WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization

Neal C. Crago
NVIDIA
ncrago@nvidia.com

Sana Damani
NVIDIA
sdamani@nvidia.com

Karthikeyan Sankaralingam
NVIDIA
karus@nvidia.com

Stephen W. Keckler
NVIDIA
skeckler@nvidia.com

***Abstract*—Graphics processing units (GPUs) are an important class of parallel processors that offer high compute throughput and memory bandwidth. GPUs are used in a variety of important computing domains, such as machine learning, high performance computing, sparse linear algebra, autonomous vehicles, and robotics. However, some applications from these domains can underperform due to sensitivity to memory latency and bandwidth. Some of this sensitivity can be reduced by better overlapping memory access with compute. Current GPUs often leverage pipeline parallelism in the form of *warp specialization* to enable better overlap.**

However, current warp specialization support on GPUs is limited in three ways. First, warp specialization is a complex and manual program transformation that is out of reach for many applications and developers. Second, it is limited to coarse-grained transfers between global memory and the shared memory scratchpad (SMEM); fine-grained memory access patterns are not well supported. Finally, the GPU hardware is unaware of the pipeline parallelism expressed by the programmer, and is unable to take advantage of this information to make better decisions at runtime.

In this paper we introduce WASP, hardware and compiler support for warp specialization that addresses these limitations. WASP enables fine-grained streaming and gather memory access patterns through the use of warp-level register file queues and hardware-accelerated address generation. Explicit warp to pipeline stage naming enables the GPU to be aware of pipeline parallelism, which WASP capitalizes on by designing pipeline-aware warp mapping, register allocation, and scheduling. Finally, we design and implement a compiler that can automatically generate warp specialized kernels, reducing programmer burden. Overall, we find that runtime performance can be improved on a variety of important applications by an average of 47% over a modern GPU baseline.

I. Introduction

GPUs are the dominant parallel programming substrate with widespread use in deep learning, high performance computing, sparse linear algebra, autonomous vehicles, and robotics [26], [35], [41]. GPU programmers spend considerable time optimizing their code to best exploit available GPU resources. However, some GPU applications are unable to consistently attain high compute throughput or memory bandwidth despite the presence of abundant parallelism [16], [17], [36], [39]. One reason that these applications cannot reach peak performance is due to memory latency and bandwidth sensitivity. This sensitivity generally comes from the inability of the kernel to *overlap* memory accesses with other useful work, causing the resources in the GPU to become underutilized. One way to provide better overlap and reduce memory sensitivity is to refactor the application to exploit pipeline parallelism *within a kernel.*

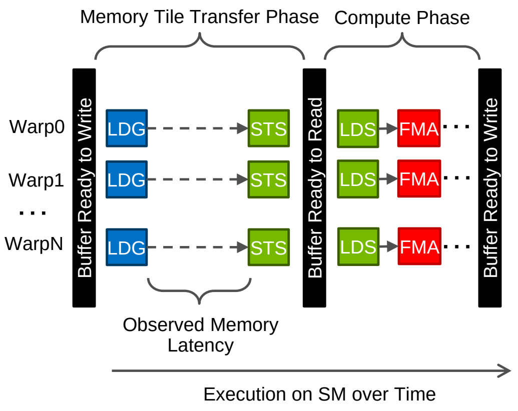
Today, GPUs deploy hardware and software libraries to support pipeline parallelism via the *warp specialization* technique [1], [22], [23]. Warp specialization *specializes* a portion of a GPU kernel to a particular task, creating pipeline stages that can overlap their execution [2], [19], [48]. This technique is commonly used in libraries supporting fast general matrix multiplication, which typically pipeline coarse-grained global memory to shared memory scratchpad (SMEM) transfers with compute operations (e.g., TensorCore) [12]. Despite the broad capability that warp specialization provides, we observe that limitations of this start-of-the-art include: the lack of compiler support to create warp specialized pipelines, the lack of support for fine-grained memory access patterns which also benefit, and the unexploited opportunity of exposing the structure of the pipeline to GPU hardware.

To address these limitations we introduce WASP, an architecture and compiler for accelerating warp specialized pipelines on GPUs. WASP natively supports a variety of fine-grained memory access patterns and explicitly named warp pipeline stages, allowing the GPU hardware to make better runtime decisions. Named queues between pipeline stages are implemented using ISA-exposed register queue operations, supporting fine-grained streaming and gather access patterns. We implement novel warp mapping and scheduling algorithms that use named pipeline stages to take advantage of warp heterogeneity. New hardware address generation units are used to provide additional efficiency through offloading fine-grained data movement operations. Finally, we propose a compiler transformation that automates warp specialization and removes the reliance on libraries and manual programmer effort.

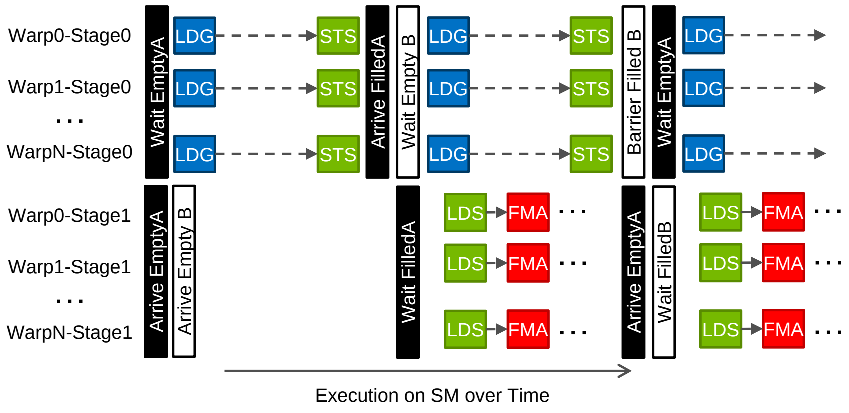
We evaluate the WASP compiler and hardware across a set of CUDA applications spanning a variety of domains. Overall, we find that the WASP hardware and compiler improves runtime performance over state-of-the-art GPUs by 47%.

In summary, we make the following contributions:

- We identify the limitations of existing warp specialization on modern GPUs.



(a) Basic CUDA pattern.



(b) Two-stage warp specialized pipeline.

Fig. 1: Example of a common CUDA pattern: loading a tile of data from global memory to SMEM before performing computation. During the memory tile transfer, observed memory latency is exposed and SM is underutilized (a). When using warp specialization, a two-stage pipeline is extracted that allows overlap of compute and memory data transfer (b).

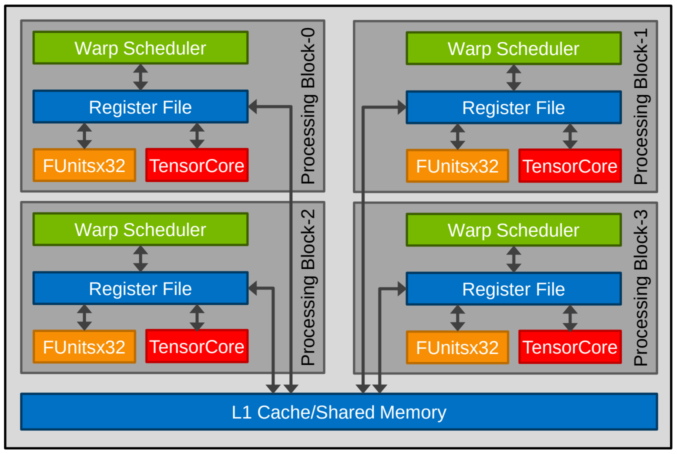


Fig. 2: GPU Streaming Multiprocessor architecture.

- We present explicit warp pipeline stage naming, which we use to implement novel pipeline-aware warp mapping, register allocation, and scheduling schemes.
- We design new hardware for fine-grained memory access patterns, including an enhanced address generation unit.
- We design a compiler that automates warp specialization and reduces programmer burden.
- We demonstrate that these WASP features greatly improve performance across a variety of applications.

II. Background and Motivation

A. GPU Architecture Overview

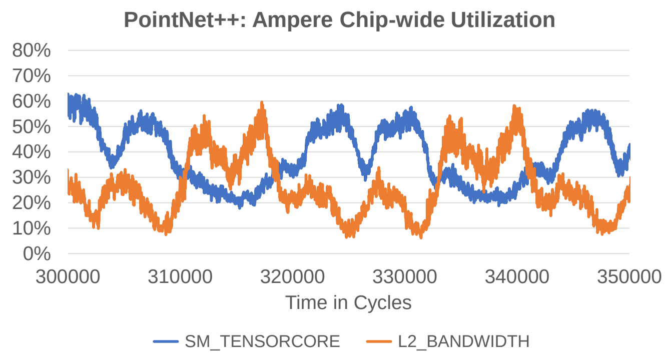
Modern GPUs are highly parallel processors featuring many *Streaming Multiprocessors* (SM) connected to shared on-chip L2 cache memory and off-chip DRAM memory. Figure 2 depicts the organization of an SM, which is made up of a set of processors known as *processing blocks* that share an L1 cache and shared memory scratchpad (SMEM) [22]. A processing block operates as a single-instruction multiple-thread (SIMT) processor, with a single instruction fetched and executed across a vector of threads known as a *warp*. The processing blocks have both per-thread and warp collective functional units (e.g., TensorCore functional unit), a large register file, and *warp scheduler* for managing the execution of a collection of warp contexts. The warp scheduler interleaves the execution of warps on a processing block to hide pipeline stalls. Nvidia's GPUs are programmed by developing *kernels* using the CUDA model and interface [25]. Each kernel is organized as a hierarchy of *threads* which perform a portion of the work of the entire program each. A *thread block* in the CUDA model is a fundamental unit consisting of one or more warps that execute concurrently and cooperatively on an SM.

Today, applications spanning a variety of compute domains take advantage of the high compute and memory throughput that GPUs offer. However, despite abundant parallelism, some applications are sensitive to long memory latencies or have difficulties in overlapping memory access and compute, leading to GPU underutilization [16], [17], [36], [39]. One particularly popular solution to this problem is exploiting pipeline parallelism in the form of *warp specialization*.

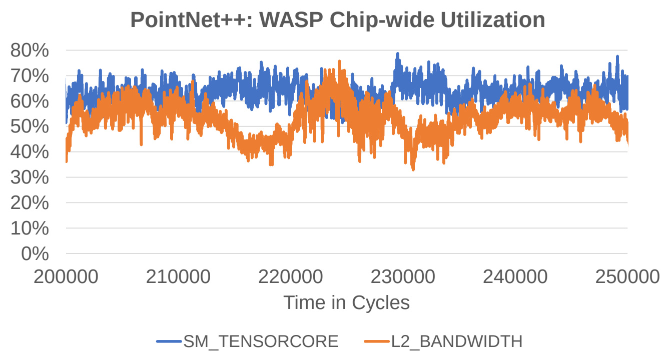
B. Warp Specialization

The warp specialization approach to GPU pipeline parallelism was first described in CudaDMA [1], and is prominently used in CUTLASS [12], the popular state-of-the-art GEMM library used for machine learning. In warp specialization, a single thread block in CUDA is programmed to implement a pipeline on an SM, using *warps* as pipeline stages. These warps are "specialized" and execute a unique part of the entire CUDA kernel. On current GPUs, communication between warps (pipeline stages) is facilitated through SM-level synchronization and data storage. Compared with alternative techniques such as intra-thread software pipelining, warp specialization is a decoupled approach that allows dynamic scheduling and fine-grained interleaving at the warp level.

Figures 1a and 1b present an example of warp specialization in action. Figure 1a depicts the execution of a thread block on a common GPU pattern: transferring memory tiles from global



(a) Ampere Baseline



(b) Ampere with WASP (warp-specialized)

Fig. 3: Pointnet++ GPU chip-wide utilization of SM TensorCore and L2 Cache bandwidth. Comparison of alternating memory-compute phased behavior on Ampere baseline (a) versus more consistent utilization on Ampere augmented with WASP (b).

memory to SMEM shared memory for use in a software-managed buffer. In this scenario, the warps of a thread block work together to synchronize on the state of the buffer and transfer data between the two levels of the memory hierarchy. First, a barrier is reached for the warps in the thread block, indicating that the buffer in SMEM is ready to be written. Next, the warps proceed to issue global memory load (`LDG`) instructions. After the long-latency `LDG` instructions complete, the buffer is written in SMEM using store-shared (`STS`) instructions. Another thread block barrier is used to ensure that all data for the memory tile has been successfully written in the SMEM, and that it is ready for use. Load-shared (`LDS`) instructions are then used freely by the warps to read data from the buffer and perform computation. Finally, a barrier is used to signal that all warps are finished with the buffer, allowing the data to be modified or replaced.

In this common pattern, the compute phase cannot start until the memory tile transfer phase successfully completes, and vice versa. Thus, memory accesses are not overlapped with compute and the strict bimodal nature of the phases ensures that global memory bandwidth and compute functional units (e.g. TensorCore, floating point) are not active at the same time. Figure 1b depicts how warp specialization can be used to create a two-stage pipeline. The original warp is split into two new warps, one for memory access and one for compute. The number of buffers and barriers is expanded to two (A and B) to allow for double buffering. Similar to CudaDMA, arrive/wait barriers are used [1]. On an arrive barrier, the executing warp registers that it has reached the barrier, but continues execution. Going back to the example in Figure 1b, once stage 0 completes filling Buffer A, it arrives at the `FilledA` barrier for that buffer, signaling to stage 0 that the data tile is ready for use. Then stage 0 continues execution by first waiting for Buffer B to be empty and ready to filled (using the `EmptyB` barrier) before issuing the appropriate `LDG` instructions. Thus, the memory tile transfer phase of timestep N+1 overlaps with the timestep N of the compute phase, with warp execution interleaved on the SM. Essentially, more *memory-level parallelism* is extracted using this pipeline parallelism, reducing overall memory sensitivity.

C. Limitations of Current GPUs

Warp specialization is a powerful technique that GPUs adopt today with hardware support for fast arrive/wait barriers and hardware offload units for memory transfers between global memory and SMEM (Hopper incorporates a new Tensor Memory Accelerator, or TMA) [22], [23]. However, current GPU hardware and software support for warp specialized pipelines is quite limited due to three key problems.

First, current GPU hardware is agnostic to pipeline parallelism expressed in software. GPU SMs in particular are designed to execute data-parallel thread blocks as part of the SIMT execution model, using warps that (for the most part) execute identical programs. As such, SM hardware assumes uniform resource utilization. On the other hand, thread blocks with warp specialized pipelines are heterogeneous in nature, with warps requiring different resources. Each pipeline stage implemented by a warp requires a unique program, computational resources (e.g. functional units), register allocation, and in some cases prefers a different warp scheduling priority [1], [2], [12], [19], [48]. If the GPU hardware was aware of the pipeline parallelism expressed, new hardware can be designed to exploit the heterogeneous nature of warp specialized pipelines, leading to better performance.

Next, state-of-the-art GPU hardware supports pipelines with coarse-grained memory tile transfers between global memory and SMEM. While this support is sufficient for some important use cases, such as the GEMM dataflows used in CUTLASS, we find that a larger set of applications could benefit from warp specialization provided support for fine-grained memory access patterns. For example, Figure 3 presents Pointnet++ which performs deep learning on point sets and contains use-once gather and streaming memory operations [33]. Despite abundant parallelism, Pointnet++ struggles to maintain high TensorCore and L2 Bandwidth utilization on Ampere due to the inability to overlap alternating compute and memory access phases (Figure 3a). WASP (this paper) is able to take advantage of the opportunity to overlap the phases and achieve better L2 bandwidth utilization (Figure 3b).

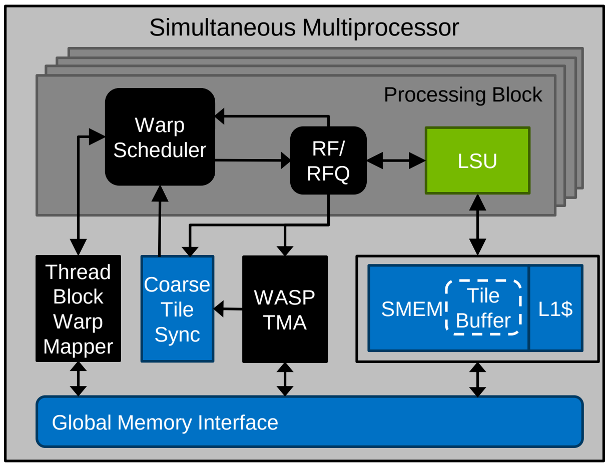


Fig. 4: WASP SM Architecture. Black boxes indicate components that are added or modified over the baseline GPU.

Finally, a major current limitation to warp specialization is that it is a complex optimization implemented manually by the programmer at the CUDA source code level. The programmer is responsible for partitioning the original kernel into a pipeline, assigning threads to pipeline stages, and managing data movement and synchronization between stages. The complexity of the manual optimization limits the number of applications that can exploit warp specialization. However, this optimization is well-structured and suitable for compiler automation, using similar techniques from prior research [9], [21].

In this paper we introduce WASP, new GPU hardware and compiler support to overcome existing warp specialization limitations and improve performance across a variety of applications. WASP enables the structure of a thread block's warp specialized pipeline to be communicated to the SM and more efficiently executed. WASP extends hardware support for fine-grained memory access patterns using a new register file queue and hardware offload unit. Finally, WASP is enabled by a compiler that reduces programmer burden and automatically warp specializes existing CUDA code.

III. WASP Architecture

In this section, we detail how the WASP architecture improves upon two of the limitations found in current GPUs (the third limitation is detailed in Section IV). At the architectural level, WASP manifests as augmentations to a GPU's SM and processing block. Figure 4 depicts the WASP SM architecture, which updates the SM to better take advantage of the available pipeline parallelism found via warp specialization.

First, thread block specifications are updated to explicitly assign warps to pipeline stages (Section III-A), which enables the SM to capitalize on the heterogeneous nature of the warp specialized pipelines. WASP implements new hardware to take advantage of per-pipeline stage behavior in the thread block warp mapper and the warp scheduler, which are crucial mechanisms for properly balancing warp execution. The warp mapper includes support for new per-stage register allocation and pipeline-aware warp-to-processing block mapping (Section III-B). WASP also includes a new warp scheduler to emphasize execution overlap (Section III-D).

Second, WASP adds support for fine-grained streaming and gather memory access patterns to better support the breadth of CUDA applications that can potentially benefit from warp specialization. Architecturally visible queues and associated ISA extensions are added to each processing block's shared register file (Section III-C). These register file queues (RFQs) provide low latency data access and naturally support the use-once streaming and gather patterns not currently supported on GPUs. Finally, we augment the TMA unit in the SM to support streaming and gather memory access patterns, further improving efficiency by reducing the warp address instruction stream executed on the processing blocks (Section III-E).

TABLE I: WASP Thread Block Specification

Specification	Set by	Description
Thread dimensions	Programmer	{x,y,z}
Number of Pipeline Stages	WASP Compiler	Default=1
Pipeline Stage Registers	WASP Compiler	Per-stage
Named Queues	WASP Compiler	{src_id, dst_id, size}
SMEM usage	Programmer WASP Compiler	Original program LDGSTS buffering

A. Explicit Warp to Pipeline Stage Naming

Explicit naming assigns warps to pipeline stages in hardware, allowing the programmer or compiler to specify the configuration of the warp specialized pipelines. At the core, each warp in a thread block is assigned a pipeline stage id. Explicit naming is paramount to performance, as it enables the other pipeline-aware features WASP employs, allowing the SM hardware to be pipeline-aware and make better mapping, register allocation, scheduling, and data movement decisions. To implement explicit naming, WASP extends the thread block specification and adds an additional dimension representing the depth of warp specialized pipeline. Table I presents this new specification, which includes other meta-information generated by the WASP compiler in Section IV. This new dimension is explicitly declared at CUDA kernel invocation time by the programmer or code generation framework. The new thread block dimension specification is in the form:

$$\{dim.x, dim.y, dim.z, num\_pipeline\_stages\}$$

When executing on the SM, a thread is assigned a `pipe_stageId`. This new hardware state is found in the special registers in the SM and can be queried by the threads of the thread block to determine its assigned pipeline stage, similar to querying other thread block parameters such $threadId.x$.

B. Pipeline Aware Warp Mapping and Register Allocation

When a thread block is scheduled to execute on an SM, each warp must be mapped (i.e., assigned) to execute on a processing block. The warp-to-processing block mapping is controlled by a hardware mapper within the SM that first receives a thread block specification from a global GPU scheduler, then distributes the warps of the thread block among the processing

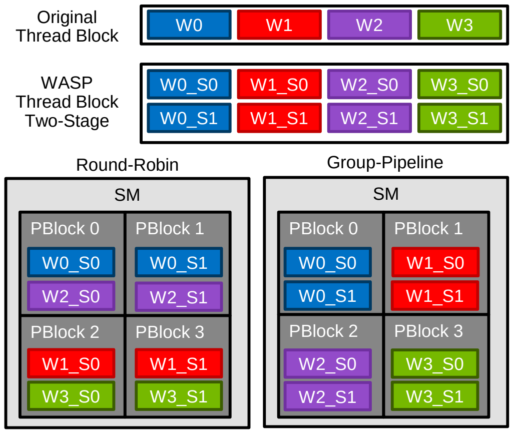


Fig. 5: WASP's group-pipeline warp mapping avoids the imbalance of round-robin where similar stages are mapped to the same processing block.

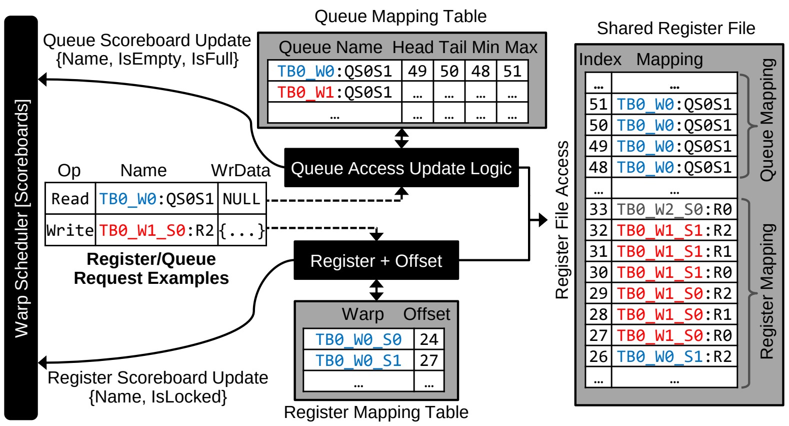


Fig. 6: WASP register file queue. All warps in processing blocks share a physical register file. Hardware support is added to virtualize the queues onto the physical register file.

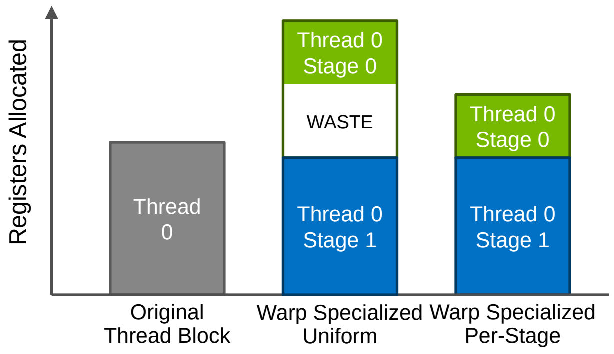


Fig. 7: Register file allocation is uniform across all threads in a baseline GPU. However, register footprint varies by pipeline stage, leading to waste. WASP enables per-stage register allocation, allowing better SM occupancy.

blocks for execution in a round-robin manner. WASP's warp mapper leverages the new information in the thread block specification to improve both the warp-to-processing block mapping algorithm and warp register allocation.

The warp mapping algorithm focuses on balancing computation resource usage by minimizing the mapping of similar pipeline stages to the same processing block. The algorithm `group_pipeline` maps all warps belonging to the same warp specialized pipeline slice together on the same processing block. Figure 5 demonstrates the potential value of the WASP mapping algorithm over the original round-robin mapping algorithm. The WASP thread block consists of a two-stage warp specialized pipeline, with each stage having four warps. Imagine the scenario in which $S0$ is a memory access stage and $S1$ is a compute stage. The round-robin algorithm maps warps one at a time alternating across the processing blocks, leading to imbalanced execution with $S0$ stages mapped to processing blocks 0 and 2, and $S1$ stages mapped to processing blocks 1 and 3. The `group_pipeline` algorithm provides better balance by executing the warps of each pipeline slice on the same processing block.

During the warp mapping process, register allocation for the warp is performed by carving out a contiguous segment of the processing block's shared register file. In warp specialization, different pipeline stages run different programs and therefore use a different number of registers. However in current GPUs, warp specialized pipelines are allocated a uniform value for warp register allocation which is the *maximum* usage among all pipeline stages. Among the kernels we evaluated, some pipeline stages require nearly the same number of registers per thread as the original program, which can result in a nearly N times increase in thread block register usage for an N-stage pipeline. WASP uses the per-pipeline-stage register values from the thread blocks specification in Table I to allocate *uniquely-sized* register segments, which saves considerable register file space and is a improvement over state-of-the-art. Figure 7 presents an example comparison for the register per thread allocation among the original program, a warp specialized program on existing GPUs, and the register savings provided by WASP. More data on register allocation savings can be found in Section V-D.

C. Register File Queues

While current GPU hardware for warp specialization supports coarse-grained memory tile data movement between global memory and SMEM, some memory-sensitive CUDA applications heavily utilize use-once data. This use case manifests as fine-grained streaming and gather memory access patterns, which WASP supports by using a queue between pipeline stages. In the queue design for WASP, we investigated where the data storage and synchronization of data queues should occur. One option is to simply use the same software mechanisms used for coarse-grained memory tiles: SMEM double buffering and the hardware synchronization unit.

However, managing data in and out of SMEM results in execution overheads. Each original global load instruction is converted into a load-global store-shared `LDGSTS` instruction within the producer stage to store into the buffer, which require additional address generation instructions and control instructions for keeping track of whether the buffer is full or not. The consumer stage's code is similar, with the exception of requiring a shared memory load (`LDS`) instruction to read the queue. When the buffers are full or empty, the producers and consumers must notify each other, using the appropriate arrive/wait barriers. While for coarse-grained memory transfers this overhead is amortized by data reuse, there is no amortization in the use-once streaming and gather memory access patterns. After evaluation we find that these overheads are high and impact performance (Section V-C) .

With these considerations in mind, we architect hardware queues in WASP by mapping them as circular buffers in the existing register file space. Figure 6 depicts how register file queues (RFQs) integrate into the SM processing block. Similar to per-warp register allocation, *named* queues connecting pipeline stages are explicitly defined in the thread block specification and allocated in the physical register file at warp mapping time. For example, $TB0\_W0\_S0S1$ specifies that thread block 0 has a named queue connecting stages 0 and 1 of original warp 0. Each named queue is supported by an accompanying hardware table to maintain the state of the queue, which includes the head pointer, tail pointer, queue allocation start index, and queue allocation end index.

We extend the GPU's ISA to use these names directly as warp instruction operands, similar to registers. For example, when $TB0\_W0\_S0$ executes the following warp instruction:

$$LDG\ Q1, [RX]$$

A decoupled load to global memory is issued, sending data to $TB0\_W0\_S1$ using the $TB0\_W0\_QS0S1$ queue. Similarly, $TB0\_W0\_S1$ can execute the following warp instruction to read the data from the $TB0\_W0\_S1$ queue:

$$MOV\ RY, Q0$$

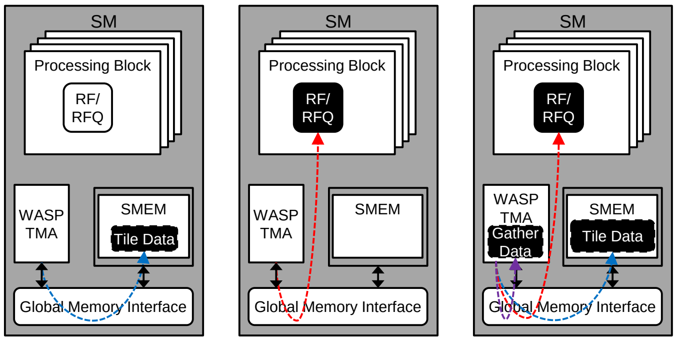
An RFQ scoreboard sitting at the warp instruction scheduler supports whether there is either data in the queue (i.e., on a read) or if there is room in the queue for more data (i.e., a write). These scoreboard bits `is_empty`, `is_full` are updated on successful reads or writes to an RFQ.

D. Pipeline Aware Warp Scheduling in the SM

We also augment the existing hardware warp scheduler within the processing blocks with a new pipeline-aware prioritization policy that promotes execution overlap. The existing hardware warp scheduler stores information about the state of each warp (e.g., waiting on long/short scoreboard) which is used to determine its scheduling priority. In WASP, each hardware warp scheduler is augmented to store the pipeline stage id and incoming data queue status (e.g., queue is not empty, queue is full) for each warp, which act as additional inputs to the scheduling decision making algorithm. Using these new inputs, we implement novel warp scheduling policies that leverage the pipeline stage meta information for prioritization.

The first policy uses the pipeline stage id input to prioritize a warp based on which stage it is in the pipeline. Our naming convention defines `pipe_stageId_0` as the first stage in the pipeline, and `pipe_stageId_N-1` as the final stage in a N-stage pipeline. Given a set of warps executing on an SM processing block, WASP provides the ability to prioritize earlier stages by increasing order of the value of their `pipe_stageId`. We find that prioritizing earlier stages of the pipeline is generally a good idea, as these stages tend to be memory access stages with long memory latencies we want to overlap.

The second policy builds upon the first and uses the RFQ status input to prioritize which warp to schedule. For each warp, we keep the following bits: `is_empty`, `is_full`. Our hardware warp scheduler uses the `is_empty`, `is_full` RFQ status bits to prioritize consumer warps when data is ready. These two policies are combined in our augmented hardware warp scheduler. In Section V-E, we find that prioritizing warps with full queues, followed by warps with ready buffers or nonempty queues, followed by earlier stages of the pipeline performs best.



(a) Global-SMEM access pattern. (b) Streaming access pattern. (c) Gather access pattern.

Fig. 8: WASP-TMA data movement patterns.

E. WASP-TMA: Accelerating Address Generation

TMA is an offload accelerator for coarse-grained memory tile transfers between global memory and SMEM. TMA uses new instructions to specify the global memory block to transfer, using meta information such as base address, stride, offset, and the dimensions [23], [24]. Offloading memory access instruction streams to TMA achieves two benefits. First, hardware acceleration reduces warp instruction issue slots and registers, allowing more resources in the SM processing block for computation, such as TensorCore operations. Second, leveraging specialized hardware generates accesses more efficiently, reducing energy consumption. WASP augments TMA with the ability to handle fine-grained memory accesses at the thread granularity, using hardware similar to prior ISA extension proposals [8]. Figure 8 presents the patterns that WASP supports, which includes memory tile transfers between global memory and SMEM (Figure 8a), streaming data between global memory and RFQs (Figure 8b), and gather operations between global memory and SMEM/RFQ (Figure 8c).

A TMA-like `global-SMEM` instruction is used to move coarse-grained data between global and SMEM, using arrive-wait barriers for synchronization. A new `global-RFQ` configuration is added to target a named RFQ instead of SMEM. Fine-grained synchronization is accomplished using the status table for the RFQ. Typical decoupled `LDG` instructions writing a register file queue acquire a single entry in queue. WASP-TMA global-RFQ instructions acquire multiple entries, delaying issue until they are available.

The gather access pattern ($C[i] = B[A[i]]$) focuses on minimizing data movement and effectively fuses two operations together. The WASP-TMA (`gather-SMEM`) and (`gather-RFQ`) instructions first generate a gather memory access stream to an array of indices in global memory. As the gather indices arrive at global memory, they are consumed by WASP-TMA and processed in a second phase. The second phase combines the gather indices with a base address to generate a memory request stream that either targets SMEM (`gather-SMEM`) or the RFQ (`gather-RFQ`). Incoming indices are held in a ping pong buffer with two entries, one for the set of indices currently being processed by TMA and another entry for receiving a new set of indices in the same cycle. By not writing the gather indices back to SMEM, WASP-TMA eliminates extra RFQ and SMEM traffic that would otherwise be required.

IV. WASP Compiler Design

Warp specialization is a non-trivial transformation currently implemented manually either by the programmer or using optimized libraries like CUTLASS for GEMMs [12]. WASP removes this limitation through a programmer-directed compiler transformation that automatically generates warp specialized CUDA kernels to run on a WASP-enabled GPU. Our compiler performs binary recompilation, using Nvidia's SASS assembly from `nvdisasm` [24]. We also make simplifying assumptions to aggressively exploit the GPU's weak consistency model: that thread block synchronization is used only for SMEM transfers, and that memory fences are not used. The programmer is expected to ensure that these two assumptions are true for their kernel before enabling the warp specialization transformation. While our implementation is a source-to-source compiler, other works for decoupled access execute processors show such ideas can be embedded into mainstream GPU compilation infrastructure [21], [43]. The target output for the compiler is a SASS program that runs on our WASP GPU performance model (Section V-A). We first extract potential pipeline stages using a program dependence graph (PDG) (Section IV-A), then finalize the warp specialized pipeline as a SASS program (Section IV-B). Detailed examples of how each memory access pattern is transformed are found in Section IV-C.

A. Pipeline Stage Extraction

Similar to past work, the WASP partitioning process focuses on overlapping long-latency memory accesses with compute by breaking the original program into pipeline stages using global memory load-use boundaries [9], [10], [20], [40], [47].

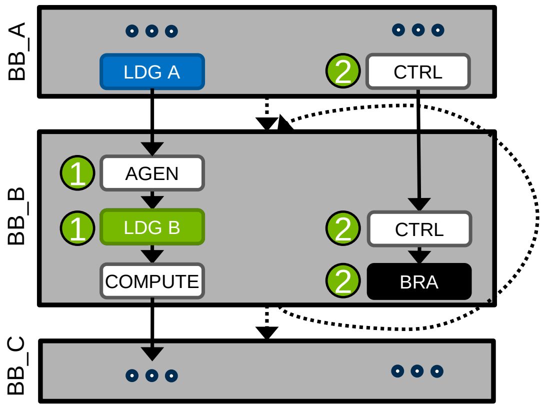


Fig. 9: Example of two-phase pipeline stage extraction.

First, we construct a program dependence graph (PDG) of the original SASS program. Our PDG maintains control flow edges and data dependencies between instructions of the program. Next, we partition the PDG into warp pipeline stages using a pipeline stage extraction scheme similar to OUTRIDER [9]. We identify all global memory load instructions (`LDG`) within the kernel and determine which are eligible for pipeline stage extraction using the instruction's backslice. The backslice includes all instructions in the original program that the `LDG` directly depends on and the instructions contributing to program control flow that the `LDG` indirectly depends on. A backslice containing SMEM load instructions (`LDS`) instructions indicates that the `LDG` may have memory dependencies to SMEM store (`STS`) instructions that cannot be tracked and is therefore not eligible for pipeline stage extraction. Similarly, a backslice containing a dependency cycle of the `LDG` with itself are also excluded. We find in practice that both situations are not common in GPU code.

Each eligible `LDG` instruction is split into two new instructions at the load-use boundary: `LDG_PRODUCER` for the address, and `LDG_CONSUMER` for the result of the load. As a special case, we identify `LDG` instructions that are only connected to SMEM stores and combine into a single load-global store-shared (`LDGSTS`) instruction. These `LDGSTS` instructions require specific synchronization as described later.

In the first extraction phase, an initial new stage is created for each `LDG_PRODUCER` and `LDGSTS` using a collection of required instructions and basic blocks. The backslice of address generation instructions for each `LDG_PRODUCER` and `LDGSTS` is identified and added to the collection by traversing the PDG. The backslice depth-first search terminates at either dependence chain endpoints or upstream `LDG_PRODUCER` instructions. Figure 9 depicts that the address generation backslice for `LDG B` terminates at `LDG A` and contains two instructions ((1)).

The second extraction phase completes the new stage's PDG subgraph by adding the minimum basic blocks and associated control flow instructions. For each instruction in the stage's collection, the parent basic blocks that flow into the

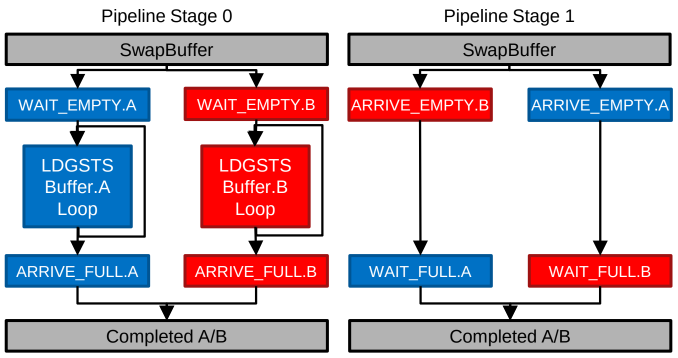


Fig. 10: Example of pipeline stage SMEM double buffering.

instruction's assigned basic block are added to a search list. For each unvisited basic block in the list, the block is marked visited and checked to see if it contains a branch instruction. If there is a branch, a backslice of instruction for the branch is generated and added to the collection. The search process continues until all instructions in the collection have been examined, signaling the collection represent all the instructions required for executing the stage. A PDG subgraph is generated using the instructions, keeping them in the original program order. Figure 9 shows that the address generation instructions in (1) are assigned to basic block B (`BB_B`) and parents `BB_A` and `BB_B` are added to the search list. `BB_A` has no branch, while `BB_B` has a branch to backslice. The instructions in (2) are added to the stage's collection, and the process continues. Basic blocks `BB_A` and `BB_B` have already been visited, so the second extraction phase completes.

An optimization pass is performed at the end of the second phase to detect if the `LDG` or `LDGSTS` instruction is contained in a loop that can be offloaded to WASP-TMA. If the loop and address generation pattern is suitable, the identified control flow and address generation instructions within the loop are replaced with a WASP-TMA configuration instruction.

B. Pipeline Finalization and Configuration

When all eligible `LDG_PRODUCER` instructions have been processed and all stages extracted, a final program representing the warp specialized pipeline is generated. First, there is a set of original program instructions that remain unallocated. This set of instructions represents the compute belonging to the final pipeline stage, including all shared memory and global store instructions. These instructions are allocated to a new stage, and remain in program order. Next, stages are optionally merged depending on the whether the number of stages extracted can be supported on the SM. We find that optimized SASS for CUDA kernels can have tens to hundreds of static `LDG` instructions that are extracted into stages. Considering the original thread block dimensions of CUDA applications we studied, such a warp specialized kernel will not fit on the SM. To reduce this large number of extracted stages, we use a scheme from prior work that merges memory access stages with similar level of memory indirection [9]. We find that this scheme performs well while fitting within the resources of the SM.

After this optional merge step is complete, the stages are given explicit pipeline stage names based on their natural order. For a pipeline of size N, stage zero is always the earliest stage and stage $N-1$ the last stage. The thread block specification is then updated with the number of pipeline stages. Table I depicts the thread block specification table that is populated during this final process.

The communication and synchronization between stages is also finalized. We handle stages with `LDG` and `LDGSTS` instructions differently. For `LDG_PRODUCER` and corresponding `LDG_CONSUMER` instructions the RFQ is used, and a new named queue connecting the two stages is added to the thread block specification with a fixed size. The `LDG_PRODUCER` and corresponding `LDG_CONSUMER` instructions are updated to use the named queue, and the `LDG_CONSUMER` instruction is optionally merged into the dependent instruction if there is only a single dependent instruction.

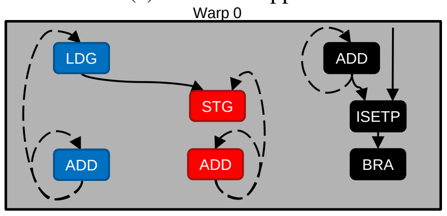
Stages with `LDGSTS` instructions must synchronize with the compute pipeline stage to ensure correct behavior. As seen in Figure 1b, this implies inserting coordinated arrive/wait barriers. Our compiler automates the process found in CudaDMA for both single and double buffering. For both cases, we first identify a pair of `BAR.SYNC` barrier instructions that enclose the `LDGSTS` by inspecting the PDG. For single buffering, each `BAR.SYNC` instructions is replaced with an arrive/wait barriers at the same program location in the two warps using `BAR.WAIT` and `BAR.ARRIVE` instructions (example provided in Section IV-C).

Double buffering is more complex and requires doubling the SMEM buffer and arrive/wait barriers [1]. We use the address backslice of the `LDGSTS` instruction and the SMEM allocation information from `nvdisasm` to identify which shared buffer is used. A check is performed to determine if there is SMEM capacity available for double buffering, before resizing and applying the transformation. Figure 10 presents how the program uses this larger buffer to achieve the result in Figure 1b. The subprogram for the global-to-SMEM transfer between the two enclosing barriers is replicated to create two new subprograms accessing different halves of the SMEM buffer, with each using unique sets of arrive/wait barriers (`A` and `B` in Figure 10). New basic blocks (`swap_buffer`, `completed`) are added before and after the subprograms with instructions that switch execution between the two buffers by alternating the base address. For the later pipeline stage (Stage 1 in Figure 10), barrier `A` is initially set as arrived.

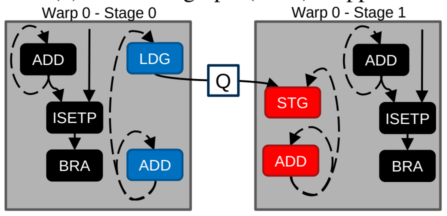
Finally the programs for the pipeline stages are combined to create a complete SASS program. First register re-allocation is performed; the WASP compiler performs a simple reallocation by compacting the registers into contiguous space. The thread block specification is updated to assign register allocation sizes per stage (Table I). The pipeline stages are then written sequentially to a final SASS program. A *jump table* is added to the top of the program to direct each warp to the appropriate code section, using the special register for WASP's explicit naming (Section III-A).

```
int* in; //global memory
int* out; //global memory
...
int start= (…) + threadId.x;
int end = start + warp_transfer_size;
for (int i=start; i<end; i+=WARP_WIDTH)
    out[i] = in[i];
...
```

(a) CUDA Snippet.



(b) Dataflow graph (DFG) snippet.

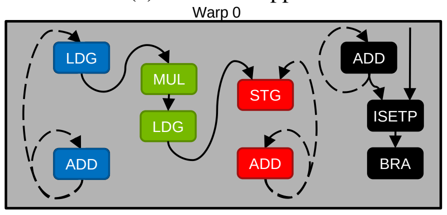


(c) WASP DFG transformation.

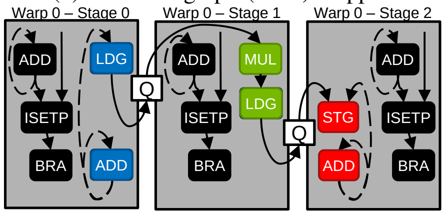
Fig. 11: Streaming pattern.

```
int* index; //global memory
int* data;  //global memory
int* out;   //global memory
...
int start= (…) + threadId.x;
int end = start + warp_transfer_size;
for (int i=start; i<end; i+=WARP_WIDTH)
    out[i] = data[ index[i] ];
...
```

(a) CUDA Snippet.



(b) Dataflow graph (DFG) snippet.

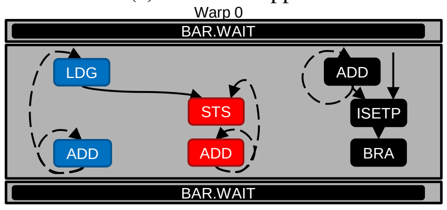


(c) WASP DFG transformation.

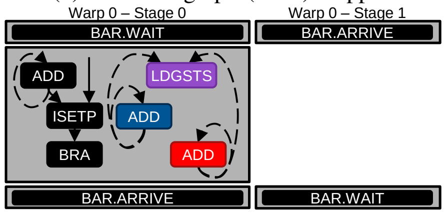
Fig. 12: Gather pattern.

```
int* in; //global memory
__shared__ int buffer[];
...
__syncthreads(); //sync buffer ready
int start= (…) + threadId.x;
int end = start + warp_transfer_size;
for (int i=start; i<end; i+=WARP_WIDTH)
    buffer[i] = in[i];
__syncthreads(); //sync transfer done
...
```

(a) CUDA Snippet.



(b) Dataflow graph (DFG) snippet.



(c) WASP DFG transformation.

Fig. 13: Global-to-SMEM pattern.

C. Memory Access Pattern Examples

Figures 11, 12, and 13 depict examples of generating of warp specialized pipelines for the streaming global-to-RFQ, gather global-to-RFQ, and global-to-SMEM patterns. Each of examples focuses on the inner loop of a CUDA kernel, and depicts an instruction-level dataflow graph for the basic block of that inner loop.

Figure 11 shows a streaming global-to-RFQ example for copying an input vector to an output vector. In the inner loop, the warps execute a global load instruction (`LDG`) that forwards data into a global store instruction (`STG`). Both the `LDG` and `STG` instructions have their own address generation instructions (`ADD`), and the inner loop has instructions that manage the control flow (black). Considering the single `LDG` instruction in the inner loop, the WASP kernel consists of two pipeline stages ($warp0\_stage0$ and $warp0\_stage1$). The backwards slice of instructions contributing to the `LDG` address are allocated to the earlier pipeline stage, and the remaining instructions to the later pipeline stage. The control flow instructions for the basic block are replicated across both warps, to maintain coherent execution between the two stages. Finally, the `LDG` destination operand is connected to the `STG` source operand using the `QS0S1` named queue.

Figure 12 shows how a gather global-to-RFQ operation leads to an additional pipeline stage. The CUDA inner loop reads from global memory array `index` and uses the result as an index into the `data` array. This indirection manifests in the DFG as an additional `LDG` and address generation. Similar to the prior global-to-RFQ example, both `LDG` instructions of the original inner loop are used as the point of partitioning. Three total stages are generated, with replicated control flow instructions queues used to connect the pipeline stages together.

Figure 13 shows how WASP generates code for global-to-SMEM memory tile transfers using single buffering. In the original CUDA program, shared memory is allocated for the `buffer` array that is shared among all threads in the thread block. The `__syncthreads()` function is used to synchronize on the state of `buffer` before and after the transfer. The dataflow graph looks otherwise similar to the streaming copy example, with the only exception a SMEM store instruction (`STS`) replacing the `STG`. First, the `LDG` and `STS` instructions are fused into a single `LDGSTS` instruction, and the appropriate address generation and control flow instructions are allocated to the earlier pipeline stage $warp0\_stage0$. The $warp0\_stage1$ is empty of instructions in the inner warp as there is no compute. Finally, the `BAR.WAIT` and `BAR.ARRIVE` instructions are inserted to synchronize the threads, similar to the example in Figure 1b.

V. Evaluation

A. Experimental Setup

We evaluate WASP using a modified version of NVIDIA's NVArchSim (NVAS), a hybrid trace- and execution-driven GPU simulator [45] that has been validated against NVIDIA's Ampere GPU. Table III summarizes the configuration we use to evaluate WASP. We augment an Ampere A100 [22] model with support for fast arrive/wait barriers and a TMA-like accelerator to model modern features utilized by warp

TABLE II: Benchmarks and Median / Maximum Kernel Speedups with WASP

Name	Category	# Unique Kernels	# Total Kernels	% cuBLAS / GEMM	Median / Max Kernel Speedup	Description
3d_unet	ML/Robotics	19	60	45%	1.14x / 6.92x	Dense Volumetric Segmentation [35]
bert	ML/Robotics	9	172	56%	1.14x / 1.51x	Encoder Transformer Network [35]
curobo	ML/Robotics	2	6	0%	1.43x / 1.64x	Kinematics for robot motion planning [41]
dlrm	ML/Robotics	7	9	56%	1.17x / 1.33x	Deep learning recommendation model [35]
gpt2	ML/Robotics	39	574	17%	1.19x / 5.20x	Generative Pre-trained Transformer [34]
pointnet	ML/Robotics	1	1	0%	1.42x / 1.42x	Deep learning point set segmentation [28], [33]
rnnt	ML/Robotics	8	16	0%	1.56x / 5.01x	Recurrent neural network [35]
spmv1_g3	cuSPARSE	2	2	0%	1.02x / 1.11x	Sparse matrix dense vector multiply [4], [26]
spmv2_web	cuSPARSE	2	2	0%	1.01x / 1.10x	Sparse matrix dense vector multiply [7], [26]
spmm1_g3	cuSPARSE	1	1	0%	1.15x / 1.15x	Sparse matrix dense matrix multiply [4], [26]
spmm2_web	cuSPARSE	1	1	0%	3.53x / 3.53x	Sparse matrix dense matrix multiply [7], [26]
spgemm1_econ	cuSPARSE	13	30	0%	1.05x / 1.53x	Sparse matrix sparse matrix multiply [6], [26]
spgemm2_road	cuSPARSE	13	30	0%	1.06x / 2.35x	Sparse matrix sparse matrix multiply [5], [26]
hpcg	HPC	1	2	0%	1.17x / 2.25x	Multigrid conjugate gradient [27]
hpgmg	HPC	14	317	0%	1.67x / 11.25x	Geometric multigrid linear solver [14]
lulesh	HPC	14	6,537	0%	1.35x / 2.79x	Hydrodynamics simulation [18]
snap	HPC	7	3,254	0%	1.24x / 1.69x	Particle transport [32]
lonestar_bfs	Graph	3	289	0%	3.61x / 4.06x	Breadth-first search [3]
lonestar_mst	Graph	6	37	0%	1.34x / 3.39x	Minimum spanning tree [3]
lonestar_sp	Graph	3	11	0%	1.94x / 2.43x	Survey propagation [3]

TABLE III: NVArchSim A100+ Configuration

SMs	108
Processing Blocks	4 per SM
Register File	256KB per SM
L1 Cache/SMEM (KB)	192
L2 Cache (MB)	40
Warp scheduling scheme	Greedy then oldest (GTO)
Warp Specialization	Hardware arrive/wait barriers TMA-like offload accelerator
WASP	32-entry RFQ per warp Pipeline-aware warp mapping & scheduling Per-stage register allocation WASP-TMA offload accelerator 16 maximum stages

specialized pipelines [23]. For WASP, we also model the new thread block specification, thread block RFQs, pipeline-aware warp mapping, register allocation, warp scheduling, and the WASP-TMA offload accelerator. For software, we automatically generate configurations using our compiler tool. We direct the compiler on a per-kernel basis whether to use warp specialization or not, depending one whether such a transformation improves performance over the baseline. The SASS programs and thread block configurations from the NVAS trace are transformed into warp specialized versions utilizing WASP hardware.

We evaluate WASP on a variety of benchmarks (Table II), which were selected after determining they benefit from warp specialization. The benchmarks come from MLPerf [35], cuSPARSE [26], machine learning, robotics, and high-performance computing and are fully-optimized. These benchmarks often represent full applications with many unique and dynamically executed kernels. Across the kernels of a benchmark, Table II presents the median and maximum kernel speedup achieved by WASP. We find that many kernels experience speedups greater than 2× with the combination of WASP compiler and hardware support, with a maximum achieved speedup of 11.25×. Our baseline GPU models CUTLASS warp specialization on the GEMM and cuBLAS kernels of the benchmarks by utilizing the WASP-compiler for coarse-grained tile transfers and idealized warp mapping.

B. Overall WASP Hardware and Compiler Performance

Figure 14 presents the overall speedup of WASP over the baseline GPU model that uses CUTLASS warp specialization (**BASELINE**). We first evaluate the WASP compiler in isolation on the baseline with two versions: one that only supports coarse-grained transfers between global and SMEM (**WASP_COMPILER_TILE**), and one that also warp specializes for streaming and gather memory transfers (**WASP_COMPILER_ALL**). In isolation, the WASP compiler configurations cannot take advantage of pipeline-aware hardware and require the use of SMEM queues for the newly support memory access patterns. We model SMEM queues as described in Section III-C. Finally, our last comparison point is the WASP GPU hardware targeted by the WASP compiler (**WASP_GPU+WASP_COMPILER_ALL**).

WASP capitalizes on warp specialization by implementing a compiler to automatically generate pipeline parallelism and reduce programmer burden. We find that **WASP_COMPILER_TILE** alone is able to improve performance over state-of-the-art libraries like CUTLASS (**BASELINE**) by transforming the kernels that are not otherwise warp specialized (Table II). However, the lack of compiler support for streaming and gather memory access patterns, managing warp mapping, and lack of hardware features to improve occupancy (e.g. per-thread register allocation) limits the performance gains of **WASP_COMPILER_TILE**. For benchmarks with that can exploit our CUTLASS model (`3d_unet`, `bert`, and `gpt2`), **WASP_COMPILER_TILE** improves runtime by 4%. **WASP_COMPILER_TILE** improves runtime by 3% on the other benchmarks that are not able to capitalize on CUTLASS as they do not feature GEMMs. Benchmarks with kernels that exhibit significant opportunity to overlap coarse-grained memory transfer, such

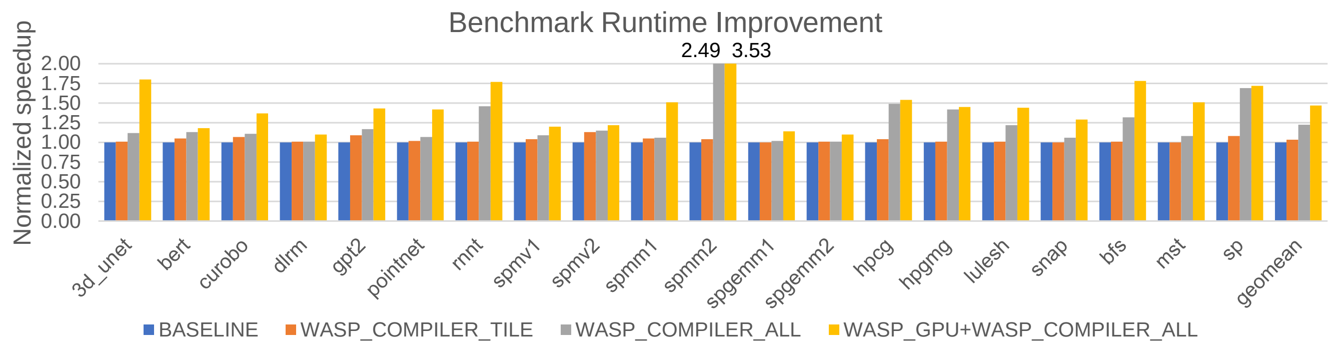


Fig. 14: Performance comparison of WASP GPU and Compiler to modern GPU baseline.

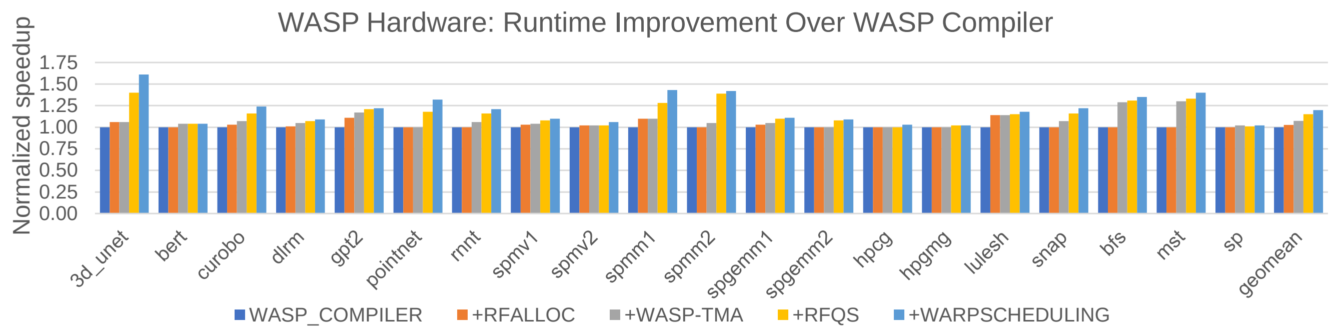


Fig. 15: Performance improvement of WASP GPU when adding new hardware features progressively.

as `gpt2`, `sp`, and `spmv2_web` see runtime improvements greater than 10%.

We find that there is much unexploited opportunity for warp specialization on gather and streaming memory patterns in these benchmarks. Across the 20 benchmarks, **WASP_COMPILER_ALL** is able to improve runtime over the baseline by greater than 10% on 12 benchmarks, by up to 149% and 23% geometric mean via better overlapping of compute with memory access streams. Five memory-latency sensitive benchmarks with frequent streaming and gather memory accesses (`rnnt`, `hpcg`, `hpgmg`, `sp`, and `spmm_web`) improve their runtime performance over **WASP_COMPILER_TILE** by more than 40% on average. However, eight of the benchmarks do not see significant benefit with the WASP compiler alone.

WASP_GPU+WASP_COMPILER_ALL provides augmented GPU hardware that synergizes with the WASP compiler to improve run-time performance by over 10% for all benchmarks. On average, **WASP_GPU+WASP_COMPILER_ALL** improves performance over **BASELINE** by 47%, and over the WASP compiler alone by 23% mean. Compute heavy machine learning benchmarks like `3d_unet` (80%), `bert` (18%), `gpt2` (43%), and `pointnet` (42%) improve performance over the CUTLASS baseline by more naturally overlapping gather and streaming memory access patterns. Other memory-sensitive benchmarks like `spmv` and `spmm` are better able to saturate DRAM and L2 memory bandwidth. Graph algorithms `bfs` (78%), `mst` (51%), and `sp` (72%) heavily leverage the WASP-TMA support for streaming and gather memory access patterns, reducing total dynamic instructions and extracting more memory-level parallelism to better utilize memory bandwidth.

C. *Performance impact of WASP hardware features*

WASP adds several hardware features to the GPU to better exploit the pipeline parallelism found in warp specialization. Specifically, WASP adds pipeline-aware warp scheduling, register file allocation, register file queues, and WASP-TMA. Figure 15 presents the runtime performance of each of these features added progressively to the baseline GPU.

Pipeline-aware warp register allocation reduces pressure for register file resources, allowing more kernel thread blocks to concurrently execute on the same SM, while also enabling larger RFQs. We find that better register allocation provides a 4% geometric mean runtime improvement across the benchmarks. Section V-D details additional analysis of register footprint savings provided by WASP.

The main benefit that WASP-TMA provides is the reduction in dynamic instructions, which frees up processing block instruction issue for other useful work. We find that instruction-issue limited benchmarks like `bfs` (+29%) and `mst` (+30%) benefit the most. WASP-TMA provides an additional 2% geometric mean runtime improvement across the remaining benchmarks. Further analysis of WASP-TMA and dynamic instructions is in Section V-G.

The use of SMEM queues required by software-only implementations (**WASP_COMPILER_ALL**) increases SMEM

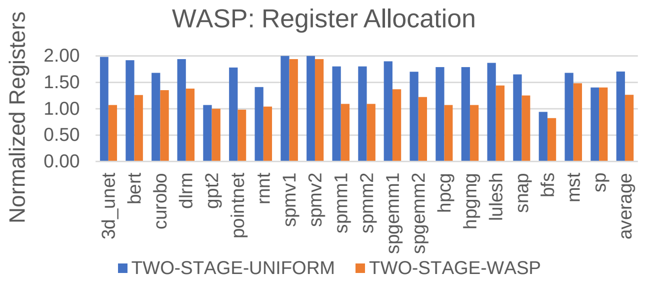


Fig. 16: Register footprint per thread block for uniform thread allocation and per-stage allocation (WASP) compared against a non-warp specialized baseline.

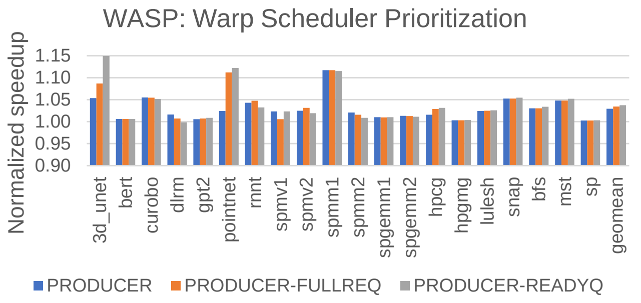


Fig. 17: Performance impact of pipeline-aware warp scheduling over a greedy-then-oldest baseline.

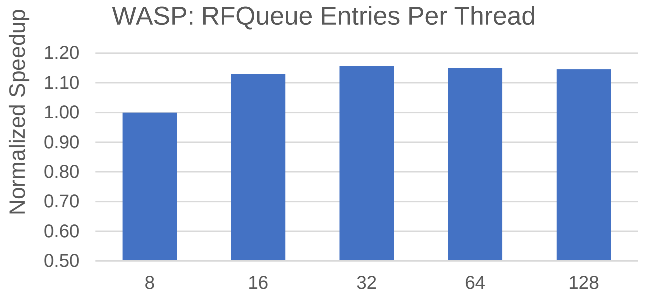


Fig. 18: Average performance improvement when varying the size of the register file queues.

bandwidth consumption and processing block instruction utilization (Section III-C). RFQs remove the expensive overhead of SMEM queues and reduce activity and contention for SM resources. `3d_unet` (+32%), `pointnet` (+18%), and `spmm1_g3` (+16%) `spmm2_web` (+32%) are SMEM bandwidth-sensitive and see more than 15% better performance improvement from RFQs, while the rest of the benchmarks are not sensitive and see an additional 4% mean improvement.

Finally, pipeline-aware warp scheduling provides an additional 4% mean runtime improvement across the benchmarks. The WASP warp scheduling policies prioritize the warps of earlier memory access stages to promote execution overlap. Benchmarks that are more imbalanced between memory access and compute instruction streams particularly benefit from improved warp scheduling. For example, `3d_unet` (+15%), `pointnet` (+12%), and `snap` (+6%) execute many more instructions in the compute stage of their pipelines, and the WASP policies ensure that earlier stages have plenty of instruction slots to keep the pipeline filled. More detailed analysis of the warp scheduling policies WASP uses is in Section V-E.

D. Pipeline-aware Register Allocation

Figure 16 presents the register allocation for thread blocks, comparing uniform register allocation to the per-stage register allocation enabled by WASP. This allocation does not include registers for RFQs, but does include register savings by offloading memory accesses to WASP-TMA. For each benchmark, the kernel contributing the most to total runtime is chosen, which varies between 33% and 100% depending on the benchmark. We find that more than half of these kernels are very unbalanced and heavily skew their register allocations towards compute stages. As a result, many of these kernels have nearly the same number of registers in the compute stages as in the original non-warp specialized kernel. When uniform register allocation is performed, 12 of the kernels increase total thread block register footprint by more than 75%, with `3d_unet`, `bert`, `dlrm`, `spmv`, and `spgemm1_econ` requiring a register footprint nearly double in size. When WASP's per-stage register allocation is enabled, the register foot print for the kernels decreases by 44% on average, allowing more registers in the SM to be used for RFQs and other thread blocks. The resulting register footprint overhead is 26% over the non-warp specialized baseline.

E. Pipeline-aware Warp Scheduling

Figure 17 presents detailed results comparing four classes of pipeline-aware schedulers to the baseline greedy-then-oldest scheme. Among the schemes presented, **producer**-first schemes prioritize based on earlier pipeline stage and **full/ready** queue schemes prioritize warps with full or ready queues first, then earlier pipeline stages. In general, prioritizing earlier pipeline stages performs best, due to the policy promoting memory level parallelism and thereby better overlap. Some benchmarks such as `3d_unet` and `pointnet` are especially unbalanced between computer and memory access stages, preferring policies leveraging queues status to perform better load balancing.

F. Register File Queue Sizing

Figure 18 shows how varying the RFQ size impacts performance on average across the benchmarks. Having more entries can improve the ability to better overlap compute with memory access, but also increases register file footprint which can reduce the number of concurrently executing thread blocks on the SM. While the queue size can be individually set per kernel, we find on average that 32 entries per channel is the best balance and provides a 15% improvement over 8 entries. After 32 entries, register pressure reduces SM thread block occupancy for many of the benchmarks. The `3d_unet`,

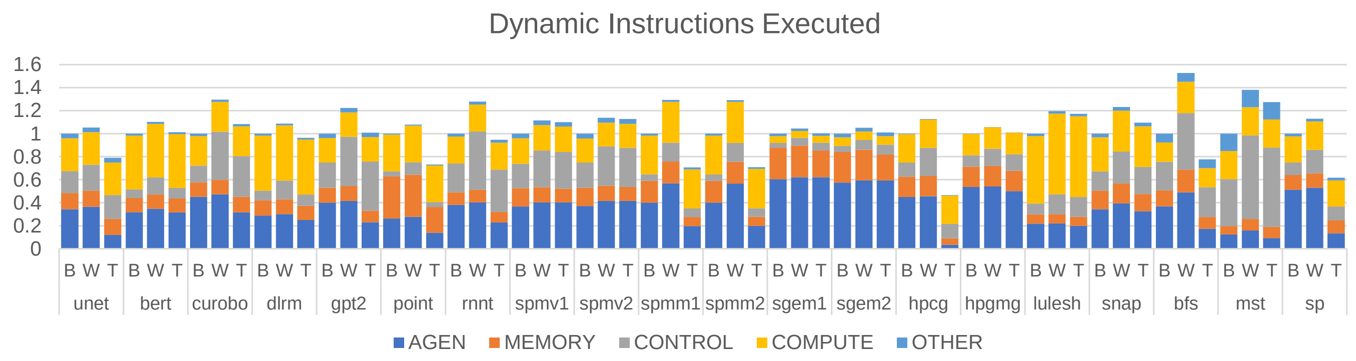


Fig. 19: Dynamic instructions executed for baseline (B), WASP software address generation (W), and WASP-TMA (T).

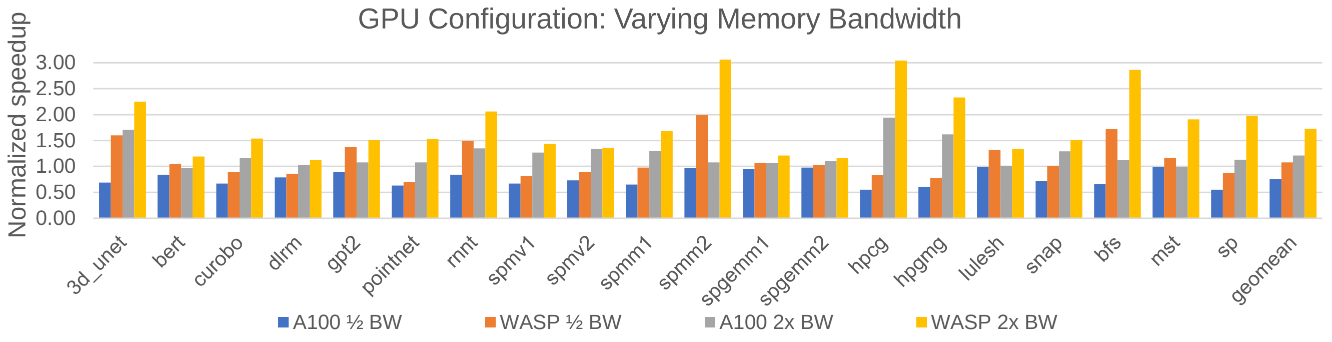


Fig. 20: Evaluation on different GPU configurations, varying memory bandwidth.

`pointnet`, `spmm2` benchmarks see the most benefit when increasing the size of the queue, with more than 20% runtime improvement when the queue size is at least 32 entries per channel instead of 8 entries.

G. Dynamic Instruction Overhead

Figure 19 shows the breakdown in dynamic instructions executed by category. Using our WASP compiler, we annotate instructions by category using PDG analysis and collect the execution statistics using the NVAS performance model. WASP kernels naturally contain overhead instructions for managing the control flow for extra warps forming the pipeline. We find on average that WASP generates 18% extra dynamic instructions, with a worst case of 52%. Generating WASP programs targeting WASP-TMA reduces dynamic instructions executed on the processing blocks by reducing memory, address generation, and control instructions in the memory-access warps. Some CUDA applications like `hpcg`, `pointnet`, `spmm`, `bfs`, and `sp` benefit greatly and reduce their dynamic instructions to as much as 46% of the baseline. These applications prominently feature kernels that are dominated by structured memory accesses inside well-defined loops.

H. L2 Bandwidth Utilization

The goal of warp specialization is to better overlap memory access and compute, which results in better utilization of compute and memory resources. Figure 21 presents the L2 bandwidth utilization of each of the benchmarks on the

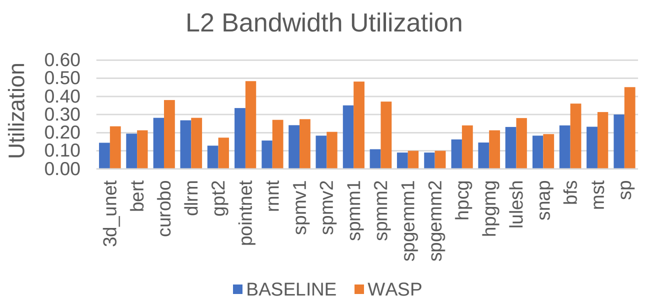


Fig. 21: L2 bandwidth utilization for WASP over the baseline.

BASELINE and **WASP_GPU+WASP_COMPILER_ALL**. We find that WASP generally improves utilization over the baseline. For `curobo`, `pointnet`, and `spgemm`, nearly all the speedup is attributable to better L2 and DRAM utilization. In some benchmarks like `3d_unet` and `spmm` we find that some of the performance benefit comes from better L1 cache hit rates due to improved memory access ordering.

I. Other GPU Configurations

Figure 20 presents a sensitivity study of WASP on other GPU configurations with different bandwidth to compute ratios. We vary the L2 and DRAM bandwidth on baseline A100 and WASP GPUs, by halving and doubling both bandwidths. We label these configurations $\frac{1}{2}$ BW and $2\times$ BW respectively. In the figure, we show performance for each application

TABLE IV: WASP Area Overhead (Storage Requirements)

Item	Per-SM Storage	Per GPU
Warp Mapper	32 CTAs x 132 bits per entry	$\sim$ 56 KB
Warp Scheduler	64 Warps x 7 bits per entry	$\sim$ 48 KB
RFQ Metadata	64 Warps x (4 x 9 bits per entry)	$\sim$ 30 KB
WASP-TMA	2 x 128 bytes per entry	$\sim$ 27 KB
Total		$\sim$ 162 KB

considering all of it's CUDA kernels, with speedup for all the configurations normalized to the A100 baseline.

First, we observe that the A100 baseline is a balanced design - across the applications, halving the bandwidth generally halves performance (geometric mean $0.75\times$). From this perspective, applications fall into two groups: bandwidth sensitive (e.g. `3d_unet`, `pointnet`, `hpcg`) and bandwidth insensitive (e.g. `spgemm1` and `lulesh`). When less bandwidth is available, WASP is able to more efficiently use what is available, reaching the performance of the baseline A100 with just $\frac{1}{2}$ memory bandwidth. Exemplars of this include (`3d_unet`, `bert`, `gpt2`, `rnnt`, `spmm`, and `bfs`), where WASP at half memory bandwidth is able to reach or exceed performance levels of the baseline at double bandwidth. This is because by design, WASP can spread out memory accesses *over time*, reducing the total bandwidth requirements of the GPU architecture.

When bandwidth is doubled with the A100 $2\times$ and WASP $2\times$ configurations, some applications see benefits when they are not computation bound. Examples include `pointnet`, `rnnt`, `bfs`, `mst`, and `sp`. In these applications the baseline is unable to utilize the additional bandwidth, being limited by address generation overheads and serialization. WASP's ability to overlap requests and create a producer-consumer pipeline allows it to better extract the available memory bandwidth.

J. WASP Hardware Complexity

WASP requires additional hardware support in the SM's warp mapper, warp scheduler, register file, and TMA unit (Figure 4). However, WASP requires only minor changes to control logic and not datapaths or data storage, resulting in a low cost amortized across the vector width of the SM. Table 4 presents the overhead for the main cost of the augmentation, which is metadata storage. The total extra storage required on the GPU is less than 162 kilobytes. Overall, we estimate that the total additional hardware area required for WASP is less than 1% of the total GPU chip area.

The warp mapper in the SM requires new storage space to store the augmented thread block specification and extra FSM logic. For each thread block, WASP requires an additional 4 bits to specify number of stages, and 16 bytes to specify the unique number of registers (maximum 256) required for each stage. The FSM requires an additional inner loop that iterates over the number of stages, allocating the specified registers for the stage.

Warp schedulers in the processing blocks require more state storage per warp, and combinatorial logic to combine the new state into a priority. The extra state needed for each warp's scheduler entry are bits for stage id, `is_empty`, and `is_full`.

For RFQs, the main cost is in the queue mapping table and update logic for the processing blocks. WASP supports one RFQ per warp on the SM, and the queue mapping table stores four indices into a 512-entry register file. The update logic requires two sets of 9-bit adders, comparators, and muxes to update the head and tail pointers using the mapping bounds.

WASP-TMA enables the ability to issues loads with SMEM, a particular warp's RFQ, or TMA (gather) as the destination. For gather operations, a two-entry ping buffer for intermediate indices and extra control logic for routing the second phase is needed.

VI. Other Related Work

There is substantial prior work on overlapping memory accesses and compute on GPUs. Warp specialization was first introduced in CudaDMA for coarse-grained memory tile transfers to SMEM [1], and has been used in many applications to overlap computation and memory [2], [19], [48]. Similar work decouples regular affine computations from the rest of GPU kernel [46]. WASP extends state-of-the-art work on warp specialization with new hardware and compiler support.

Prior work also focuses on improving warp and thread block scheduling to better overlap memory access and compute [39] and with improved memory access scheduling for cache locality [16], [36]. One particular work investigated *horizontal fusion* as a way to concurrently execute kernels with opposing memory-intensive and compute-intensive behavior [17]. WASP uses warp specialization to explicitly extract parallelism within a single kernel and overlap memory access and compute.

Prior work on coarse-grained transfers include D2MA and Hopper that employ a hardware accelerator exposed to programmer [11], [23] and vector instructions added to the GPU's ISA for gather/strided patterns [8]. WASP uses WASP-TMA, a new offload accelerator for fine- and coarse-grained data movement that integrates with warp specialized pipelines on GPUs. Prefetching has been explored in the context of GPUs broadly [15], [38], to improve thread block scheduling [13], [29], and for indirect memory gathers [49]. WASP leverages warp specialization to make only demand requests to the memory subsystem and avoids speculative memory accesses.

Modern GPUs also provide support for pipeline parallelism between threads blocks executing on different SMs. Nvidia's Hopper H100 allows direct data communication between SMs via distributed shared memory [23]. VersaPipe enables thread blocks to dynamically act as different pipeline stages using task queues existing in global memory [50]. Unlike WASP, both of these techniques do not explicitly take advantage of the heterogeneity of pipelines stages either in hardware or software, nor provide execution overlap at the SM level. Symphony is a new GPU-like accelerator and programming language design which enables explicit programming of pipeline stages that are connected together [31]. WASP provides compiler support for pipeline parallelism more transparently to the programmer while integrating supporting hardware features on existing GPUs.

Decoupled access-execute processors (DAE) explicitly split memory and compute into two distinct programs that execute concurrently [40]. DAE has been introduced into multicores [10], [47] augmented to handle memory indirection on a multi-threaded manycore processor [9], integrated into an out-of-order core to support graph algorithms [20]. Other related work developed languages and compilers partition a program into sub-contexts for execution on parallel processors that communicate using queues [30], [37], [42], [44]. WASP focuses on new hardware and software mechanisms specifically for GPUs that enables pipeline-aware execution.

VII. Conclusion

In this work, we present WASP, hardware and compiler support for warp specialization, a powerful technique for overlapping memory access and compute operations to accomplish better GPU performance. WASP improves over state-of-the-art GPUs by introducing: new hardware to handle fine-grained memory access patterns, pipeline-aware warp mapping and scheduling, and a compiler that reduces programmer burden. We find that the WASP compiler improves runtime performance over state-of-the-art GPUs by 23%, and by 47% when combined with the new WASP hardware.

References

[1] M. Bauer, H. Cook, and B. Khailany, "Cudadma: Optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011.

[2] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 119–130.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[4] S. M. Collection. (2023) Amd/g3_circuit. [Online]. Available: https://sparse.tamu.edu/AMD/G3_circuit

[5] S. M. Collection. (2023) Snap/roadnet-ca. [Online]. Available: https://sparse.tamu.edu/SNAP/roadNet-CA

[6] S. M. Collection. (2023) Williams/mac_econ_fwd500. [Online]. Available: https://sparse.tamu.edu/Williams/mac_econ_fwd500

[7] S. M. Collection. (2023) Williams/webbase-1m. [Online]. Available: https://sparse.tamu.edu/Williams/webbase-1M

[8] N. C. Crago, M. Stephenson, and S. W. Keckler, "Exposing memory access patterns to improve instruction and memory efficiency in gpus," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, oct 2018.

[9] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, p. 117–128, jun 2011.

[10] T. J. Ham, J. L. Aragón, and M. Martonosi, "Desc: Decoupled supply-compute communication management for heterogeneous architectures," ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 191–203.

[11] D. A. Jamshidi, M. Samadi, and S. Mahlke, "D2ma: Accelerating coarse-grained data transfer for gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 431–442.

[12] A. Kerr, D. Merrill, J. Demouth, and J. Tran. (2017) Cutlass: Fast linear algebra in cuda c++. [Online]. Available: https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[13] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "Cta-aware prefetching and scheduling for gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 137–148.

[14] L. B. N. L. (LBL). (2023) Hpgmg: High performance geometric multigrid. [Online]. Available: https://bitbucket.org/hpgmg/hpgmg/

[15] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 213–224.

[16] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 515–527.

[17] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for gpu kernels," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 14–27.

[18] L. L. N. L. (LLNL). (2023) Livermore unstructured lagrangian explicit shock hydrodynamics. [Online]. Available: https://github.com/LLNL/LULESH/tree/2.0.2-dev/cuda

[19] N. Maruyama and T. Aoki, "Optimizing stencil computations for nvidia kepler gpus," 2014.

[20] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 596–608.

[21] Q. M. Nguyen and D. Sanchez, "Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1262–1274.

[22] Nvidia. (2020) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[23] Nvidia. (2022) Nvidia h100 tensor core gpu architecture. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[24] Nvidia. (2023) Cuda binary utilities. [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[25] Nvidia. (2023) Cuda c++ programming guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[26] Nvidia. (2023) cusparse: Basic linear algebra for sparse matrices on nvidia gpus. [Online]. Available: https://developer.nvidia.com/cusparse

[27] Nvidia. (2023) Nvidia hpc-benchmarks. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks

[28] Nvidia. (2023) Tiny cuda neural networks. [Online]. Available: https://github.com/NVlabs/tiny-cuda-nn

[29] Y. Oh, K. Kim, K. Y. Myung, H. P. Jong, Y. Park, W. R. Won, and M. Annavaram, "Apres: Improving cache efficiency by exploiting load characteristics on gpus," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 191–203.

[30] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005, pp. 12 pp.–118.

[31] M. Pellauer, J. Clemons, V. Balaji, N. Crago, A. Jaleel, D. Lee, M. O'Connor, A. Parashar, S. Treichler, P.-A. Tsai, S. W. Keckler, and J. S. Emer, "Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing," *ACM Trans. Comput. Syst.*, vol. 41, no. 1–4, dec 2023.

[32] E. C. Project. (2023) Exascale proxy applications benchmark suite. [Online]. Available: https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/

[33] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *arXiv preprint arXiv:1706.02413*, 2017.

[34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[35] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada,

B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[36] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.

[37] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 422–433, may 2003.

[38] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 73–82.

[39] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 174–185.

[40] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119.

[41] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. Van Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, N. Ratliff, and D. Fox, "Curobo: Parallelized collision-free robot motion generation," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 8112–8119.

[42] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11*. Springer, 2002, pp. 179–196.

[43] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen, and P. Bird, "Compiling and optimizing for decoupled architectures," in *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, 1995, pp. 40–40.

[44] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. USA: IEEE Computer Society, 2008, p. 3–12.

[45] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 868–880.

[46] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 295–306.

[47] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 736–749.

[48] H. Wei, E. Liu, Y. Zhao, and H. Yu, "Efficient non-fused winograd on gpus," in *Advances in Computer Graphics: 37th Computer Graphics International Conference, CGI 2020, Geneva, Switzerland, October 20–23, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 411–418.

[49] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 178–190.

[50] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: a versatile programming framework for pipelined computing on gpu," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 587–599.