Kitsune: Enabling Dataflow Execution on GPUs with Spatial Pipelines

MICHAEL DAVIES, NVIDIA, Santa Clara, United States
NEAL CRAGO, NVIDIA, Santa Clara, United States
KARTHIKEYAN SANKARALINGAM, NVIDIA, Santa Clara, United States and Computer Sciences, University of Wisconsin-Madison, Madison, USA
STEPHEN KECKLER, NVIDIA Research, Santa Clara, United States

State-of-the-art DL models are growing in size and complexity, with many modern models also increasing in heterogeneity of behavior. GPUs are still the dominant platform for DL applications, relying on a bulk-synchronous execution model which has many drawbacks and is ill-suited for the graph structure of DL applications. Many industry and academic works attempt to overcome these by employing vertical fusion – combining multiple sequential operations into a single kernel – but this approach still fails to realize three untapped opportunities: (1) the fact that many resources on the GPU are idle while only one operator executes due to temporal multiplexing of the SM; (2) lower energy from more intelligent on-chip data-movement which lends to higher performance in a power-provisioned environment. (3) inability to exploit reduction dimensions as a source of parallelism to ease pressure on batch size. This article explores relatively uncharted territory, answering the following key question: *Can modest adjustments to the current GPU architecture enable efficient dataflow execution, thereby circumventing the constraints of vertical fusion without necessitating a clean-slate architecture design.* We develop Kitsune – a set of primitives to construct spatial pipelines which enable dataflow execution on GPUs, and an end-to-end compiler based on PyTorch Dynamo. Across 5 challenge applications, Kitsune can provide up to 2.8× and 2.2× performance improvement as well as up to 99% and 45% off-chip traffic reduction for inference and training, respectively.

CCS Concepts: • **Computer systems organization → Data flow architectures**; • **Computing methodologies → Distributed computing methodologies**; *Artificial intelligence;*

Additional Key Words and Phrases: GPU architecture, spatial pipelining, artificial intelligence

ACM Reference Format:
Michael Davies, Neal Crago, Karthikeyan Sankaralingam, and Stephen Keckler. 2025. Kitsune: Enabling Dataflow Execution on GPUs with Spatial Pipelines. *ACM Trans. Arch. Code Optim.* 22, 4, Article 146 (December 2025), 22 pages. https://doi.org/10.1145/3777466

Authors' Contact Information: Michael Davies (corresponding author), NVIDIA, Santa Clara, California, USA; e-mail: mdavies@nvidia.com; Neal Crago, NVIDIA, Santa Clara, California, USA; e-mail: ncrago@nvidia.com; Karthikeyan Sankaralingam, NVIDIA, Santa Clara, California, United States and Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; e-mail: karus@nvidia.com; Stephen Keckler, NVIDIA Research, Santa Clara, California, USA; e-mail: skeckler@nvidia.com.

ACM 1544-3973/2025/12-ART146
https://doi.org/10.1145/3777466

1 Introduction

Graphics Processing Units (**GPUs**) have become the dominant platform for executing **deep learning** (**DL**) algorithms due to their amenability to matrix-multiplication and other common DL operations. Historically designed for **Single Instruction, Multiple Thread** (**SIMT**) execution with extensive register files, GPUs have evolved significantly. They now boast intricate memory hierarchies, specialized Tensor Cores for **general matrix-multiply** (**GEMM**) computations, and support for atomic memory instructions [36]. Depicted in Figure 1, GPUs (a) employ a relatively simple **bulk-synchronous programming** (**BSP**) model (c), where a set of independent work items for a single operator (commonly implemented as a single kernel) are run to completion followed by a global barrier before the next set is dispatched. **However, the BSP model is a misfit to the directed-acyclic graph structure of DL applications, and hence encounters inefficiencies centered around three key areas:** the inability to exploit on-chip data locality of intermediate data passed between operators due to large memory footprints spilling to DRAM, and idle resources due to limited parallelism or low arithmetic intensity within operators.

Vertical fusion, depicted in Figure 1(c), is an approach for GPUs to amortize kernel launch overheads and improve data locality between operators and thus reduce off-chip memory traffic through fusing multiple operators into a single CUDA "mega kernel", establishing the need for flexibility in GPU execution. Under this paradigm, GPU's execution resources are *temporally* multiplexed between several "fused" operators, interleaving partial executions of each operator, allowing tiles of intermediate data to stay resident on chip for reuse, and removing the need for kernel barriers between fused operators. This multiplexing is depicted in Figure 1(c) by how at a given time, only the TensorCores or SIMT resources are active. This technique has been commercialized in tools such as TensorRT [51] and advanced through academic endeavors like Welder [47], Astitch [64] and others [9, 15, 59, 60, 63]. Despite its effectiveness, vertical fusion leaves three performance opportunities untapped. First, because of temporal multiplexing, the technique does not take advantage of the many idle resources available while one operator is executing. Second, because of how vertically fused operations are structured, spilling large intermediates to DRAM can become unavoidable, incurring a round-trip DRAM latency penalty. Third, it is unable to exploit reduction or hidden dimensions for parallelism to ease the need for large batch-level parallelism.

Many academic and industry approaches (Groq for e.g.,) recognize that dataflow execution (i.e., concurrently executing operators across *space* rather than time) aligns more naturally to the graph structure of DL applications—mitigating the above inefficiencies of BSP and vertical fusion with *clean-slate* architectures [1, 2, 21, 41, 42, 46, 52]. The focus of these efforts is dataflow execution of DL (sub)graph nodes at the single-chip level, while recognizing other aspects of dataflow execution also exist at the system level [3] and within the matrix-engines themselves [5, 6, 45]. **This article explores relatively uncharted territory, answering whether modest adjustments to the dominant GPU architecture and software stack can enable efficient dataflow execution at the chip-level.**

Our key insight is two complementary software-hardware primitives are sufficient to enable dataflow execution on GPUs. They are: (1) a software-only ring queue which facilitates inter-CTA (Cooperative Thread Array) communication by using the L2 cache and global atomics; (2) a modest change to the GPU's grid scheduler to enable it to exploit the heterogeneity of concurrently executing operators. We find that an effective end-to-end compiler can be built that uses these primitives to allow automatic lowering of DL applications to dataflow execution on GPUs, avoiding the need for new IRs or a complex code generation backend. This system, named "Kitsune", addresses the problems of the BSP model: executing more than one operator concurrently in a *spatial pipeline* and passing tiles of intermediate data through on-chip queues increases available parallelism

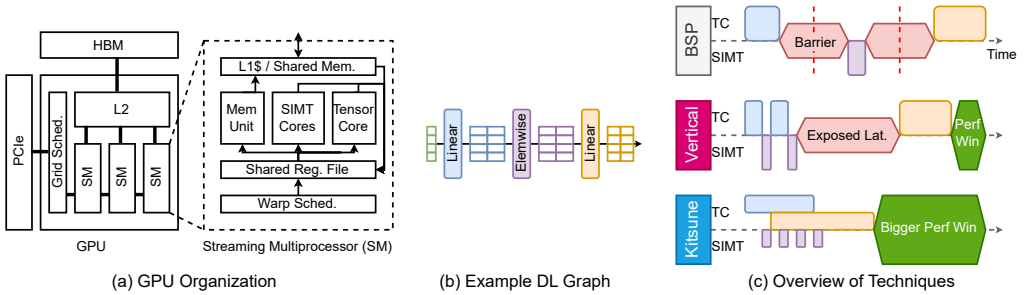

Fig. 1. (a) Overview of GPU organization, (b) example DL graph, and (c) stylized comparison of execution techniques. In (c), TensorCore and SIMT resources of the GPU are depicted separately.

and reduces memory bandwidth pressure. This is depicted in Figure 1(c), where multiple different operators can executed simultaneously across both the SIMT and TensorCore resources on the GPU.

The contributions of this work are as follows.

(1) **A systematic characterization of DL applications that highlights the mismatch between graph behavior and GPU bulk-synchronous execution.**
(2) **A design and analysis of Kitsune's SW/HW primitives needed to construct spatial pipelines and enable synchronous dataflow execution on GPUs.**
(3) **A design and implementation for the Kitsune compiler which enables applications to transparently leverage dataflow execution on GPUs.**
(4) **An evaluation of Kitsune across several diverse DL models, spanning inference and training, on a SOTA A100 class GPU.** We show speedups up to 2.8×, with up to 99% reduction in memory traffic (which indirectly serves as a form of power/energy savings). We also compare Kitsune to SOTA vertical fusion techniques and elucidate the reasons why Kitsune is able to achieve superior performance.

2 Background

This section presents an overview of DL, GPU hardware, it's connection to the BSP execution model, and pointers to recent hardware support.

Deep Learning. DL applications use learned parameters to make predictions on data across a variety of application domains, combining input and parameter **tensors** (multidimensional arrays) with mathematical **operators** such as linear projections (`Linear`) to produce outputs. A computational graph is constructed during execution which is then used in automatic-differentiation for computing gradients to "train" parameters. Common operators include linear projection, element-wise operators such as ReLU and addition, attention, layernorm, softmax, and convolution. Linear projection, attention, and convolution are all computationally similar; reducing to GEMMs whose dimensions are dictated by the operator parameters. Often GEMMs are colloquially used to express the entirety of work done by these operators.

GPU Hardware. Figure 1 (a) presents a modern GPU chip design [34]. A GPU comprises a set of multiple **Streaming Multiprocessor** (**SM**) processing cores, a globally shared L2 cache (among all the SMs), and main memory accessible through a high bandwidth interface. SM execution is managed by a GPU-global grid scheduler which is responsible for dispatching work sent from the driver over PCIe. The SM includes local data storage, including a large register file and a memory that can either be configured as an L1 cache or a software-managed scratchpad memory (also known as shared memory). Each SM also includes compute functional units for general computation (SIMT

Table 1. Description of Selected Applications

Application	Year	Use Case
DLRM	2019	Predicting ad clicks
MeshGraphNets	2020	Mesh based physical simulation
NeRF	2021	View synthesis
GraphCast	2022	Weather forecast prediction
Llama 3 8B	2024	Language modeling
Llama 4 Scout	2025	Language modeling

Cores), and dedicated hardware for accelerating tensor operations such as matrix-multiplication (Tensor Cores). The memory system additionally includes support for atomics which are facilitated by the L2. SM counts range from 80 for V100 [37], 108 for A100 [35], and 132 for H100 [36]. Roughly speaking, the L2's bandwidth is 3× of main memory bandwidth [11, 13, 14].

GPU Execution model. A GPU **kernel** (typically mapped one-to-one with DL operators) is code that is compiled and run on the GPU's SIMT abstraction. Kernels are run with a BSP execution model where one kernel occupies the GPU at a time and finishes completely before the next kernel starts. Each kernel's threads are organized into collections of threads known as **cooperative thread arrays** (CTAs, a.k.a. "threadblocks"), and all the CTAs of a kernel make up a **grid**. A CTA is a non-divisible quanta of work that is mapped to and executes to completion on an SM, where each thread maintains private state in the register file, and communicates with other threads in the same CTA via shared memory.

In the microarchitecture, threads within a CTA are grouped into fixed-size **warps** (32 for most modern GPUs) which execute in lock step. In modern GPUs, multiple CTAs can run simultaneously on a single SM. Modern GPUs allow multiple grids to execute simultaneously in limited situations, and have included rich support for atomic memory operations, allowing threads within a CTA, grid and across grids to synchronize with global atomics [31]. CUDA Streams [22] and CUDA Graphs [32] are APIs that enable users to specify which kernels are independent and can run simultaneously. In practice, neither of these result in co-executing kernels—due to first-in-first-out ordering and queuing hardware in the global grid scheduler. Current GPUs restrict that a new kernel can only start dispatching once all the CTAs from the current one have dispatched resulting in minimal execution overlap of the two kernels [33, 50].

3 Motivation and Program Behavior Characterization

In this section, we motivate dataflow execution by examining the opportunities present across a range of DL applications' operator graphs. The applications we focus on are summarized qualitatively in Table 1. Our DL applications include DLRM [29], MeshGraphNets [40], NeRF [28], GraphCast [18], Llama 3 8B [7], and Llama 4 Scout [26]. Note for Llama, we discuss it in terms of three separate use-cases: (1) training which encompasses the forward and backward pass for a whole set of tokens, (2) context-phase ("ctx") which encompasses just the forward pass prefill step of inference, and (3) decode-phase ("tok") which encompasses the autoregressive token-generation step of inference. The context and decode phases are inference only and will not appear in training results. We first discuss several common patterns, summarized in Figure 2, which we observe are frequently exhibited in popular DL applications, focusing on the limitations of state-of-the-art vertical fusion compared with Kitsune dataflow.

Operator Patterns. Figure 2 depicts three common graph patterns composed from `Linear`, `Elementwise`, and `Reduce` operators. These patterns are abstracted from detailed shapes for

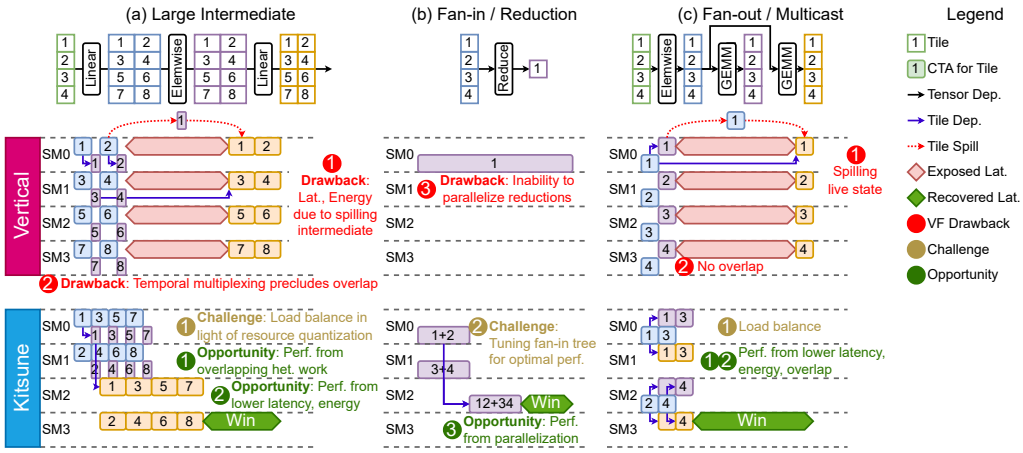

Fig. 2. Visualization of the difference between Kitsune and Vertical Fusion for (a) an MLP with a large hidden dimension, (b) a reduction operation, and (c) an operator that sends intermediate to multiple consumers.

our applications encompassing examples found in both inference (forward-pass) and training (back-propagation). `Elementwise` and `Reduce` operations are not computationally intensive and cannot use the TensorCores on the GPU, unlike GEMM operations which do. Figure 2(a) depicts a common scenario where a linear layer (i.e., a GEMM) produces a large output dimension ("N") which is then fed to a downstream `Elementwise` and subsequent linear layer. This is seen in many MLPs, and is especially common in the feed-forward network in transformer models which perform a projection (`Linear`) into a high-dimension followed by a non-linear operation and subsequent projection back to a smaller dimension. Figure 2(b) depicts a simple reduction operation. This can be found typically in split-K GEMM operations where partial sums need to be reduced. In addition, reductions over the batch dimension are very common in back-propagation. Finally, Figure 2(c) depicts a scenario where one `Elementwise` feeds two consumer `GEMM` operations. This is very common in back-propagation, notably for a `Linear-Activation` pair the backward pass involves computing the gradient for the activation function which feeds two gradient GEMMs—one for the input activation and one for the weights.

Vertical Fusion Mechanism. Vertical fusion seeks to improve DL performance by combining multiple DL nodes and *temporally* switching between partial executions of each node to avoid main-memory traffic of intermediate data. Different code-regions in a single vertically fused kernel encode the entire computation of the fused subgraph, with each CTA working on data-parallel shards of the problem. Keeping with the BSP execution model in which vertical fusion operates, CTAs do not interact with each other and tiles of intermediate data are only passed between the partial executions *within* a CTA. Therefore, implementations of vertical fusion prioritize staging data in shared memory or the register file [47, 64].

Vertical Fusion's Utilization Limitations. Vertical fusion is *unable to exploit idle GPU resources.* Figure 3 shows, for our application selection, a breakdown of application runtime with respect to SM and DRAM utilization measured from performance counters by NSIGHT Compute for vanilla PyTorch and inference compiled with TensorRT (representative of vertical fusion). We define "low" utilization as less than 33% of peak, generating four categories. "Both Low" implies that both DRAM and SM utilization are less than 33%, "Low SM" and "Low DRAM" categories have only one resource below 33% utilization, and "Neither Low" is time spent with DRAM and SM utilization above 33%.

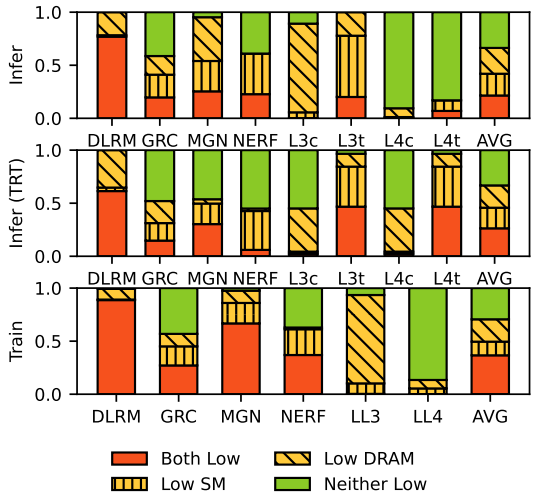

Fig. 3. Application runtime spent in different combinations of measured SM and DRAM utilization. Low utilization means less than 33% of peak.

While "Low" categories indicate portions of time spent with GPU resource severely underutilized, there remains some opportunity even in the "Neither Low" case.

Note that TensorRT does not support training so we only show TensorRT result for inference. Across our applications for bulk-synchronous (unfused) execution, we see 20–25% and 37–67% of runtime is spent with both low SM and DRAM utilization for inference and training, respectively; with the exception of DLRM (which has 77% and 89%) and Llama Context / Train (which has 0.1% and 0.3%). Indeed TensorRT fusion does improve this picture for inference with all applications showing a decrease in "low" utilization with the exception of MeshGraphNets. Despite this, there is still ample opportunity for dataflow to capitalize on idle resources shown by the large amount of runtime spent with low utilization of one or both resources. Even if neither resource are low, as exemplified by NERF inference with TensorRT, there's still opportunity for dataflow: operators executing by dataflow eliminates DRAM traffic which would lower the effective DRAM utilization, leading to additional headroom.

Vertical Fusion's Coverage Limitations. We discuss coverage limitations by considering graph patterns shown in Figure 2. In Figure 2(a), when an operator produces an intermediate with a large hidden dimension (e.g., MLP with $N \geq 768$ on an A100 with 192 KB of shared memory), the resultant GEMM tiles exceed the shared-memory capacity. Because of this, even modestly sized intermediates can cause spills to off-chip DRAM[1] As a result of this, the latency from a round-trip to/from off-chip DRAM is incurred for spilled data. On an A100 GPU, this latency is $\approx$ 409ns or 572 cycles at 1.4GHz. In addition to spilling, because of how vertically fused kernels temporally multiplex the SM, either the SIMT cores or Tensor cores will be idle during computation of each operation, leading to under-utilization of the SM. Naively mitigating this by assigning multiple CTAs to an SM has a major drawback of cutting the effective shared-memory per CTA by the same factor, exacerbating the capacity problem.

Figure 2 (b) depicts a reduction operation. One notable and unavoidable place where reductions are common is in back-propagation where gradients are often reduced over the batch dimension.

[1]Indeed the L2 could provide some additional buffering but since every SM runs a data-parallel replica of the same subgraph with the same intermediate storage requirements, that capacity is quickly overrun as well.

Despite the batch dimension usually being a source of abundant parallelism, here neither BSP or Vertical Fusion are able to extract parallelism from the batch dimension for gradient reduction operation. This means that a small number of CTAs end up performing a reduction, leaving most SMs idle.

Finally, Figure 2(c) depicts a case where one operator's output is consumed by multiple downstream operations. In particular, this pattern of multicast is representative of back-propagation for a standard `Linear+Activation` graph. Similar to (a) we find this can lead to spilling tiles of data to off-chip memory since the state needed for one successor child may over-run the shared memory, evicting an intermediate that is needed for a different child. We also see that heterogenous operations cannot simultaneously execute on the SM, leading to underutilization. In general, we observe prior work on vertical fusion does not support back-propagation at all, though we depict in our figure how it would be implemented.

Kitsune. Our insight is that dataflow – i.e., having different operators co-execute *spatially across SMs*, rather than temporally switching between executing operators across *time* – solves all these problems, while preserving the benefits of vertical fusion. Kitsune constructs a spatial pipeline to achieve dataflow execution: mapping single operators to CTAs, then passing tiles of intermediate data to downstream operator CTAs using inter-CTA queues residing in on-chip memory to avoid off-chip memory accesses. In doing so, operator CTAs are concurrently mapped and executed across the SMs of the GPU. Multi-cast and parallel reduction simply become one-to-many and many-to-one communication patterns using our data-queue. The capacity issue, is then trivially solved by splitting hidden dimensions spatially. Using our modified grid scheduler, hardware under-utilization can be solved by assigning different *types* of CTAs to an SM for co-execution.

Revisiting Figure 2, Kitsune can extract performance wins for all of these patterns. First, Kitsune is able to simultaneously execute heterogeneous operations on an SM, addressing under-utilization. Second, with significantly reduced data-movement (especially to/from off-chip DRAM) energy is saved, potentially allowing for higher clock frequencies to be sustained. Finally third, Kitsune is able to extract parallelism from hidden and reduction dimensions.

4 Kitsune Primitives for Dataflow on GPUs

Kitsune enables the GPU to logically operate with a synchronous dataflow execution model that relaxes the assumptions of bulk-synchronous execution, relying on and leveraging dependence and communication *between* CTAs from different pipeline stages. The execution model comprises of CTAs explicitly communicating with each other which triggers and throttles execution speeds. When data is available in a queue, a CTA starts its execution, writing results to its producer queue. When there is no data in its queue, it idles. The **first** node of a subgraph reads activations from main-memory (essentially outputs of preceding subgraphs or bulk-synchronous code), and the last node writes results to main-memory. In addition to reading from a queue, a CTA is free to read any other values from memory, and similarly can write to main-memory in addition to writes to its producer queue to trigger its successor. In the formal context of execution models models [19], Kitsune falls under the category of *synchronous dataflow*. Future work can examine further extensions like dynamic dataflow.

The following subsections develop Kitsune's two key primitives that enable this synchronous dataflow execution model. The first is a synchronized queue structure which allows inter-CTA communication (Section 4.1). The second is a modified grid scheduler that exploits heterogeneity among executing CTAs to facilitate fine-grained overlapping execution on the SM (Section 4.2). We conclude this section by discussing the logical execution model that Kitsune's primitives now provide.

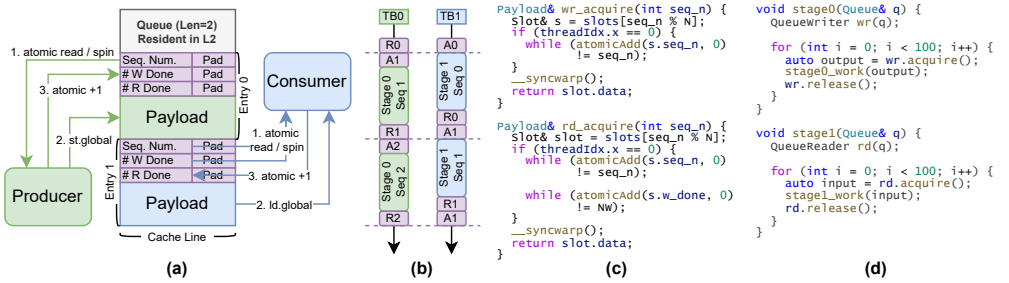

Fig. 4. Queue design. Note: release routines are not shown for space reasons. They involve simple `atomicAdd` calls to update synchronization metadata and a CTA barrier with `__syncthreads()`.

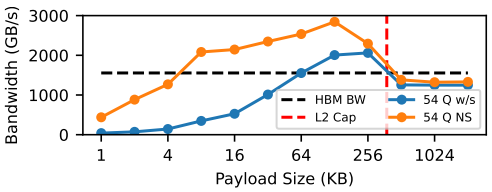

Fig. 5. Performance of GPU atomics.

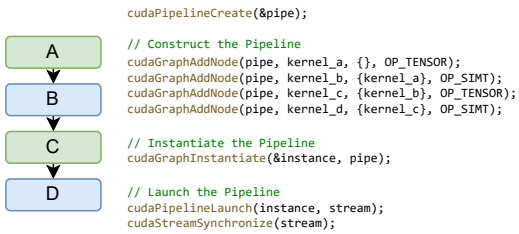

Fig. 6. Code snippet of the proposed cudaPipeline API.

4.1 Producer Consumer Communication

We use GPU atomics to design a synchronized, ring buffer queue for passing data between CTAs. Queues are pinned in the L2 cache using CUDA API functions [35] (Figure 4(a)). Each entry contains metadata protected by atomic accesses. Figure 4 shows (a) a diagram of our queue design (with two entries for double-buffering), (b) a timeline of producer-consumer operations, (c) stylized code implementing the queue, and (d) application-level usage. Two CTAs communicating (Figure 4(b)) "acquire" and "release" entries, achieving ordering via sequence numbers. The producer and consumer acquire entries (`wr_acquire` and `rd_acquire` in Figure 4(c)) by spinning on metadata variables until an entry is freed for use. `acquire` and `release` are exposed as an API which handle sequencing automatically. Typically, only one CTA is spinning on a variable at a given time—meaning our queue design results in very low contention.

Our queue is implemented as a library with two API functions: `acquire` and `release`. This allows for easy software integration, introducing minimal overhead exploiting the modern GPU's sophisticated warp-scheduler. Queue code is wrapped with `if threadid==0`, ensuring only one thread in a CTA does any of the queue management. To avoid data-races, "release" operations require a CTA-level barrier. Figure 4(d) shows how it can be used intuitively by a CUDA programmer or inserted by a source-to-source compiler into existing CUDA kernels. Synchronization variables are all padded to the size of a cache line to avoid false-sharing.

Queue Performance. Using a microbenchmark, we measure the A100 can sustain 100 M atomics / sec / CTA when under no contention. Based on additional measurements, we find this lends to an upper bound of 385-1541 GB/s *per queue*, far exceeding L2 and HBM bandwidth (≈61 GB/s per SM). We evaluate our queue's performance by measuring SM-SM bandwidth with varying payload sizes for 54 queues (108 CTAs for the 108 SMs of the A100 GPU). Figure 5 shows queue management overhead by measuring the performance of data transfers with and without synchronizing atomics enabled. We find with 128-256 KB payloads, aggregate bandwidth reaches 2 TB/s (37 GB/s/queue). Beyond 256 KB, performance drops due to queue sizes reaching the L2 capacity, causing accesses to

spill out to HBM (Limiting us to 1.5 TB/s for A100). Synchronization overhead is large for small queue sizes: 12× reduction in bandwidth for 1KB payloads. With larger payloads this reduces: synchronization overhead is less than 63% for ≥64KB payloads. *Overall, we find GPU global atomics performance is more than enough for our use case. We also find our atomics-based L2 resident queue provides substantial inter-CTA communication bandwidth even in the presence of contention for payloads ranging between 64-256KB.*

Deadlock and Correctness. Our queue implementation is a simple bounded buffer where each slot in the buffer is synchronized between producer and consumer using a variation of a ticket lock [24]. The queue will block the producer if it is full, and block the consumer if it is empty. Our queue implementation is correct, using standard principles of atomic operations for synchronization. In bare-metal implementations that use the queue, the user is responsible for determining the producer/consumer relationships to avoid incorrect code that could result in deadlock. When instantiated by a compiler, it must do an analysis pass to determine the producer/consumer patterns and data production rates to avoid deadlock, which we discuss in Section 5. Finally, for the purpose of ensuring memory consistency between producer and consumer, the `wr_release` path includes a fence operation to ensure write ordering, and the `rd_acquire` path invalidates the consumer SM's L1 cache (not depicted in Figure 4).

4.2 Scheduling Heterogeneous CTAs

In order to capitalize on idle resources of the SM—for example, make full use of both the Tensor Core and SIMT Core simultaneously—we propose a modest change to the CUDA API and GPU Grid Scheduler to specify spatial pipelines (shortly defined) of kernels and maximize GPU resource usage. This is important for enabling and managing true co-execution of kernels which is not supported on current GPUs (Section 2).

CUDA API Exposure. We introduce an abstraction we call a CUDA "spatial pipeline", with a similar API to CUDA Graphs but different semantics. Like a CUDA graph, a spatial pipeline specifies a collection of kernels to execute with the key difference being it implies all kernels in the collection require being co-resident on the GPU. The calling code is responsible for limiting the number of CTAs launched per kernel to ensure co-residency is possible (Section 5.3). Figure 6 shows a snippet of host code which specifies and configures the launch of a spatial pipeline. Data dependence information is specified similar to CUDA graphs, and kernels are configured with new metadata that specifies the primary type of dynamic resource they require, either SIMT or TENSOR.

Hardware Implementation. To complement our CUDA spatial pipeline abstraction, we propose a modest change to the grid scheduler that allow it to leverage the type information now passed via the kernel call header. On current GPUs, the grid scheduler hardware stores occupancy info for how much of each SM's resources are consumed, which is used to greedily find the first available SM for CTA dispatch using a hardware arbiter (i.e., round-robin) [50]. However, this greedy policy does not work for Kitsune as it does not guarantee overlap; We need to ensure that CTAs of different types are effectively paired for execution on the SM. We augment the round-robin prioritization hardware with two arbiters, one for each type. The two arbiters enables the scheduler to effectively pair different types together, by separately considering dispatch to the same SM. When a new kernel arrives, the arbiter is selected based on the type. The CTA scheduling then proceeds as usual, checking the occupancy of the current SM under consideration for dispatch.

5 Kitsune Compiler Design

In this section, we develop the Kitsune compiler, which enables DL applications to transparently leverage dataflow. We implement Kitsune as a PyTorch [25] compiler backend. We use PyTorch

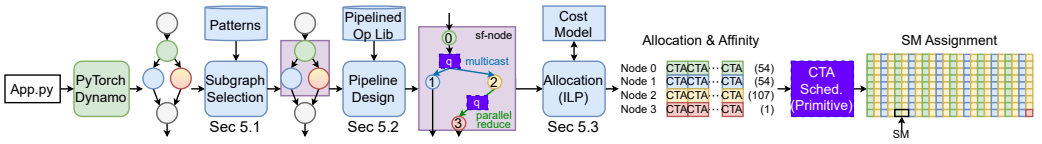

Fig. 7. Depiction of the Kitsune compiler flow. Our enabling primitives from the previous section are highlighted in purple.

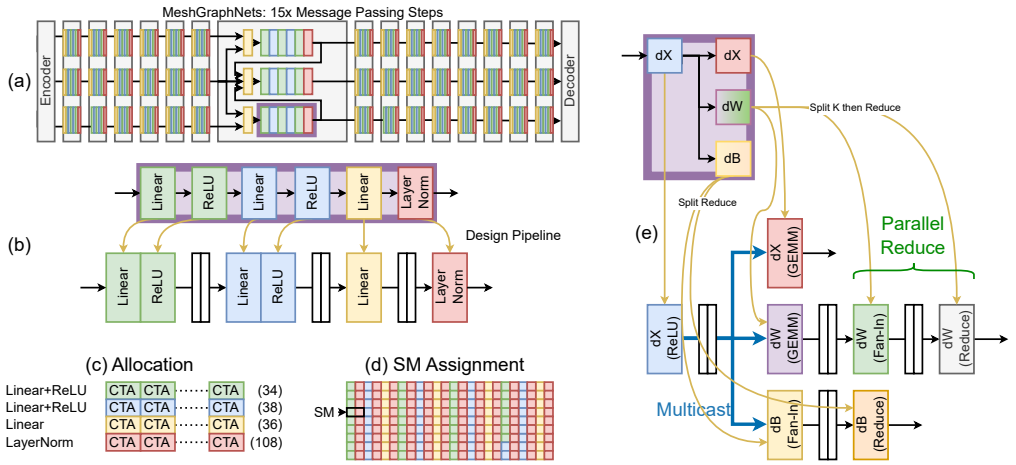

Fig. 8. Running example of two subgraphs selected from (a) MeshGraphNets. (b) shows an MLP selected from the full application graph forward pass and its pipeline design. (c) and (d) show the allocation and assignment. (e) shows a subgraph from the backward pass for a Linear+ReLU and its pipeline design. We omit allocation/assignment for space. White rectangles in (b) and (e) represent queues.

2.0's Dynamo interface for extracting application graphs including both the forward pass and back-propagation for training. Our compiler backend consumes these graphs and constructs spatial pipelines for execution.

To realize this compiler, several challenges must be addressed. First, subgraphs must be selected from the original application graph for fusion (Section 5.1). Second, a pipeline must be designed for the subgraph with stages corresponding to operators (Section 5.2). Third, the stages must be assigned GPU resources to achieve optimal performance addressing load-balancing (Section 5.3). Figure 7 depicts how each of these pieces are applied to a PyTorch application. The compiler leverages our software queue structure and modified GPU grid scheduler to enable inter-stage communication and intelligent CTA placement to exploit fine-grain SM resource sharing. Figure 8 depicts how our compiler lowers MeshGraphNets, and will serve as a running example throughout this section.

5.1 Subgraph Selection

We first need to select subgraphs for dataflow execution which involves marking groups of DL operators in the computation graph for co-execution in a pipeline. We denote these groups of operators / nodes that form a pipeline as an `sf-node`. The output of this phase is labeled graph with `sf-nodes` identified. At the graph level, a spatially-fused group (`sf-node`) of operations must be "contiguous" as defined in [49]—that is, there must be no edge which exits the subgraph with a down stream edge that reenters it. Subgraph selection influences pipeline design, allocation,

ALGORITHM 1: Algorithm for pipeline design

```
for n in Graph do
    if IsReduction(n) then
        fanin, final ← SplitReduction(n)
        Graph.replace([n], [fanin, final])
        n ← final
    end
    if IsIntermediate(n) then
        q ← CreateQueue(n)
        for c in Dependents(n) do
            c.producer ← q
        end
        n.dependents ← [q]
    end
end
```

assignment and hence performance, potentially requiring an iterative solution. As a practical solution, we implement a single-pass design that use two rules to exclude a node from a subgraph: nodes that are bulk-sync friendly and nodes that index / gather across all data (gather nodes for embedding for example). With such node exclusions defined, subgraph selection converts to pattern-matching.

Our design and implementation of subgraph selection is heuristic based and uses manual pattern matching. By examining applications properties we identified node patterns that are candidates for subgraph exposing the vulnerabilities of bulk-synchronous execution and vertical fusion. It is essentially a set of regular expressions that express patterns including those seen in Figure 2. In particular, our implementation operates at the topological order which linearizes the graph into a list in PyTorch Dynamo (which is deterministic). In practice, additional regular expressions to express different orderings for the topological order are easy. A more formal automata that captures all possible linearizations of a subgraph is beyond the scope of this work.

We leverage PyTorch's Dynamo to extract whole operator graphs of the forward and backward passes for an application. We then created a library of patterns that expresses patterns that are candidates for subgraphs. We implement a pattern-matching algorithm for then selecting subgraphs from the original application graph for dataflow execution. This approach searches for user-specified chains of operators in a topological order. Adding new patterns is a trivial task of adding to our pattern library. Figure 8(a) shows how we selected a subgraph for MeshGraphNets.

5.2 Pipeline Design

The pipeline design problem comprises of inserting queues between nodes of an `sf-node`, and if the work done between two nodes is trivially fusable, fuse them using epilogue fusion (or vertical fusion). The output is a transformed graph which includes one or more queue nodes added, which can then be lowered to CUDA code during code-generation.

Conceptually what this means is taking the original set of operations in the graph and either combining or splitting them to map to pipeline stages that are realized by pipeline-enabled CUDA kernels. For simple patterns with 1-1 producer-consumer relationships, the decision is trivial - and involves insertion of queue nodes between nodes of an `sf-node`. For more complex patterns like attention and back-propagation, we implement a parallel reduce which uses our queues to form a reduction tree. Figure 8(b) and (e) show a pipelined graph starting from our MeshGraphNets subgraph and back-propagation of a single `Linear` layer, respectively.

ALGORITHM 2: ILP formulation for load balancing.

$$
\begin{aligned}
\text{maximize} \quad & thrpt \\
\text{subject to} \quad & thrpt < r_i * s_i * t_i \quad (i = 1, \ldots, n) \\
& thrpt * (\text{DRAM Bytes}) < \text{DRAM}_{peak} \\
& thrpt * (\text{L2 Bytes}) < \text{L2}_{peak} \\
& t_i = \text{Bulk-Sync Thrpt. for Op } i \\
& r_i = \text{ResourceScale}(a_i) \\
& s_i = \text{Speedup}(a_i) \\
& 1 \leq a_i \leq \#\text{ SMs} \\
& \sum_{i=1}^{n} \text{IsSimt}_i * a_i = \#\text{ SMs} \\
& \sum_{i=1}^{n} \text{IsTensor}_i * a_i = \#\text{ SMs}
\end{aligned}
$$

In terms of implementation this involves three steps. The graph rewrite algorithm is shown in Algorithm 1. In terms of code-generation, the queue implementation is discussed in Section 4.1. The third step is to take CUDA kernels and transform them to read/write from queues, instead of from global memory. This last step also includes the process of working on tiles, since a queue's payload needs to be limited. In all cases, the notion of tiling already exists or is trivially doable; for GEMMs the code is already written to work on tiles of inputs and outputs. Completely automating this step for arbitrary code is likely infeasible and involves all the challenges of aliasing analysis etc. For Kitsune, we performed this step manually - it took about 8 person-hours or less for each kernel, with the source-code lines changed ranging from 10 to 40. The limitation this adds to Kitsune is that it is not completely turn-key for new operators not previously seen by the compiler, requiring very modest library modifications of the underlying *new* DL operator. In practice, library developers like NVIDIA and AMD can incorporate such a flow trivially into their development process.

5.3 Load Balance

Load balancing in Kitsune involves the logical allocation of # CTAs to each node in an `sf-node`. Its output is an allocation as shown in Figure 8(c). This needs to be done cognizant of overlapped execution of dissimilar CTAs on the same SM.

We use a zero-latency performance model to estimate the throughput of a spatially-fused subgraph based on an allocation of CTAs to each stage. We then formulate the allocation problem as an **integer linear program** (**ILP**) which can be used with standard solvers to produce an optimal assignment which maximizes throughput of the subgraph. We augment our ILP formulation to enable over-subscribing CTAs onto SMs to enable overlapping dissimilar behavior—specifically, we consider two classes of operations: SIMT-heavy, and TensorCore-heavy, and assume an SM can simultaneously execute one of each with no performance degradation. We discuss in Section 4.2 low level details of how this overlap can be leveraged on modern GPU hardware.

Algorithm 2 shows our ILP formulation. We model throughput as the minimum throughput pipeline stage in the fused subgraph additionally constrained by memory bandwidth and aggregate L2 bandwidth based on analytic evaluation of the total number of bytes read/written from DRAM (DRAM Bytes), and L2 (L2 Bytes). For the i^{th} of n operators in a spatial pipeline, we estimate the performance by combining a measured BSP throughput (t_i) with an estimate of how the performance will scale (speedup or slowdown) based on how many CTAs it is assigned (r_i =ResourceScale(a_i)) and an estimate of speedup afforded by operating where some number of its operands are now

resident in on-chip storage instead of DRAM (s_i =Speedup(a_i)). In practice, we define Speedup(a_i) to be $1/u$ where u is the maximum resource utilization of the SIMT or TensorCore pipelines. We allow the number of SIMT and Tensor stages to independently be assigned SMs to exploit overlapping these dynamic resources. In practical deployment terms, we require either a two-pass compiler, run-time optimization pass, or a dictionary of kernel characteristics to get u_i to guide the ILP. Since DL models generally run in a curated environment (TensorRT for example), any of those approaches are practical, and do not introduce any application slowdowns.

5.4 Deployment Considerations

There are several considerations for deploying Kitsune in a practical setting which we now discuss.

Dynamic Shapes. DL workloads occasionally will have certain dimensions that can vary in size depending on the input to the model. A notable example being for transformers, the input token length and batch dimensions can both be arbitrary size. Kitsune uses a combination of two approaches to handle this. First, PyTorch's Dynamo graph capturing will concretize most tensor dimensions ahead of the compiler being called. Often only the batch dimension is left unspecified, and Kitsune will then choose this dimension to stream to the spatial pipeline. In general, modern DL workloads like the ones we study never exhibit a dimension size that cannot be inferred by Dynamo *and also* cannot be streamed to the spatial pipeline.

Multi-GPU Execution. In real-world settings, splitting work across multiple GPUs may be employed to improve performance or capacity. Kitsune naturally composes with distributed DL. For producer-consumer relationships which require inter-GPU collectives, a spatial pipeline would be broken into two graphs, inserting the inter-GPU collective between them. In Sec 7 we discuss other approaches to handling this to hide communication latency.

Compiler Overhead. Once each individual operator is modified to be capable of spatial fusion, generating the code to handle the pipeline and compiling it is handled by a traditional compiler flow and takes on the order of seconds to complete for a whole spatially fused pipeline. This fits naturally into framework compilation flows such as PyTorch dynamo, where some start-up overhead is expected.

6 Evaluation

We now examine the effectiveness of Kitsune across our applications and GPU models. We guide our evaluation with the following questions: (i) How well does Kitsune support composing arbitrary operations across DL applications? (ii) What is the end-to-end performance of applications running with Kitsune and what are the reasons for variation across applications and modes of operation? (iii) What is the sensitivity of our performance and gains to machine parameters including on-chip compute (number of SMs), off-chip DRAM bandwidth, and L2 and crossbar bandwidth?

6.1 Methodology

Our evaluation is based on running our 5 applications in a validated GPU simulator emulating an A100 GPU which takes as input the compiled versions of our applications. We built our compiler and a queue library (Section 4.1) (characterized and run on silicon). Because we need our grid scheduler modifications to allow the overlap afforded by Kitsune (Section 4.2), we evaluate Kitsune using a modified version of NVIDIA's NVArchSim (NVAS), a hybrid trace- and execution-driven GPU simulator [53] that has been validated against NVIDIA's Ampere GPU. This also allows us to study sensitivity to individual hardware features, instead of being restricted to particular SKUs.

Our baseline for speedup results is unmodified PyTorch execution. We use our compiler and modeling flow based on NVAS to present speedups afforded by both vertical fusion and Kitsune. Figure 9 shows the fusions that are chosen by our compiler: thick orange boxes on the left side show

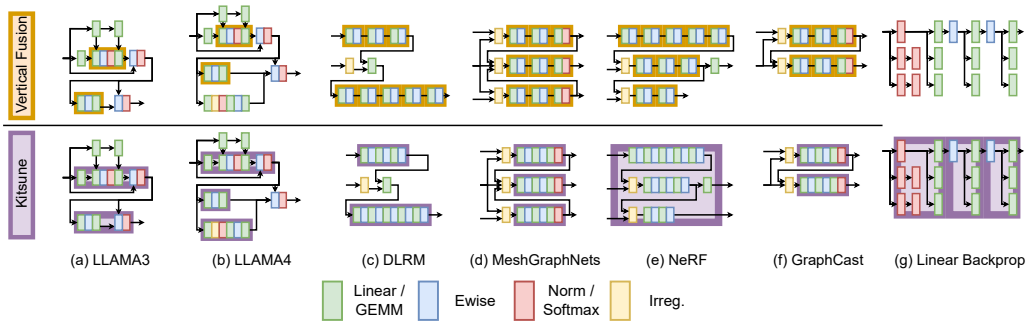

Fig. 9. Depiction of applications and the fusions we apply.

the fusions we select based on vertical fusion techniques, while thick purple boxes show the fusions made possible with Kitsune. **Note: our model of vertical fusion combines the techniques and mechanisms from state-of-the-art industry and academic approaches of TensorRT [51], AStitch [64] and Welder [47].**

We first describe the quantitative scope of the opportunity that Kitsune provides. We then discuss inference and training separately. For Kitsune, we present results for both the subgraphs of the applications as well as the speedup for the entire application.

Recall our queue library is implemented in CUDA, validated and performance measured on actual GPU chips. Owing to the aforementioned issue of grid scheduler changes needed (even though very modest) we use a simulation based approach for end to end Kitsune performance evaluation on full applications.

6.2 DL Application Operator Coverage

Table 2 provides a characterization of the applications at the DL operator level denoting the number of operators that are grouped into pipelines. The top half of rows are for inference and the bottom half are training. Note this data is for operator count (we discuss time below). For the majority of our applications, >70% of operators are candidates for grouping, with higher coverage for inference. We note that vertical fusion covers only the forward pass operators for training[2] and it's coverage is typically lower. The last two columns show memory traffic savings both for Vertical Fusion and Kitsune. Traffic savings is useful in itself, as it results in energy/power savings (by downclocking the memory frequency to sustain the lower bandwidth needs). O'Connor et al. [38] and others [8, 16] have argued that GPUs are becoming memory power limited.

6.3 Inference Performance

Figure 10 shows the speedup Kitsune provides for each of the subgraphs in each of the applications. Figure 11's timeline show the time contributed to overall execution by each of the subgraphs, and in gray we show the time the application spends in kernels/operators that run in bulk-synchronous mode. Figure 11's bar-charts show full application speedup.

Overall, sub-graphs speedup up to 3.4× across the applications, with a geomean of 1.7×. Llama 3 and 4 generally observe the lowest speedups, primarily because both models are quite large, making loading parameters from off-chip memory the primary concern for performance. For attention in the context phase in particular, the baseline using flash-attention achieves similar memory traffic efficiency to what Kitsune provides. Higher speedup is achieved in the attention mechanism

[2]We note that none of the academic work or TensorRT have demonstrated execution of training yet—our results are thus optimistic for vertical fusion.

Table 2. Summary of Fusions and Traffic Reductions

		Fusion Coverage		Traffic Red.	
App	**# Ops**	**Vertical**	**Kitsune**	**Vert.**	**Kitsu.**
Inference					
DLRM	21	17 (81%)	17 (81%)	22.53 %	47.78 %
GRC	35	21 (60%)	29 (83%)	23.98 %	57.25 %
MGN	51	36 (71%)	41 (80%)	56.54 %	57.81 %
NERF	24	18 (75%)	24 (100%)	40.19 %	98.66 %
L3-CTX	27	10 (37 %)	19 (70 %)	10.04 %	49.07 %
L3-TOK	27	10 (37 %)	19 (70 %)	0.01 %	0.07 %
L4-CTX	39	10 (26 %)	32 (82 %)	0.48 %	70.79 %
L4-TOK	39	10 (26 %)	32 (82 %)	0.00 %	0.07 %
Training					
DLRM	59	18 (31%)	46 (78%)	7.86 %	27.51 %
GRC	101	20 (20%)	76 (75%)	9.06 %	40.11 %
MGN	148	36 (24%)	108 (73%)	21.76 %	40.33 %
NERF	69	18 (26%)	56 (81%)	14.13 %	45.52 %
LLAMA3	88	10 (11 %)	34 (39 %)	2.85 %	45.16 %
LLAMA4	125	10 (8 %)	66 (53 %)	0.11 %	71.46 %

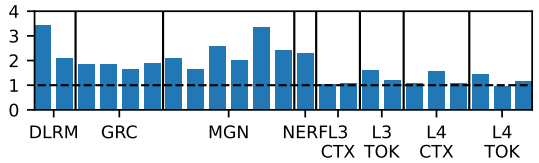

Fig. 10. Inference subgraph speedups.

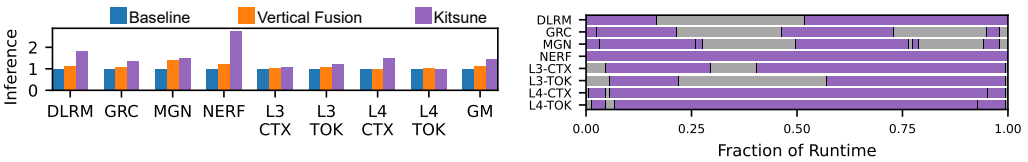

Fig. 11. Inference end-to-end speedup over bulk-sync.

during token generation for both models since overall traffic volume is much lower; making the savings from spatially pipelining activations more impactful. For Llama 4 in particular, the MoE mechanism benefits in the context phase by streaming the batch and token dimensions, eliminating memory traffic from writing intermediates from each expert to memory. NeRF is an example where large speedup is achieved (2.3×), highlighting many of Kitsune's benefits: all the nodes of NeRF's forward pass are spatially fused, allowing most layers to pull intermediates from a queue instead of main-memory; and the concat operations are free to occupy the SIMT units of the SMs while the GEMMs use the TensorCores. Due to the intermediate sizes, vertical fusion cannot fuse NeRF's linear layers[3].

[3]We use the original NERF configuration which uses hidden dim = 256.

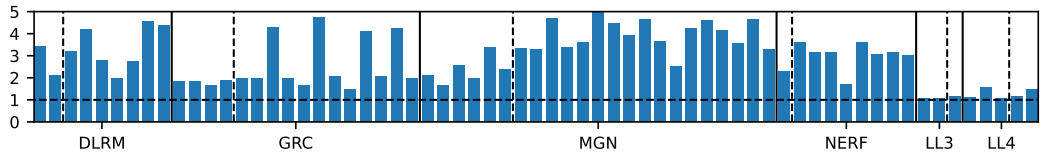

Fig. 12. Training subgraph speedups. Dashed lines separate forward and backward passes.

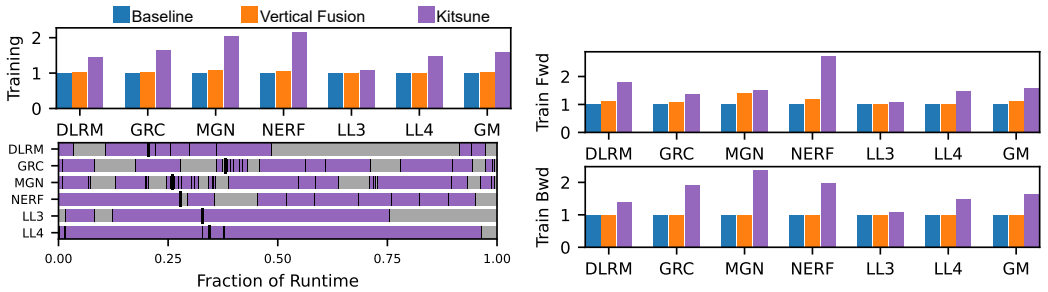

Fig. 13. Training end-to-end speedup over bulk-sync.

When looking at full application performance, we observe two phenomenon: large portions of time are spent in the sub-graphs (typically > 50%), and a single application has few subgraphs (the black lines in Figure 11 indicate end of a sub-graph). For end-to-end performance, we see geomean 1.5× speedup. Llama 3 context shows the least speedups because its subgraphs' speedup is modest (4–8%), despite its sub-graph coverage in time is 84%.

Takeaway: *We find Kitsune provides substantial performance opportunity for DL inference with this generally scaling with number of fused operations. We observe DRAM traffic is substantially reduced, suggesting higher performance could be possible without increasing bandwidth.*

6.4 Training Performance

Figures 12 and 13 show the corresponding results for training, with training broken down further in terms of the forward and backward pass. The forward pass is similar to inference, with the added issue of intermediate activations being stored to main-memory for computing gradients. The backward pass then uses these to compute gradients for parameters.

Considering end-to-end speedup, we see two trends. As expected, the backward pass takes about 2× the time of the forward pass. Less fractional time of the backward pass is spent in spatial mode, especially for DLRM, where the backward pass for the feature interaction which is not spatially fused takes substantial runtime, causing an Amdahl's law effect on training back-backpropagation. End-to-end speedups range from only 1.1× to as high as 2.2×.

Takeaway: *Kitsune still enables performance gains for Deep Learning training, with lower improvements due to smaller fusions in the backward pass compared with forward. Because of Kitsune's ability to parallelize reductions, training benefits more from spatial fusion compared with the parallelism-limited bulk-synchronous baseline.*

6.5 Comparing to Vertical Fusion

Due to the limitations outlined in Section 3, effectiveness of Vertical Fusion is substantially lower than Kitsune for inference, with MeshGraphNets showing the best speedup (1.4×) with geo-mean 1.1× (Figure 11). Since it only applies for the forward pass, training speedups are even lower (Figure 13). Related works like Welder, for inference, have reached similar findings: when applied to production settings of running with TensorCore and meaningful batch-size (like 32 or larger),

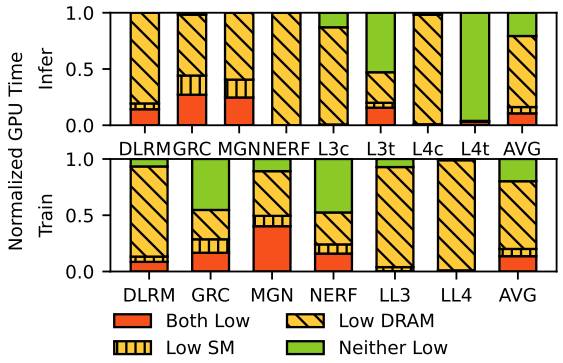

Fig. 14. Application runtime spent in different combinations of SM and DRAM utilization as reported by our model. Low utilization means less than 33% of peak.

speedups over un-optimized PyTorch (worse than our baseline) is 30% or so, with no speedup over TensorRT on Nvidia V100 [47]. Those works target additional scenarios like FP32 based computation (thus eliding our overlap opportunity) and edge-case scenarios like batch-size=1, which are less important in production data-center deployment. Philosophically they target improvements through software in the configuration space where GPUs are inefficient (bs=1, fp32 mode etc). We focus on production scenarios: batched training and inference using TensorCores to address inefficiencies.

6.6 Comparing SM and DRAM Utilization

Figure 14 shows a breakdown of application runtime spent with different resource utilization when running with Kitsune. For inference, comparing to our data in Figure 3, we see 21% and 11% of runtime is spent with both low utilization for BSP and Kitsune, respectively. For training, we observe on average, Kitsune only spends 14% of runtime in low utilization compared with 37% for bulk-synchronous. In addition, Kitsune on average spends much more runtime with just low DRAM utilization for training: 77% vs 58%. This difference is less pronounced for training compared with inference because training requires more DRAM traffic to save intermediate activations for back-propagation.

Takeaway: *We find Kitsune is able to capitalize on the under-utilized resources of the GPU, reducing runtime spent with low resource utilization for most of our applications.*

6.7 Ablation Study

We now breakdown the components of Kitsune and examine which of the features of our system are critical for realizing the performance benefits of spatial pipelining. Figure 15 shows three different configurations of spatial pipelining. "Just Pipes" demonstrates the performance of a spatial pipeline with even allocation and no scheduler modification to enable co-occupancy of heterogenous work. "Pipes+Alloc." demonstrates the performance if only our load-balancing ILP formulation is applied to allocate CTAs to each stage. The results for Kitsune are shown which incorporate both allocation and co-occupancy.

Across the models we see that simply applying spatial pipelining naively leads to slow down in most cases, and little speedup in some cases. Without taking into account the balance of work in each stage, unless an even allocation is sufficient, the spatial pipeline will be unbalanced, leading to this. In general, this naive pipe-only approach gets a geomean 0.78× and 0.84× performance degradation for inference and training, respectively.

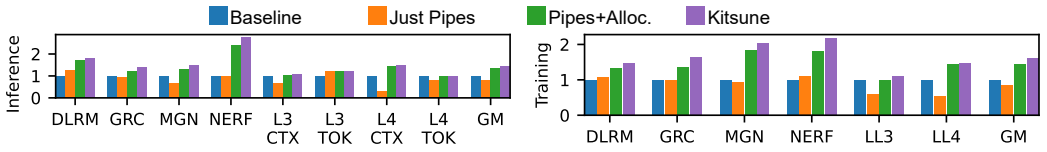

Fig. 15. Training end-to-end speedup over bulk-sync.

We find load balancing is the largest enabler of performance once spatial pipelining is applied. Compared with only pipes, we see a geomean of 1.78× and 1.70× performance improvement for inference and training.

The benefit of enabling co-occupancy of heterogeneous is seen primarily in models with a better balance of Tensor and SIMT operations. With NeRF, there is an activation function after nearly every linear layer, leading to a 15% performance benefit for inference from this kind of overlap. Compare this to large language models which have relatively little SIMT work compared with tensor operations which see marginal benefit (< 6% in most cases).

7 Future Work

We have demonstrated Kitsune provides considerable opportunity for traffic reduction and performance for several DL applications. We now study future directions for spatial pipelines on GPUs.

Convolution. Convolution operations can already be pipelined with Kitsune using the batch and filter dimensions. However, for very large spatial dimensions, it may not be feasible to contain an entire spatial feature map in a cache-resident queue. One future direction of Kitsune is to consider spatial pipelining for the spatial dimensions of a convolution operation. This is a challenge due to how the access pattern of a spatial feature map often creates halos of overlapping input regions that are reused for spatially proximate output pixels.

Fusing at Different Levels of Memory. Currently, Kitsune uses L2 cache-resident queues for transmitting date between spatial pipeline stages. However, with careful placement of stages, communication via local memories such as the register file, shared memory, and distributed shared memory could be possible and provide latency and energy benefits. Future work will explore how to discover these opportunities and facilitate work-placement needed to leverage localized private memory for communication.

In-pipeline Collective Communication. While Kitsune is composable with distributed execution, it currently will leave spatial-execution mode for any producer-consumer that requires off-chip collective communication. Future work will explore how collective operations can be handled "in-situ" in a spatial pipeline, to elide the need for global barriers during execution.

8 Related Work

DL Operator Mapping. Pipeline design for Kitsune is related to the problem of "operator mapping". This has largely been looked at in the context of spatially exposed hardware for *single operators* including works such as TimeLoop [39], MAESTRO [17], AMOS [62], and CoSA [10], which treat an operator as a transformable loop-nest, and TVM [4] which lowers semantics expressed with einsums to low-level code.

DL Operator Fusion. Traditional GPU kernel fusion focuses on fusing memory-intensive kernels together [43, 44, 54, 57], and modern DL compilers often support simple operator fusion at the register level [23, 30, 61] or for improving data reuse for identical and related operators [12, 48, 55].

Building on single-operator mapping, many recent academic works address vertical fusion including ALCOP [9], Apollo [60], AStitch [64], Chimera [63], Deepcuts [15], GraphTurbo [59], and Welder [47]. We discuss the capability of AStitch, Welder, and state of art vertical fusion in Section 3. AStitch, Welder and GraphTurbo all use some notion of an anchor-and-propa-gate scheme to handle streaming compatibility between fused layers. Kitsune is more composable and general than all of these, being able to fuse many more operators into co-resident GPU kernels. Other drawbacks and limitations of vertical fusion have been discussed at length in Section 3.

GPU Multitasking. HFuse [20] presents a methodology for horizontal fusion which can leverage overlap of heterogeneous work but is restricted to only fusing pairs of nodes with no data dependencies. Works such as ISPA [58] and SMK [56] provide a pure software, and hardware-codesign solutions (respectively) for achieving fine-grained multitasking on GPUs. SMK uses hardware mechanisms to enable preemption of CTAs on the SM for "partial context switching" – the goal of which is achieve higher overall utilization of SM resources with heterogeneous CTAs. ISPA uses a pure software approach for co-scheduling pairs of Tensor-heavy and SIMT-heavy kernels. It uses several software techniques to promote efficiency of co-occupancy, but ultimately relies on the existing GPU thread scheduler to make CTA placement decisions. All these approaches focus on co-scheduling just two kernels with no data dependence. Kitsune enables any number of kernels to co-execute in spatial pipelines with data-dependencies supported by our queues and relying on a modified CTA scheduler to make smart decisions about placement of CTAs to best utilize SM resources.

Data-Triggered Execution. WorkGraphs [27] is a recent development in the graphics space to afford data triggered execution on GPUs. However, it does not address on-chip data-orchestration to maintain cache residency of intermediates. Additionally, it operates on a level of granularity much smaller than Kitsune, using individual records and shader invocations as the unit of work. Kitsune in contrast is designed to orchestrate producer-consumer communication on-chip at a granularity of tensor tiles of around 64KB payloads. Finally, WorkGraphs does not support join operations with different input record types, vastly reducing the generality and applicability beyond shader pipelines.

9 Conclusion

We observe that the GPU BSP model limits its effectiveness for various important DL workloads, with state-of-art vertical fusion still leaving performance opportunities untapped. We design and implement Kitsune which enables synchronous dataflow execution for modern GPUs, leveraging existing support for synchronization and integrating into both CUDA and PyTorch. It's only hardware modification is extension of the GPU grid scheduler to be aware of affinity of CTAs to the SIMT vs TensorCore units. Kitsune reduces both main memory traffic and end-to-end runtime across DL networks on GPUs for both inference and training.

References

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al. 2020. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. DOI: https://doi.org/10.1109/ISCA45697.2020.00023

[2] AMD. 2020. Versal: The first adaptive compute acceleration platform (ACAP). https://docs.amd.com/v/u/en-US/wp505-versal-acap

[3] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems* 4 (2022), 430–449.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[6] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng ChenT, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed WahibT, et al. 2021. Matrix engines for high performance computing: A paragon of performance or grasping at straws? In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1056–1065.

[7] Grattafiori et. al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783. Retrieved from https://arxiv.org/abs/2407.21783

[8] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1342–1352.

[9] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. 2023. ALCOP: Automatic load-compute pipelining in deep learning compiler for AI-GPUs. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.). Curan, 680–694. Retrieved from https://proceedings.mlsys.org/paper_files/paper/2023/file/d6cde2c1b161daa31be560d062cf2251-Paper-mlsys2023.pdf

[10] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 554–566. DOI: https://doi.org/10.1109/ISCA52012.2021.00050

[11] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. Retrieved from https://arxiv.org/abs/1804.06826

[12] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[13] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. Retrieved from https://arxiv.org/abs/1912.03413

[14] Zhe Jia, P. V. Sandt, M. Maggioni, J. Smith, and D. P. Scarpazza. 2021. Dissecting the ampere GPU architecture through microbenchmarking. GTC. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322 (2021).

[15] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: A deep learning optimization framework for versatile GPU workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 190–205.

[16] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A power modeling framework for modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 738–753.

[17] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE Micro* 40, 3 (2020), 20–29. DOI: https://doi.org/10.1109/MM.2020.2985963

[18] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Ferran Alet, Suman Ravuri, Timo Ewalds, Zach Eaton-Rosen, Weihua Hu, Alexander Merose, Stephan Hoyer, George Holland, Oriol Vinyals, Jacklynn Stott, Alexander Pritzel, Shakir Mohamed, and Peter Battaglia. 2023. GraphCast: Learning skillful medium-range global weather forecasting. Retrieved from https://arxiv.org/abs/2212.12794

[19] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. DOI: https://doi.org/10.1109/PROC.1987.13876

[20] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 14–27. DOI: https://doi.org/10.1109/CGO53902.2022.9741270

[21] Sean Lie. 2023. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro* 43, 3 (2023), 18–30.

[22] Justin Luitjens. 2015. Cuda streams: Best practices and common pitfalls. In *GPU Techonology Conference*.

[23] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. RAMMER: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 50, 17 pages.

[24] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.
[25] Meta. [n. d.]. PyTorch. Retrieved from https://pytorch.org/
[26] META. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. Retrieved from https://ai.meta.com/blog/llama-4-multimodal-intelligence/
[27] Microsoft. [n. d.]. D3D12 Work Graphs. Retrieved from https://devblogs.microsoft.com/directx/d3d12-work-graphs/
[28] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2021), 99–106.
[29] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. Retrieved from https://arxiv.org/abs/1906.00091
[30] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. DOI: https://doi.org/10.1145/3453483.3454083
[31] NVIDIA. [n. d.]. cuda::atomic. Retrieved from https://nvidia.github.io/libcudacxx/extended_api/synchronization_primitives/atomic.html
[32] NVIDIA. [n. d.]. Getting Started with CUDA Graphs. Retrieved from https://developer.nvidia.com/blog/cuda-graphs/
[33] NVIDIA. [n. d.]. GPU Management and Deployment: Multi-process service. Retrieved from https://docs.nvidia.com/deploy/mps/index.html
[34] NVIDIA. [n. d.]. GPU Performance Background User's Guide. Retrieved from https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html
[35] NVIDIA. [n. d.]. NVIDIA A100 Tensor Core GPU Architecture. Retrieved from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
[36] NVIDIA. [n. d.]. NVIDIA H100 Tensor Core GPU Architecture. Retrieved from https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
[37] NVIDIA. [n. d.]. NVIDIA Tesla V100 GPU Architecture. Retrieved from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[38] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 41–54.
[39] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 304–315.
[40] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. 2021. Learning mesh-based simulation with graph networks. Retrieved from https://arxiv.org/abs/2010.03409
[41] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. 2022. SambaNova SN10 RDU: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. IEEE, San Francisco, CA, USA, 350–352. DOI: https://doi.org/10.1109/ISSCC42614.2022.9731612
[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel paterns. *ACM SIGARCH Computer Architecture News* 45, 2 (Sept. 2017), 389–402. DOI: https://doi.org/10.1145/3140659.3080256
[43] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2018. Automatic kernel fusion for image processing DSLs. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. 76–85.
[44] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2019. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 242–253.
[45] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of machine learning accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–12.
[46] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Columbus OH USA, 14–27. DOI: https://doi.org/10.1145/3352460.3358302

[47] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

[48] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 909–923.

[49] Jakub M. Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of DNN graph operators. *Advances in Neural Information Processing Systems* 33 (2020), 15451–15463.

[50] Aditya Ukarande, Suryakant Patidar, and Ram Rangan. 2021. Locality-aware CTA scheduling for gaming applications. *ACM Trans. Archit. Code Optim.* 19, 1, Article 1 (dec 2021), 26 pages. DOI: https://doi.org/10.1145/3477497

[51] Han Vanholder. 2016. Efficient inference with tensorrt. In *GPU Technology Conference*, Vol. 1.

[52] Jasmina Vasiljevic, Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic, Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, Utku Aydonat, Tony Zhou, Syed Zohaib Gilani, Armond Paiva, Joseph Chu, Djordje Maksimovic, Stephen Alexander Chin, Zahi Moudallal, Akhmed Rakhmati, Sean Nijjar, Almeet Bhullar, Boris Drazic, Charles Lee, James Sun, Kei-Ming Kwong, James Connolly, Miles Dooley, Hassan Farooq, Joy Yu Ting Chen, Matthew Walker, Keivan Dabiri, Kyle Mabee, Rakesh Shaji Lal, Namal Rajatheva, Renjith Retnamma, Shripad Karodi, Daniel Rosen, Emilio Munoz, Andrew Lewycky, Aleksandar Knezevic, Raymond Kim, Allan Rui, Alexander Drouillard, and David Thompson. 2021. Compute substrate for software 2.0. *IEEE Micro* 41, 2 (March 2021), 50–55. DOI: https://doi.org/10.1109/MM.2021.3061912

[53] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 868–880.

[54] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[55] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. 2020. Accelerating deep learning inference with cross-layer data reuse on GPUs. In *European Conference on Parallel Processing*. Springer, 219–233.

[56] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 358–369.

[57] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 107–118.

[58] Han Zhao, Weihao Cui, Quan Chen, and Minyi Guo. 2022. ISPA: Exploiting intra-SM parallelism in GPUs via fine-grained resource management. *IEEE Trans. Comput.* 72, 5 (2022), 1473–1487.

[59] Jie Zhao, Siyuan Feng, Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyuan Lv, and Qikai Xie. 2023. Effectively scheduling computational graphs of deep neural networks toward their {domain-specific} accelerators. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 719–737.

[60] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.

[61] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {high-performance} tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 863–879.

[62] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 874–887.

[63] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1113–1126.

[64] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

Received 25 March 2025; revised 6 November 2025; accepted 7 November 2025