing the opportunity to schedule independent kernels when resources are available. MPS extends this abstraction to kernels from different processes (which would have otherwise used expensive preemption). Unfortunately, while these approaches (partially) mitigate the above scheduling issues, modern GPU scheduling still leaves much to be desired.

HoL blocking in today's GPUs. As one example, we observe that today's GPUs can still exhibit HoL blocking between streams as hardware queues are limited. To preview the potential impact, we compare a naïve submission policy where all kernels are submitted together to a framework that understands GPU occupancy and can submit each job's kernel when they are able to run (*i.e.*, Paella). We use a GeForce GTX 1660 SUPER, which has 22 SMs, 1024 threads

[1] Pascal and later GPUs include preemption capabilities but they impose prohibitive swapping overheads and are not typically used in practice [7].

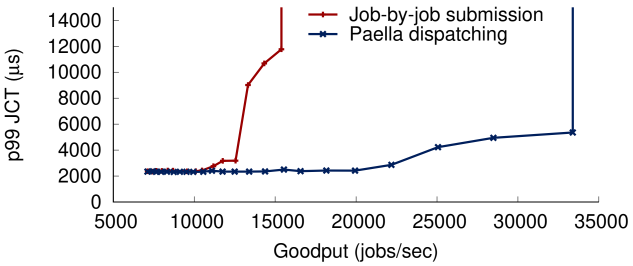

Figure 2. The traditional method of submitting all the kernels of a job together fills all of the GPU's hardware queues, potentially leaving resources idle while other jobs are queued. In contrast, a well-informed dispatcher (Paella) can increase sustainable throughput while maintaining good tail latency.

per SM, and 32 hardware queues. We craft a synthetic workload where each job has 8 kernels, each with a block of 128 threads, 9 registers, and no shared memory. Each kernel executes for ~300 µs, leading to potential concurrency of up to 176 independent kernels.

Figure 2 shows the p99 latency experienced by the workload as load increases for both the baseline and Paella. Everything aside from the timing of kernel submission is identical. Because of HoL blocking, in the worst case, the naïve submission method will fill up the 32 queues with dependent kernels and use only 32/176 = 18% of the GPU. In contrast, Paella, with accurate and timely information about when kernels start and stop, can optimally interleave the work, maximizing device usage and providing 2.2× better goodput.

Ignorance of application metrics. Finally, even when the hardware scheduler can keep the GPU occupied, today's schedulers provide an overly simplistic interface that ignores important application-level objectives. For example, the ideal schedule of Figure 1 cannot be achieved using any supported submission ordering. This is in addition to other classic problems with strict FIFO, e.g., a lack of fairness, ignorance of deadlines, missing job-level preemption capabilities, etc.

Because all scheduling decisions are handled entirely in the runtime/hardware, the schedulers of existing model-serving frameworks have no control over deficiencies at the GPU runtime/hardware layer. Worse, optimal scheduling policies vary between microarchitectures, making good kernel submission ordering a moving target.

2.2 Framework Overheads

The remainder of the request processing pipeline includes many other bottlenecks to end-to-end performance. Typical overheads, incurred on every inference request, range in the hundreds-of-µs to millisecond scale and stem from tasks like serialization/deserialization, batching, and—unfortunately—service-level scheduling and queue management, all of which apply even when the network latency is zero.

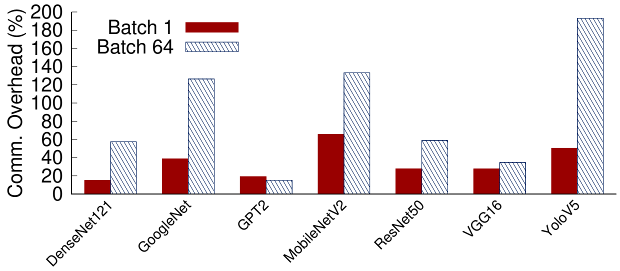

Figure 3. Average overhead of a single batch of requests to an NVIDIA Triton [52] server. The overhead is the end-to-end client latency minus the CUDA kernel executions and memory copies over the latter value. For some models, the serving platform's latency can double the execution time.

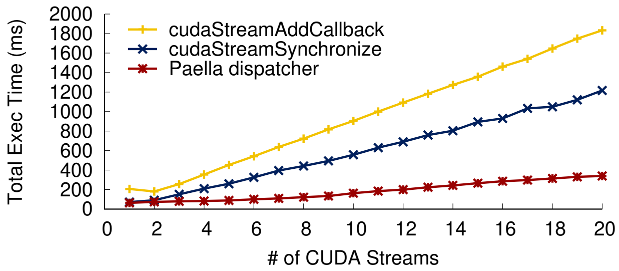

Figure 4. Total time to execute empty kernels under different synchronization methods. Each stream submits 1000 empty kernels that are embarrassingly parallelizable.

To quantify the total effect, we set an NVIDIA Triton inference server [52] and, on the same machine, issue inference queries for several models that cover a wide range of sizes (from 38 to 2,499 nodes in the computation graphs). Results are taken after the average latency has converged. For larger batch sizes, we submit the entire batch immediately so as to elide the configurable overhead of dynamic batching; as such, our results can be seen as a lower bound on overheads. Figure 3 shows the percentage of the total single-request latency that is attributable to the serving platform (*i.e.*, everything except the CUDA memory copies and operator executions)[2]

For a single request in isolation, the serving platform's overhead can be up to 66% of the total execution time. While pipelining can fill some (but not all) of this overhead with other useful work, it does not decrease the time to a useful result. Similarly, optimizations like amortizing overheads over batches of requests may mitigate the effects on throughput but not latency. Dynamic batching, in particular, fails to meet the stringent latency requirements of real-time inference [16, 20, 55] as not only is the overhead typically higher due to larger serialization overhead on the critical path (see Figure 3), the platform must also wait for either sufficient requests to arrive or a timeout.

[2] Overheads include serializing the inputs, sending the request through gRPC, deserializing it at the platform, and with batching, forming a batch input. The result follows a similar pipeline.

These overheads can compound when the GPU is under heavy contention. Figure 4 shows the total execution time of multiple CUDA streams under different synchronization methods. The kernels have no dependencies or memory usage, so synchronization is the primary source of overhead. As a contrast, we also show the overheads of Paella dispatching methods, as described in Section 5.

3 Design Overview

This paper presents a model serving system, Paella, that introduces at least two innovations:

1. A compiler-library-scheduler co-design that abstracts the GPU task submission pipeline, enabling the complete bypass of the GPU hardware scheduler and opening the door for full, extensible, and portable control over GPU scheduling decisions.
2. A service architecture and hybrid interrupt/polling protocol for request submission and result retrieval that, together with the above instrumentation, simultaneously minimizes latency and resource overheads.

For (1), Paella uses lightweight, automated compiler instrumentation to quickly and efficiently track the ground truth of what is scheduled and where. Using this fine-grained information about GPU utilization, the Paella dispatcher can hold kernels until it can guarantee that they can be placed soon/immediately, thereby avoiding the GPU scheduler entirely. Instead, it is free to make careful, precise scheduling decisions based on configurable end-to-end metrics—a benefit that will persist even if GPU vendors introduce improved GPU schedulers in future device generations (as no single scheduling policy is optimal for all workloads [68]).

Supporting the instrumentation and dispatcher is (2), which includes a series of optimized communication channels that facilitate low-latency and low-overhead coordination between all of the components of the system: clients, RPC handlers, the dispatcher, and instrumented kernels. In fact, we show that Paella can reduce serialization and scheduler overheads by more than a factor of 3, all with only a single CPU core for the Paella scheduler and dispatcher. The design of these channels are inspired by recent single-address-space unikernels designed for low-latency networking [10, 38, 59], but with each channel specialized for its particular task.

System components. Paella consists of three components:

- A *compiler*, built on TVM [15], that instruments kernels to expose, at runtime, detailed information about the occupancy and utilization of the GPU's SMs.
- A *client library and communication protocol* that issues requests and responses to the Paella service using a hybrid system of inter-process communication that incurs minimal latency overheads.
- The *Paella dispatcher*, which accepts requests and monitors GPU resource usage in order to make fine-grained scheduling decisions. Careful planning allows Paella to elide most of the GPU block scheduler's responsibilities and make them software-defined so that scheduling decisions can incorporate higher-level information.

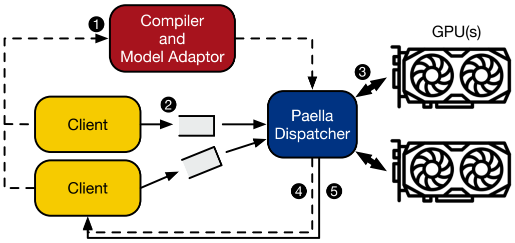

Figure 5. Paella system design. Solid arrows connecting components indicate operations on the critical path of inference. Dotted lines are not on the critical path.

We describe each of the above components in the following three sections. Note that, for simplicity, we primarily focus on local inference tasks where latency overheads are most apparent but also describe how users achieve remote inference in Section 5.

Workflow (Figure 5). Users begin (❶) by compiling their models using Paella's modified compiler. In addition to typical compiler tasks, Paella instruments users' kernels with monitoring code that exports information about where and when the kernel was placed. Users submit the compiled library along with an *adaptor*-style class that properly loads and calls into the library with a provided input/output buffer; together, these two components form an entry in the library of models that are launchable by the Paella dispatcher.

Local clients (or RPC servers) submit inference requests (❷) to Paella by writing raw input vectors to a shared memory queue; the dispatcher uses ground-truth knowledge of the current occupancy and utilization of the GPU(s) to schedule and deploy all of the constituent kernels of each client inference request (❸) without the interference of hardware and runtime queues/schedulers. Before the final kernel of the job completes, the dispatcher will inform the client application of the impending job completion using a simple IPC socket (❹). Only after the client receives the interrupt-based notification will it begin to poll (❺) for results. Shared memory used by the client to receive the result is freed asynchronously.

4 Transforming CUDA Kernels and Jobs

At its core, Paella enables software-defined GPU scheduling by modifying inference jobs to delay their release to the CUDA runtime. The goal is to ensure that the GPU is only provided enough work for full utilization and no more—the rest are held by the Paella dispatcher for on-time release. Paella accomplishes this through transparent transformations of the device and host code of existing CUDA jobs.

Resource	Computation
Blocks	$\lvert SM \rvert$
Threads	$\sum_{i \in SM} \mathtt{Db}_i$
Registers	$\sum_{i \in SM} \mathtt{Db}_i$ * `regs_per_thd`(i)
Shared memory	$\sum_{i \in SM} \mathtt{Ns}_i$

Table 1. Per-SM physical limits for GPUs and how Paella computes their usage. *SM* is the set of all blocks resident on an SM, as determined by runtime tracking; ⋘`Dg,Db,Ns`⋙ is the execution configuration of the kernel; and `regs_per_thd` retrieves the (post-compilation) register requirements.

4.1 Device-side Resource Tracking

When building the kernels of the model, the Paella compiler inserts instrumentation to expose, at a fine granularity, the effects of the GPU hardware scheduler, *i.e.*, the exact set of resources used at a given moment in time. The applicable set of (per-SM) resources is shown in Table 1. Their usage determines whether additional kernels can be scheduled and are used by Paella to saturate the GPU without incurring unnecessary queuing.

Execution configuration information. As mentioned in Section 2.1, the resource usage of each kernel is static. In particular, *before* the launch of the kernel Paella can learn:

- *Grid size:* The number of blocks to be scheduled.
- *Block size:* The number of threads per block.
- *Shared memory:* The amount of shmem per thread block.
- *Register count:* The number of utilized registers per thread.

In the common case, the first three are extracted directly from the execution configuration included in the kernel launch command. The fourth can be extracted at compile-time and included as part of the model metadata. In the case of runtime compilation, *e.g.*, using NVRTC [3] or another JIT compiler, Paella can extract the statistics after the PTX [4] module has been loaded by querying the resource attributes of the `CUmodule`'s constituent functions.

Device-side instrumentation. While information about a kernel's resource requirements can be gleaned before the kernel is even launched, information about which thread blocks have been scheduled and the SMs on which they were scheduled is not available before the blocks are placed by the hardware scheduler. Further, the black-box nature of the GPU's block scheduler means that not only can we not predict the blocks' placement *a priori* but we also cannot reliably determine them *a posteriori*, at least not from the scheduler itself. Instead, Paella tracks these metrics with the help of source-level instrumentation and fast runtime communication to both the dispatcher and the client of the inference request. Specifically, the Paella compiler will automatically instrument kernels to export the following from every running block:

```
__global__ void
myKernel(..., notifQ, kernId) {
  if(threadIdx.x == 0) {
    startCount = atomicInc(&(notifQ.startCount)) + 1;
    if(startCount % 16 == 0 startCount == TOTAL_BLOCKS) {
      asm("mov.u32 %0, %smid;" : "=r"(smId));
      index = atomicInc(&(notifQ.tail), MAX_SIZE);
      notifQ.data[index] = {PLACEMENT, smId, kernId};
  }}
  // computation
  __syncthreads()
  if(threadIdx.x == 0) {
    endCount = atomicInc(&(notifQ.endCount)) + 1;
    if(endCount % 16 == 0 endCount == TOTAL_BLOCKS) {
      index = atomicInc(&(notifQ.tail), MAX_SIZE);
      notifQ.data[index] = {COMPLETION, smId, kernId};
}}}
```

Figure 6. Simplified code of a device-side CUDA function that has been instrumented to export information about scheduling decisions. `notifQ` is the device→host channel.

- *Block start:* An indication that a particular block has successfully been scheduled to an SM.
- *SM identifier:* The ID of the SM to which this block has been placed. Written with the block-start notification.
- *Kernel identifier:* A unique ID that helps the dispatcher to distinguish between different executions of the same kernel, which may have different resource requirements, *e.g.*, due to JIT compilation.
- *Block end:* An indication that a particular block has successfully completed, that its outputs are written and that its resources will soon be freed.

Figure 6 shows an abbreviated version of an instrumented device-side function. Two additional function parameters are required. The first is a host-device shared-memory queue on which *notifications* of thread block placement/completion are posted to the host. The second is the unique kernel ID, which is generated by the dispatcher at execution time and included when writing to the notification queue. The kernel writes to the queue twice: once at the beginning and once at the end of the kernel. For both, a single designated thread from each block is responsible for generating notifications to the Paella dispatcher to inform it that the block was either just placed or just completed. See Section 5.2 for details on the queues' operation and how they are written/read efficiently.

Note that the compiler transformations are uniform across all kernels, regardless of their content. As such, the transformations are simple, and Paella can use any existing TVM model implementation with *no* user modifications.

4.2 Host-side CUDA Emulation

On the host side, Paella intercepts all CUDA calls and emulates the scheduling responsibilities of the CUDA runtime and GPU. It does so by wrapping all relevant CUDA library functions in a stub and automatically replacing calls in the

```
Function kernelLaunch(Dg, Db, Ns, S, ...):
    if S = 0 and waitlist[jobID] has blocking streams then
        Add launch to waitlist as inactive
    else if waitlist[jobID] already has kernels for S or (S is blocking and waitlist[jobId] has stream 0 kernels) then
        Add launch to waitlist as inactive
    else
        Add launch to waitlist as active
Function deviceSynchronize():
    while waitlist[jobId] has any kernels do
        coroutine_context.yield()
```

Figure 7. Pseudocode for a subset of Paella wrapper functions. The functions intercept CUDA stream operations so that they may be handled on Paella's timeline.

user's code with Paella versions. Whenever a kernel or memory copy would have been submitted to the CUDA runtime, Paella instead adds them to a *waitlist* until the GPU is ready to handle them—all without user intervention.

Kernel waitlists. Per-job waitlists, maintained by the Paella dispatcher, replace the functionality of CUDA streams and hardware queues. The waitlists track the set of kernels that are currently schedulable (*active*) and the set that are dependent on the completion of other operations (*inactive*). The active/inactive designation follows CUDA stream semantics [7, 53], *e.g.*, only the first incomplete kernel of each stream is 'active,' the default stream is serialized compared to all other streams unless otherwise specified, etc.

Coroutines. Given a set of user-defined models/adaptors and incoming requests, a straightforward implementation might execute each request in its own thread and yield execution to other threads only when encountering a blocking call or forced preemption. While this retains the interface and abstractions of a traditional CUDA program, it presents issues with CPU utilization and context-switching overheads arising from the potentially large number of resulting threads.

To maintain a traditional CUDA programming abstraction while bounding the number of active threads, Paella implements cooperative multitasking via Boost coroutines, a stackful coroutine library. When users call a CUDA function that is synchronous with respect to the host, the wrapped version will yield the coroutine context; otherwise, it will continue executing. All of this logic is hidden inside the wrapped CUDA functions. Figure 7 illustrates two representative cases. With this architecture, Paella only needs a single thread; however, it can be parallelized by sharding jobs across threads.

Note that the use of coroutines means that users should not modify common objects or memory addresses. In addition, for performance, users should not spin up threads of their own (CPU or GPU), jobs should not have high execution times, and programs should not contain non-CUDA blocking calls; programs that use these features will interfere with Paella's fine-grained control over scheduling/dispatching and should be restructured. As job definitions must be preloaded before they are used, we anticipate that these controls can be enforced by the service owner out-of-band.

```
class MyJob : public PaellaJob {
public:
    void init() override {
        mod_ = LoadFromFile("model.so");
        gmod_ = mod_.GetFunction("default")(CTX_GPU);
        run_ = gmod_.GetFunction("run");
        set_input_ = gmod_.GetFunction("set_input");
        get_output_ = gmod_.GetFunction("get_output"); }
    void run(const size_t len, void* io_ptr) override {
        input = io_ptr; output = io_ptr + INPUT_SIZE;
        set_input_(0, input);
        run_();
        get_output_(0, output); }
private:
    Module mod_, gmod_;
    PackedFunc run_, set_input_, get_output_; }
```

Figure 8. An example job definition in Paella. `init()` loads a compiled TVM library; `run()` takes a shared-memory input/output buffer from the client and executes the request.

'Almost finished' notification. Finally, to support the client operation of Section 5.3, Paella adds an annotation to the job that indicates that the job and its outputs will be ready soon. In our current implementation, this notification is triggered either before the final device-to-host memory copy or, in the case of a pinned output, before the last kernel launch. The goal of the annotation is to wake the client with enough time to catch the finished notification without wasting unnecessary cycles. We note that it may be possible to predict lower-bound execution times for most ML kernels [28] and, thus, conservatively predict the wake-up timer, but we leave the design of such techniques to future work.

5 The Paella Dispatcher

The Paella dispatcher runs on a dedicated CPU core and is responsible for receiving inference requests, dispatching its kernels, and managing communication between the client and GPU. To ensure the lowest possible latency at each stage, Paella takes inspiration from recent work on low-latency single-address-space library OSes [10, 38, 59] by leveraging shared-memory queues for zero-copy, kernel-free communication. To that end, each client obtains a shared memory region when setting up the connection to the dispatcher.

5.1 The Client → Paella Channel

Users add new job definitions to the Paella service by submitting a compiled shared library and an adaptor class that assists in executing it. Figure 8 shows an example adaptor. Afterward, clients submit requests to Paella as follows:

```
req_id = paella.predict('model_name', len, io_ptr,
                        options);
```

model_name refers to a previously instrumented and loaded model; io_ptr is a pointer to a shared-memory buffer of length len that will be passed to the job's run() function (containing, for instance, the input and output tensors); and options specifies any extended functionality of the Paella service (*e.g.*, model variants). The return value, req_id, differentiates between results from different jobs (see Section 5.3).

This interface avoids the need to marshal or unmarshal data—a primary source of overhead in many of today's serving systems. Buffer overflows can be prevented by verifying that [io_ptr, io_ptr + len) fits within the allocated region.

On the other end of the connection, the dispatcher polls each shared-memory queue for incoming client inference requests in a round-robin fashion. The client job is executed immediately, though its constituent kernels may be delayed and/or interleaved with kernels from other jobs according to the scheduling policies of Section 6.

Remote inference. Paella handles remote inference with a straightforward extension to the above design. Specifically, it establishes a local client that acts as an RPC server for remote requests. The local client transparently forwards messages between the remote client and RPC server, and both ends leverage kernel-bypass techniques like eRPC [39]. A more optimized design may be possible, but a full exploration of low-latency remote inference is out of the scope of this paper.

5.2 The Paella ↔ GPU Channel

Dispatching kernels. The Paella dispatcher and the GPU coordinate closely as well. In the Paella → GPU direction, Paella submits *waitlist*ed GPU operations (*e.g.*, memory copies in-/out of the GPU and execution of a sequence of kernels) when the scheduling algorithm deems it appropriate. When that time comes, Paella launches kernels directly using the kernels' original execution configuration and parameters, with two notable modifications: (1) the additional instrumentation-related kernel parameters introduced in Section 4.1 and (2) a carefully replaced stream identifier.

For the second, as we saw in Section 2.1, while recent GPUs attempt to ensure that streams are asynchronous with respect to one other, false dependencies and HoL blocking can still arise if there are more streams than hardware queues. Because of this, Paella also overrides the cudaStreamCreate function to return a virtual stream, which will be replaced at kernel launch time with an available CUDA stream.

Polling the instrumented statistics. During execution, the dispatcher will continually poll the notifQ on the GPU to read the instrumented statistics of Figure 6 and to track, at fine granularity, the current occupancy of the GPU. Like the client-to-Paella channel, notifQ is unidirectional and implemented using shared memory (specifically, pinned memory and unified virtual addressing). It contains two types of events, *placements* and *completions*, which are each accompanied by the kernels' unique ID and SM ID and are used to update the current GPU resource utilization. A third *invalid* type indicates that the entry is stale or inconsistent—the dispatcher sets an entry's status to *invalid* after reading it and the kernels set a valid type as part of their notifQ write.

We note that notifQ is a circular buffer that does not check for overruns by the device-side writer. In general, this can lead to data loss, but in Paella, the demand on the queue is capped by the number of outstanding blocks. Paella can, thus, enforce flow control by delaying kernel dispatches.

Ensuring efficiency. Given the addition of code to every kernel and the centrality of polling to the critical path of the Paella dispatcher, we find that heavy optimization of the instrumentation and its communication is essential for good performance. Paella includes several such optimizations.

One such optimization is to implement notifications as a lock-free queue in which notifications are in the form of a single write to a 64-bit integer: 8 b for the type, 8 b for the SM ID, and 32 b for the unique kernel ID. Because the write fits within a primitive type, filling the actual queue element (and resetting it to invalid) is guaranteed to be atomic on most GPU architectures [17]. Further, Paella uses a single notifQ for each dispatcher thread, reducing the locations the dispatcher polled. The tradeoff is the need for a single atomic increment per enqueue, but our tests demonstrated that this overhead was minimal (see Figure 4).

Finally, to reduce the size of the queue, we batch start and end notifications such that each notification signals the start/end of a group of up to 16 thread blocks. For models with high parallelism, this significantly reduces overheads.

5.3 The GPU → Client Channel

When outputs are available, clients retrieve results with:

```
req_id = paella.readResult(options);
```

The function returns the first available completion's ID. options modifies the behavior of the function, with the most important option being the NONBLOCK flag. Without the flag, the function will wait for at least one indication that a job has been completed and that its output is ready to read directly from shared memory (again, without marshaling overheads). Adding the flag, it may return the negative code EAGAIN.

Internally, the function relies on a similar shared-memory technique as the other two communication channels in this section. Unfortunately, applied naïvely, this would require each blocking client to maintain a continuous polling thread to achieve optimal latency—increasing the number of utilized CPU cores proportionally to the number of active clients or necessitating a tradeoff between latency and CPU utilization.

Instead, in order to achieve the best of both worlds, blocking read calls utilize a hybrid mechanism for inter-process communication. Initially, clients listen on a Unix socket for the wake-up signal introduced in Section 4.2. Only after will the client begin polling for completion events. This is similar to Linux's NAPI interrupt-mitigation techniques, except that

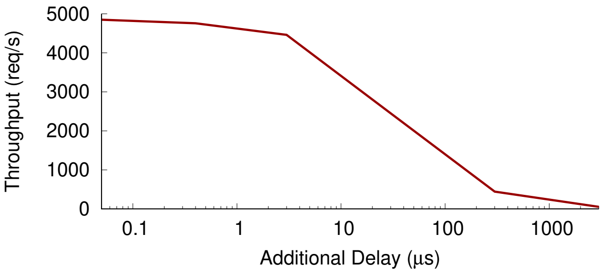

Figure 9. The impact of additional scheduling complexity on Paella's performance. Varying amounts of synthetic delay are added to its default scheduling algorithm. The late-binding nature of the Paella dispatcher necessitates efficiency.

the switch to polling is predictable in our case.

6 Scheduling Strategy

Leveraging the efficient communication channels between instrumented kernels and the serving platform, Paella lifts scheduling out of the GPU device and into the Paella dispatcher, enabling flexible and extensible scheduling approaches that are not possible today, *e.g.*, the ability to schedule individual kernels based on remaining job execution time.

The space of possible algorithms is unbounded and beyond the scope of this paper, but we note that not all algorithms are suitable for low-latency serving. In particular, because Paella tries its best to minimize the occupancy of the GPU hardware queues, any computational overhead in the dispatcher has the potential to impact critical-path latency. This effect is amplified because Paella makes scheduling decisions on a per-block or per-block-group granularity (as opposed to a per-kernel granularity). Figure 9 illustrates the effect with synthetic injected delay in the scheduling algorithm. Paella's default scheduler addresses this challenge with a simple but effective algorithm that considers the following:

(1) Remaining time. Paella implements a scheduling strategy based on shortest-remaining-processing-time (SRPT). Paella does not know the precise running time of each kernel (or even which kernels may execute in a given job) *a priori* due to non-deterministic contention and application control flows; however, it can estimate it as follows.

When submitting the model, Paella will first run a series of simple profiling runs of the job, tracking the average execution count and time of each kernel (distinguished by their locations in the shared library); these profiles can be further refined online. As a heuristic, we assume independence and compute the remaining time as:

$$remaining = \sum_{i \in K} \max(0, \overline{C_i} - c_i) \cdot \overline{T_i})$$

where K is the set of unique kernels in the job, T_i is the execution time of job i, C_i is the execution count of job i, and c_i is the number of times the kernel has run thus far. $\overline{x}$ is the average of all previous observations of x. Although variation exists (e.g., due to contention between concurrent kernels), empirically, this running average is good enough for sorting the jobs according to their remaining time.

Model	TVM Exec Time	Size
ResNet-18 [30]	1.58 ms	75 MB
MobileNetV2 [33]	1.67 ms	14 MB
ResNet-34 [30]	2.55 ms	144 MB
Squeezenet1.1 [36]	4.79 ms	5.2 MB
ResNet-50 [30]	5.76 ms	124 MB
DenseNet [35]	6.08 ms	41 MB
GoogleNet [66]	7.86 ms	28 MB
InceptionV3 [65]	31.2 ms	93 MB

Table 2. A list of models used in our evaluation benchmarks. TVM Exec Time is the time to execute the model directly in C++, without any serving infrastructure.

(2) Fairness. Paella also borrows inspiration from prior work in its use of deficit counters [27, 64] to efficiently and effectively constrain the SRPT-related unfairness between different users. Conceptually, when a kernel, i, is scheduled, its *deficit* decreases by $(1 - \frac{1}{\#\ users})$ while the *deficit* of all other users increases by $\frac{1}{\#\ users}$. If any user's deficit exceeds a user-defined threshold, the oldest job of the client with the highest deficit counter is chosen; otherwise, Paella uses (1).

(3) Full utilization. While Paella receives fine-grained information about current GPU utilization, in practice, there is some communication delay between the GPU and the dispatcher. To ensure that the GPU is always saturated, Paella will submit a configurable B blocks beyond its full utilization.

Efficient implementation and overall strategy. Paella tracks jobs' (1) and (2) with a red-black tree for each. When it detects the placement of sufficient blocks (using the mechanisms of Section 5.2) to reduce the hardware queue below B, Paella will dispatch a kernel of the job with either the lowest *remaining* time (if under the unfairness threshold) or the highest *deficit* (otherwise). It will repeat this until it conservatively estimates sufficient outstanding kernels to fully utilize at least one of the GPU's resources (+B).

Dispatching a kernel removes its job from both trees; the job may be reinserted once another kernel is ready to execute. For the *deficit* tree, reinsertion decreases the deficit counter by 1 and shifts the original threshold by $\frac{1}{\#\ jobs}$. This shift converts an $O(n)$ deficit update into an $O(1)$ operation, though an $O(n)$ reset is required on `double` underflow.

7 Evaluation

Implementation. We implemented a prototype of Paella in C++. Paella is built on top of existing tools such as TVM and Boost co-routines. For the client library, dispatcher, and compiler modifications, Paella contains 4,221 lines of additional C++ and CUDA code. The Paella dispatcher runs in Linux on a designated core with real-time scheduling priority. In order to use the system, clients adding a new model write only the TVM model definition and the job adaptor of Section 4.2.

System	Key	Interface	Dispatch	Sched.
Single CUDA Stream	CUDA-SS	Direct	job	FIFO
Multiple CUDA Streams	CUDA-MS	Direct	job	CUDA
MPS	MPS	Direct	job	MPS
Clockwork [28]	Clockwork	Boost Asio	job	FIFO
Triton [52]	Triton	gRPC	job	CUDA
Paella w/ Single Stream	Paella-SS	mem channels	job	FIFO
Paella w/ Multiple Streams	Paella-MS-jbj	mem channels	job	CUDA
Paella w/ Multiple Streams	Paella-MS-kbk	mem channels	kernel	CUDA
Paella	**Paella**	**mem channels**	**kernel**	**§6**
Paella w/ Shortest-job-first	Paella-SJF	mem channels	kernel	SJF
Paella w/ Round-robin	Paella-RR	mem channels	kernel	RR

Table 3. Compared systems and variants. Triton's scheduler has additional features (*e.g.*, cross-GPU or inter-backend), but they are mostly orthogonal for our configurations.

Methodology. We evaluated our system on a server with a 2.2 GHz Xeon Silver 4114 CPU, 64 GB RAM, and an NVIDIA Tesla T4 GPU. The server runs Linux 4.15.0, CUDA 11.7, and uses a modified version of TVM v0.10.0 as described above. We also evaluated our system on a Tesla P100 but omitted those results as the trends were identical.

The models we used are listed in Table 2 and are from repositories of pre-trained models such as the ONNX Model Zoo [5]. Unless specified otherwise, the request inter-arrival pattern follows a lognormal distribution with either $\sigma = 2$ (bursty) or $\sigma = 1.5$ (less bursty) and varying mean, μ.

In all cases, we waited for results to stabilize before gathering measurements for any of the below experiments. This eliminates transient start-up costs associated with Paella provisioning and TVM runtime compilation.

Baselines. As baselines, we compare against several systems, all listed in Table 3. These include scenarios without any serving system, where one or more processes directly submit kernels to the CUDA runtime. They also include three ablations of Paella that retain Paella's frontend and dispatcher architecture but with trivial scheduling (one where each model is dispatched to a single stream and a pair where they are dispatched immediately to a unique CUDA stream, differing in their dispatch granularity), as well as versions of Paella with other simpler scheduling algorithms.

As a reference state-of-the-art model serving system, we ran NVIDIA's Triton v23.03. To ensure a fair comparison, we used TVM as a backend. As this interaction is not supported in Triton, we needed to implement a custom wrapper for TVM graphs in a TensorFlow operation, which can then be exported as a TensorFlow SavedModel. We also compared baseline latency to Clockwork, but omitted them from larger experiments as they are designed for predictability rather than throughput (*e.g.*, executing only one model at a time).

7.1 Paella Improves Inference Latency

We first evaluate whether and by how much Paella improves the end-to-end latency of inference. Figure 10 shows a breakdown of the overheads incurred by Paella and the serving systems in Table 3 for a single MobileNet request. Paella exhibits comparable latency in scheduling and framework overheads to Triton (which uses a simple FIFO scheduler) but less than Clockwork (which is split over separate controller and worker processes and which expends effort to offer high predictability); even compared to FIFO versions of Paella, the latency is minimal. Its communication latency (encompassing serialization and syscall overheads) is also lower than Clockwork and significantly faster than Triton, which is based on gRPC.

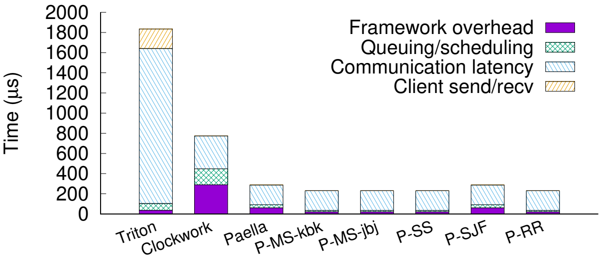

Figure 10. A comparison of the overheads of Paella and its (Multi/Single Stream) ablations versus Triton and Clockwork. All network latency and CUDA execution time are excluded.

Paella under variable load. Under load, Paella increases its advantage. Figure 11 shows the results for a uniform mix of inference models (listed in Table 2), two levels of burstiness, and the baselines in Table 3[3]. We evaluate the p99 latency and average throughput of requests. For all systems, we increased the offered load until the systems reached saturation.

Compared to Triton, Paella maintains a lower latency baseline and sustains 1–3 orders of magnitude higher throughput before saturation. For these workloads and metrics, the performance benefits stem primarily from Paella's efficient architecture, designed from first principles for minimal overhead and ultra-low latency. As Paella's default scheduling policy is based on SRPT, its benefits to throughput (i.e., the difference between Paella and the Paella-MSes) are modest.

Short vs. long jobs. Where Paella truly shines is in the benefits of its scheduling improvements to different job sizes. For this experiment, we pit ResNet-18 against InceptionV3, which differ significantly in their size. Workloads are based on the same lognormal distributions as above, and the ratio of smaller to larger jobs is inversely proportional to their size. Figure 12 shows that p99 ResNet-18 baseline latency improves by up to 3× compared to CUDA-MS.

7.2 Paella Provides Tunable Scheduling

Using Paella, providers can implement and tune scheduling policies that are independent of the GPU's scheduler. For example, the algorithm presented in Section 6 implements an

[3] We omit MPS in this experiment as it does not support more than seven processes, and any aggregation would introduce significant confounding effects. Instead, a comparison of Paella and MPS can be found in Figure 12.

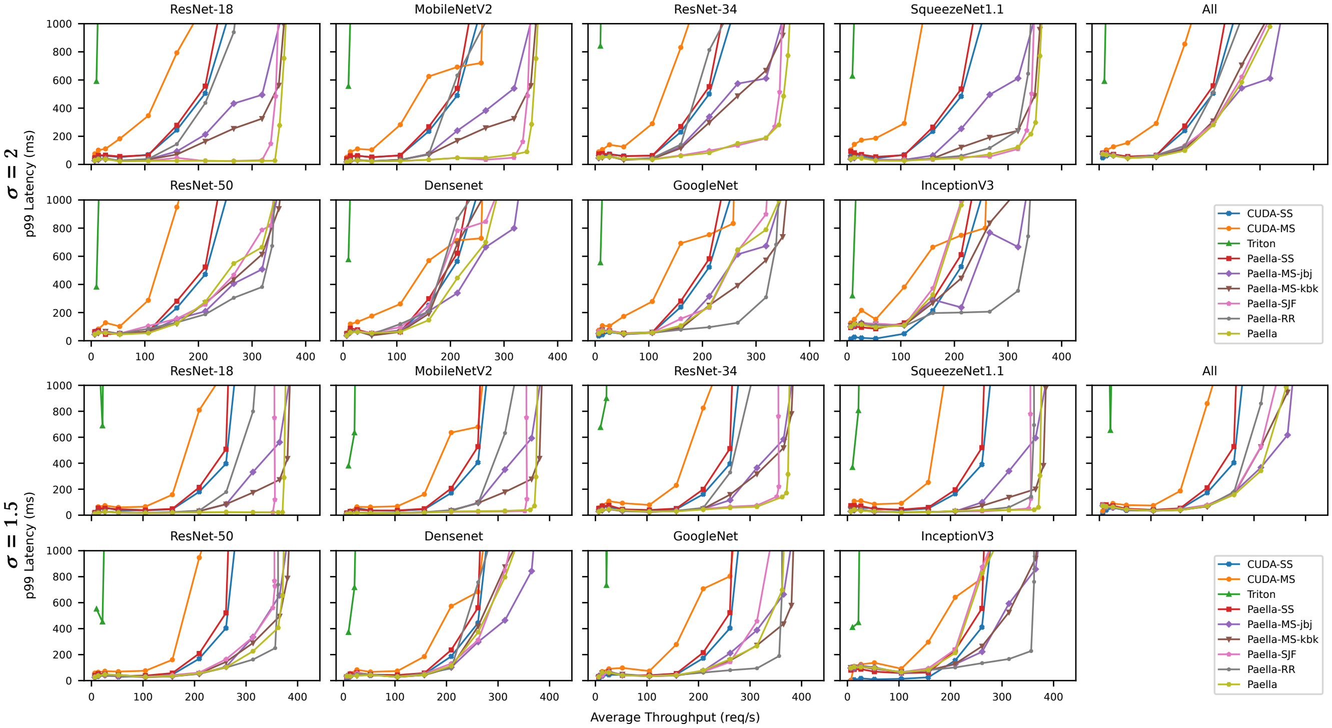

Figure 11. Average throughput versus p99 latency for a mix of inference models. Results are for two lognormal distributions. Compared to other systems, Paella maintains a lower latency baseline and sustains higher throughput before saturation.

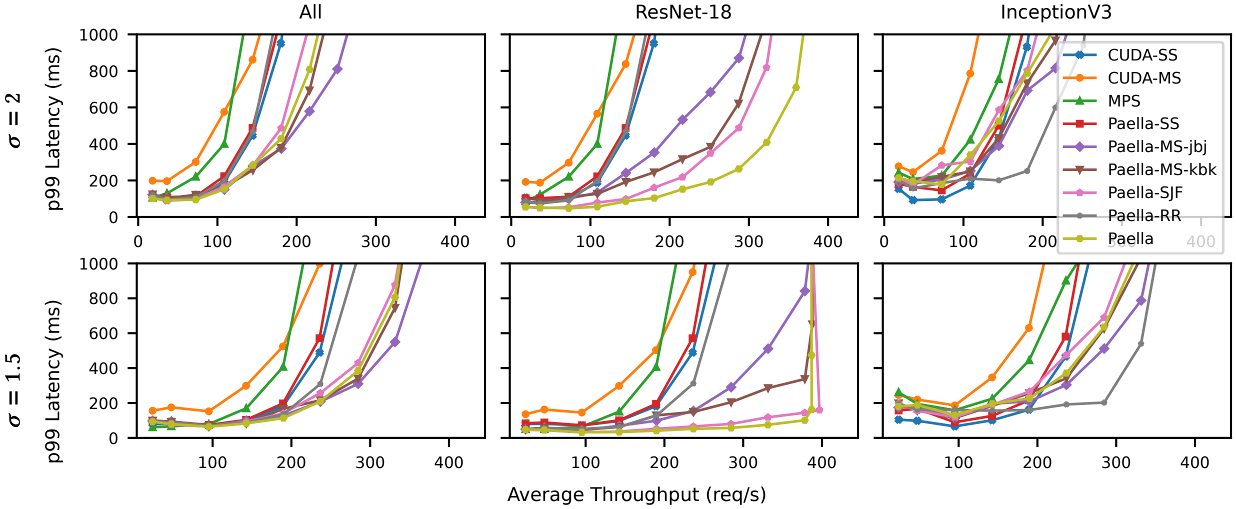

Figure 12. Average throughput versus p99 latency for workloads with two models of differing sizes and two workload distribution parameters. For both workload parameters, short jobs can benefit substantially from Paella's SRPT-like algorithm.

SRPT-like strategy in which shorter and partially completed jobs are prioritized over longer ones to achieve lower overall latency. To counteract SRPT's well-known issues with the potential starvation of long jobs, it adds a fairness threshold.

Figure 13 demonstrates the effect of this fairness mechanism and its threshold using a workload with two job types, with one having 5× as many kernels to run. As we decrease the fairness threshold, the deficit counter of the long jobs triggers immediate scheduling earlier. As the threshold approaches zero, the system emulates Paella-SS.

In addition to FIFO and Paella's default algorithm (a mix of SRPT and deficit-based Priority Scheduling), we implemented algorithms like shortest-job-first (SJF) and round-robin (RR) to demonstrate the flexibility of Paella further. The effects of these alternatives follow conventional wisdom. For example, Figures 11 and 12 show that RR gives the larger jobs a more fair share of resources, resulting in a different

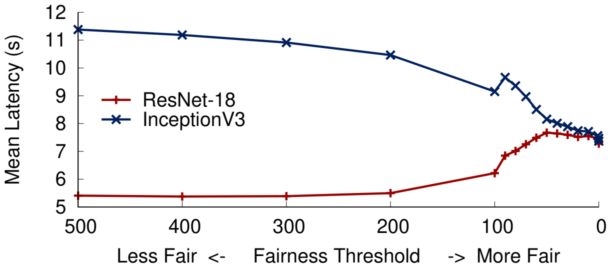

Figure 13. Mean latency for short (ResNet-18) and long (InceptionV3) jobs versus the fairness threshold of Section 6.

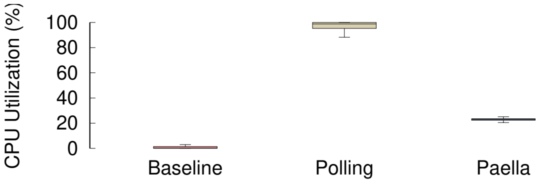

Figure 14. The CPU utilization of a client submitting many small jobs using various communication protocols. The baseline is a simple Unix socket IPC.

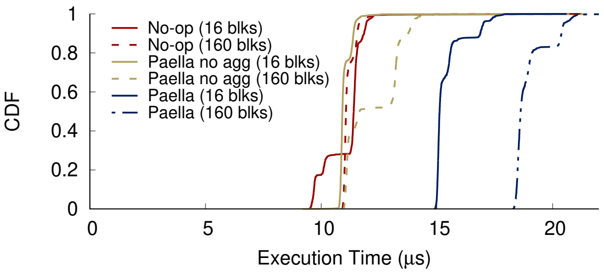

Figure 15. The overheads of Paella's kernel instrumentation with and without block aggregation. Execution time encompasses the time on the host between the initiation of the kernel launch and the return from a synchronization barrier.

trade-off compared to our default algorithm or SJF.

7.3 Paella Incurs Low Overhead

Host-side overhead. To evaluate the CPU overhead of Paella and the impact of its hybrid inter-process communication mechanism, we evaluate three different implementations: (1) a traditional Unix socket channel to coordinate between the client and dispatcher, (2) unmitigated client polling, and (3) Paella's hybrid scheme. Into each system, we send ∼6,700 requests per second of a small synthetic model and sample client CPU utilization every 5 ms. This experiment represents an upper bound on the client load.

Figure 14 shows the results: the first two mechanisms sit at extremes of CPU utilization, while in Paella, the client's average utilization is around 23%, a number that is dependent on the fraction of time spent on the last operator compared to the full job. Just as important, Paella does not sacrifice any appreciable inference latency compared to polling, while the baseline is, on average, ∼10% slower.

Device-side overhead. Finally, we evaluate the overhead of Paella's kernel instrumentation technique. To stress the system and maximize the potential contention on the notification queue, we test an instrumented empty kernel whose only task is to enqueue placement/completion notifications to the `notifQ`. We compare this kernel against a version that omits the notification aggregation (notifying for every block) as well as one that has not been instrumented.

Figure 15 shows CDFs of the execution times of all three variants and two different kernel grid sizes. The notifications alone add little overhead, even for 160 blocks (for 90-pct: 2.2 µs). Adding a conditional to support aggregation adds additional overhead (for 90-pct: 16 blocks, 5.5 µs; 160 blocks, 6.6 µs); however, the benefits to the dispatcher overhead make the tradeoff worth it. We argue that these overheads are acceptable for the benefits that instrumentation affords.

8 Discussion and Future Work

Interaction with Multi-Instance GPUs and cluster-level scheduling. GPU scheduling in modern data centers can sometimes interact with surrounding mechanisms in complex ways. Within a GPU, NVIDIA's MIG can affect Paella's design by slicing a GPU's resources into several partitions. For known, static partitions, Paella's techniques apply directly; the interaction between GPU scheduling and dynamic repartitioning is a potentially fruitful direction for future work. Across machines/GPUs, cluster-level scheduling decisions can also impact Paella, even though Paella targets lower-level scheduling decisions. For this, we refer users to the rich literature on hierarchical scheduling—Paella enables the direct application of these solutions.

Scaling to larger GPUs. As GPUs continue to grow in SM count and memory capacity, we anticipate that the opportunities for fine-grained multiplexing will also increase, e.g., to support legacy or parameter-efficient models. The more kernel-level concurrency, the more scheduling is needed.

One possible concern is that software scheduling overheads may increase as scheduling becomes more complex; however, Figure 9 demonstrates that Paella still has some headroom before these overheads start to impact performance. Note that Figure 9 uses modest Xeon Silver 4114 CPUs and MNIST, a model with 1000× smaller execution time and size than the smallest model in Table 2, and thus, it represents the worst case. Assuming MobileNet and the same per-job dispatch overheads of Figure 10, job dispatch rates could increase by ∼30× before significantly impacting Paella's throughput.

Dynamic batching. As mentioned in Section 2.2, the dynamic batching used in many of today's serving frameworks is detrimental to critical-path request latency because of its need to wait for additional requests and copy input data into a batched format. We note, however, that at high loads where throughput bottlenecks contribute to latency, the efficiency gains may make batching worth performing. Paella can be extended to detect saturation and batch in these cases, but it currently does not as TVM does not support dynamic batching (because it conflicts with TVM's autotuning optimizations and substantially reduces performance).

GPU vendor support. Paella enables the implementation of scheduling algorithms in software. While GPU hardware schedulers could implement any particular scheduling algorithm more efficiently, a software solution has an advantage in enabling extremely flexible scheduling, which, as mentioned at the end of Section 1, cannot be replaced by any single hardware-implemented scheduling algorithm. We note that a promising direction is to co-design the software scheduler with the GPU hardware. We leave this exploration to future work, but we believe the core of Paella's transparency (visibility into scheduling decisions) is necessary regardless.

Pre-compiled kernels. While Paella requires users to submit their job definitions with the original source code, we note that pre-compiled ML binaries, such as the popular cuDNN library may also be adapted to Paella. In particular, the simplicity of Paella's device code transformations and their independence to application logic means that, in principle, the instrumentation can be added via static binary translation. Unfortunately, the key hurdle is not technical: EULAs for these libraries often disallow reverse engineering, decompilation, or disassembly. Instead, we anticipate that vendors could publish instrumented, compatible versions of library functions for use in the Paella framework.

9 Related Work

Model serving frameworks. Paella builds on prior model serving frameworks [18, 19, 29, 44, 52, 54, 63]. These systems have made significant progress in lowering the latency and raising the throughput of inference. Clipper [19], Cocktail [29], and INFaaS [61], for instance, adaptively select model variants and introduce related optimizations in autoscaling, caching, and batching of their execution. These types of techniques, typically implemented a layer above the actual model execution frameworks, are complementary to Paella. Instead, Paella attempts to address deficiencies in the GPU hardware scheduler, which requires a co-design of CUDA kernels and model serving.

We note that a subset of this work also targets low-latency through better scheduling and dispatching protocols; however, because of the traditional opacity of the GPU scheduler, many assume only a single model is running on the GPU at a given time [8, 28, 34, 47, 72]. Rammer [47], for instance, takes an individual job and, at compile time, determines the optimal intra-job schedule among parallelizable operators within a job. They then use a persistent thread primitive to (like Paella) avoid the GPU scheduler and enforce their optimized schedule. When models are large enough to fully occupy the GPU on their own and scheduler pre-emption of jobs is not required, Rammer may be sufficient; however, a more dynamic solution like Paella is necessary when multiplexing is necessary and inter-job effects matter.

Real-time inference. Also relevant is a large body of work in the real-time community [50]. In fact, it is from this work where much of our understanding of GPU scheduling policies stems [7, 9, 57]. In terms of solutions, however, real-time systems attempt to eliminate as much overhead and uncertainty as possible to ensure that tasks are completed within some specified time constraint (hard or soft). Our work is inspired by these earlier systems [12, 13, 24, 48, 49, 67], but we take a markedly different approach in our treatment of deadlines and/or target use cases. For example, Clockwork [28] is a recent system that leverages the predictability of typical ML kernels in order to provide hard real-time guarantees. In exchange for this predictability, only a single model can run on a given GPU at a time. Paella targets a different point in the design space, opting to maximize GPU occupancy for high throughput and minimize JCT given that constraint.

GPU scheduling. We note that other work has previously examined the problem of GPU scheduling. Many of these systems have proposed modifications to the GPU or drivers themselves in order to improve block/warp scheduling [26, 51, 58]; Paella's approach is mostly orthogonal to these improvements as long as placement is roughly predictable. Others have advocated for alternative programming models [11, 74] or scheduling algorithms for kernel dispatch [32, 40, 47] with the goal of promoting higher GPU utilization. To the best of our knowledge, Paella is the first to leverage thread-block instrumentation to softwareize the scheduling decisions of GPUs.

In-memory channels. Finally, we note that other systems, particularly those that provide kernel bypass for storage and networking, also use in-memory channels to great effect [73]. For example, ReFlex [42] demonstrated managing in-memory channels for clients can scale to thousands of tenants. Paella is complementary and can leverage the optimizations and abstractions introduced by these systems.

10 Conclusion

In this paper, we (1) show that existing serving systems are hamstrung by these components; (2) present a technique to lift scheduling responsibilities out of the GPU hardware and into software through a co-design of the compiler, kernel instrumentation, and an efficient dispatcher; and (3) demonstrate its utility with a useful scheduling algorithm on top of Paella. In addition to software-defined scheduling, Paella's dispatcher includes optimizations such as shared-memory communication channels, co-routine multiprocessing, and optimized placement heuristics to achieve performance benefits for diverse workloads.

Acknowledgments

We gratefully acknowledge our shepherd Lin Zhong and the anonymous SOSP reviewers for all of their help and thoughtful comments. This work was funded in part by Google, Meta, VMWare, and NSF grants CNS-1845749 and CNS-2107147.

References

[1] NVIDIA HyperQ. https://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[2] NVIDIA MPS. https://docs.nvidia.com/deploy/mps/index.html.

[3] NVRTC (Runtime Compilation). https://docs.nvidia.com/cuda/nvrtc/index.html.

[4] Parallel Thread Execution ISA). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[5] Onnx model zoo, 2020. https://github.com/onnx/models.

[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, Berkeley, CA, USA, 2017. USENIX Association.

[7] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, November 2020.

[9] J. Bakita, Nathan Otterness, J. Anderson, and F. D. Smith. Scaling up: The validation of empirically derived scheduling rules on NVIDIA GPUs. In *14th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2018.

[10] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[11] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130, 2018.

[12] A. X. M. Chang and E. Culurciello. Hardware accelerators for recurrent neural networks on fpga. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[13] G. Chen and X. Shen. Free launch: Optimizing gpu dynamic kernel launches through thread reuse. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–419, 2015.

[14] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 191–205, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[16] CN Coelho, A Kuusela, S Li, H Zhuang, T Aarrestad, V Loncar, J Ngadiuba, M Pierini, AA Pol, and S Summers. Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders. *arXiv preprint arXiv:2006.10159*.

[17] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, 2021.

[18] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[20] Martijn de Rooij. Ultra low latency deep neural network inference for gravitational waves interferometer. 2021.

[21] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 693–708, Renton, WA, July 2019. USENIX Association.

[23] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. Fast inference of deep neural networks in fpgas for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[24] Glenn A. Elliott and James H. Anderson. Real-world constraints of gpus in real-time systems. In *Proceedings of the 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications - Volume 02*, RTCSA '11, page 48–54, USA, 2011. IEEE Computer Society.

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[26] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420, 2007.

[27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. OSDI'16, page 81–97, USA, 2016. USENIX Association.

[28] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[29] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, Renton, WA, April 2022. USENIX Association.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[31] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning and Representation Learning*

Workshop, 2015.

[32] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Comput. Surv.*, 50(3), June 2017.

[33] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[34] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[36] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[37] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking, 2019.

[38] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[39] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[40] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, page 2, USA, 2011. USENIX Association.

[41] Charles W. Kazer, João Sedoc, Kelvin K.W. Ng, Vincent Liu, and Lyle H. Ungar. Fast network simulation through approximation or: How blind men can describe elephants. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 141–147, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ~ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Redis Labs and Tensorwerk. Redisai, 2020. https://github.com/RedisAI/RedisAI.

[45] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future, 2016.

[46] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data Mining and Knowledge Discovery*, 6(4):393–423, December 2002.

[47] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[48] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda. Characterizing cuda unified memory (um)-aware mpi designs on modern gpu architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, GPGPU '19, page 43–52, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Michele Martinelli. Poster: Gpu i/o persistent kernel for latency bound systems. In *ACM Symposium on High-Performance Parallel and Distributed Computing*, 2017.

[50] Pınar Muyan-Özçelik and John D. Owens. Methods for multitasking among real-time embedded compute tasks running on the gpu. *Concurrency and Computation: Practice and Experience*, 29(15):e4118, 2017. e4118 cpe.4118.

[51] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 308–317, New York, NY, USA, 2011. Association for Computing Machinery.

[52] NVIDIA. Triton inference server, 2020. https://github.com/triton-inference-server/server.

[53] Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–225, 2020.

[54] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[55] Aaron Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George Driessche, Edward Lockhart, Luis Cobo, Florian Stimberg, et al. Parallel wavenet: Fast high-fidelity speech synthesis. In *International conference on machine learning*, pages 3918–3926. PMLR, 2018.

[56] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364, 2017.

[57] Sreepathi Pai. How the fermi thread block scheduler works (illustrated), Mar 2014. https://cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html.

[58] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 31–44, New York, NY, USA, 2016. Association for Computing Machinery.

[59] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, February 2009.

[61] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[62] A. Shawahna, S. M. Sait, and A. El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[63] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[64] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, page 231–242, New York, NY, USA, 1995. Association for Computing Machinery.

[65] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, Los Alamitos, CA, USA, jun 2016. IEEE Computer Society.

[66] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[67] Concurrent Real-Time Linux Development Team. Real-time performance during cuda. Technical report, Concurrent Real-Time, 11 2010.

[68] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[69] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 123–134, New York, NY, USA, 2013. ACM.

[70] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[71] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association.

[72] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[73] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[74] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2014.