

Better Call SoL: Enhancing GPUs with a Sea-of-LUTs Core

Shashank Nag, Justin Garrigus, Yuting Chen, Lizy K. John

The University of Texas at Austin, USA

Email: shashanknag@utexas.edu, justingarrigus@utexas.edu, ytc@utexas.edu, ljohn@ece.utexas.edu

Abstract—GPUs have emerged as a pervasive compute acceleration platform for a wide range of workloads. Their highly data-parallel architecture enables high-throughput applications, including machine learning, scientific computing, and graphics rendering. In recent years, the prevalence of GEMM-based applications led to the introduction of dedicated Tensor Cores in the GPU architecture, which were optimized for such workloads. However, current GPUs lack support for custom dataflow architectures and operations, and often perform sub-optimally for workloads with primitives not supported on GPU hardware.

In this work, we propose introducing a Sea-of-LUTs (SoL) core to the GPU alongside typical functional units, providing a programmable logic area that can be dynamically reconfigured from within the GPU to implement custom operators or dataflow architectures. By tightly integrating this core on the same GPU die, we offer a fast, reconfigurable architecture while avoiding the external pin and I/O bottlenecks observed in heterogeneous platforms involving FPGAs. We evaluate the SoL core using a combination of GPU performance evaluation tools (Accel-Sim & AccelWattch) and FPGA architecture modeling tools (COFFE & VTR). We assess a diverse set of workloads on the GPU-SoL architecture, offloading custom functions such as Softmax to the SoL core. With only 6% area overhead, the proposed SoL core enables over $1.34\times$ performance improvement and $1.22\times$ energy reduction across the evaluated workloads.

Index Terms—GPU, FPGA, Fine-grained Programmable Logic, Custom Operations, Dataflow Architectures

I. INTRODUCTION

GPUs have become ubiquitous in recent times, with widespread applications ranging from machine-learning (ML) training and inference serving, to graphics processing and data analytics. In addition to offering massive parallelism and accelerated performance, the ease of programmability for generic applications makes them stand out compared to other domain specific or custom accelerators. GPU architectures have seen constant improvements to support the most prominent workloads, with current architectures having significant optimizations for matrix multiplication [9], [49], [50].

While GPUs offer remarkable performance with highly data-parallel and regular operations, their performance on many other types of irregular workloads such as dynamic programming, graph analytics, and sparse data traversal is sub-optimal compared to custom dataflow architectures implemented on FPGAs [16]. GPUs are designed for static and predictable execution pipelines, and do not support custom dataflows involving fine-grained pipelines and dynamic data routing and scheduling. Techniques like coalescing memory accesses [22] alleviate the memory bandwidth bottleneck, at

the expense of further enforcing strict data access behavior – as a consequence, applications like sparse matrix algebra and table lookups which involve scattered memory accesses end up being inefficient. Despite highly quantized and narrow bit-width ML models gaining prominence [44], [48], GPUs end up underutilizing their potential with their limited support for low-precision operations [64]. On the other hand, most of the above limitations can be overcome by using custom hardware devices such as FPGAs or ASICs: FPGAs can implement custom dataflow architectures and fine-grained pipelines supporting specific access patterns and data types, with recent work on low-precision model inferences demonstrating $8\times$ improvement in energy efficiency over GPUs [77].

GPU-FPGA systems. Though GPU-FPGA heterogeneous systems are less common, there are several workloads which would greatly benefit from them [28], [75]. For instance, GPUs are the primary hardware platform used for Large Language Model (LLM) inference today, as they support dominant LLM operations such as general matrix multiplications (GEMMs). However, latency-bound non-linear operations such as Softmax and LayerNorm often form the bottleneck in modern LLM inference on GPUs [36], [54]. This becomes even more critical in long-context architectures, with Softmax contributing up to 30% of the overall inference latency [19], [35]. In contrast, these operations can be efficiently implemented on FPGAs, offering substantial speedup compared to GPU implementations [35], [67] (Fig. 1(a)). Moreover, although GPUs are highly optimized for many of the dominant workloads of today, they may perform sub-optimally on future workloads that incorporate new, custom, or unconventional operations that differ from current computational patterns. Having an FPGA-like system in conjunction with a GPU can support any new operations, with custom dataflow architectures offering optimized implementations [39], [59], [70].

Challenges and Motivation. Heterogeneous integrations involving FPGAs are bottlenecked by the interconnect bandwidth and latency, as they contain limited I/O ports [61]. As shown in Fig. 1(b), workloads often have repeating structures, and offloading intermediate operations to other devices through interconnects would cause significant energy and latency penalties due to back-and-forth data movement, motivating the need for a tighter integration.

Several applications including ML inference, robotics, and data-adaptive algorithms require FPGAs to be dynamically reconfigured for different hardware functionality [42]. However,

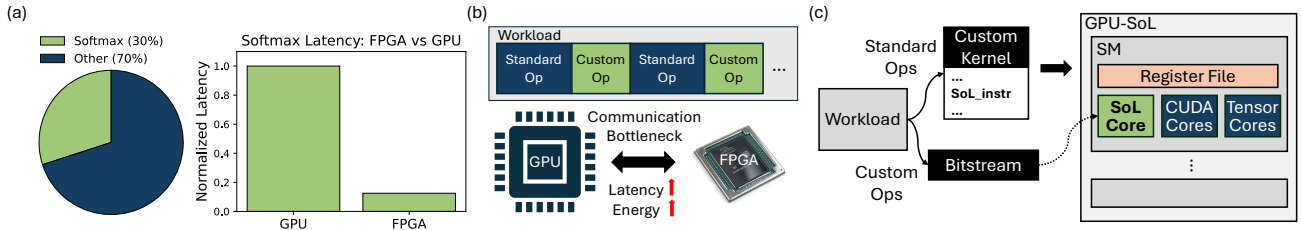


Fig. 1: (a) GPU workloads often have latency-critical segments that can be efficiently accelerated on FPGAs. E.g., Softmax could account for up to 30% latency in LLM inference, and can achieve up to $8\times$ speedup on FPGAs. (b) In current heterogeneous platforms, such workloads with alternating layers mapped to GPUs and FPGAs often suffer from communication bottlenecks, driving latency and energy costs. (c) Our proposal: GPUs enhanced with a Sea-of-LUTs (SoL) core. Workload-specific ops that are better suited for custom dataflow architectures are performed on SoL Cores, while the regular ops are computed as usual.

most current FPGAs have a single JTAG port for configuring the PL that serially loads in the configuration data (bitstream), causing long programming latencies, making them not very suitable for dynamic reprogramming [73]. Therefore, fast dynamic reconfiguration support is critical for a hybrid GPU-FPGA system.

A Case for GPU-PL Integration. Our analysis suggests that a tight integration of GPU and Programmable Logic (PL) resources, with the latter being rapidly and dynamically reconfigurable, would benefit a range of applications as compared to GPU-only or FPGA-only solutions. Data transfer between the GPU and the PL fabric can achieve high bandwidth as it does not cross chip boundaries. Further, such an integrated system can have the GPU operating at its native high frequency, without being constrained by the operating limits of a programmable fabric. We also note that since GPUs already contain memory and floating-point functional units, only logic blocks would be needed in the PL, and BRAMs and DSP slices could be avoided. The ease of programmability for such an integrated solution would be key to its widespread adoption – recent advances in conversion of higher level language (for example, PyTorch) to Verilog and FPGA bitstreams [10], [20], [60] enable this.

Our Proposal. We propose to enhance GPUs with a tightly integrated Sea-of-LUTs (SoL) core to implement fine-grained custom operations and dataflows. The SoL core includes a programmable logic (PL) region with LUTs organized into tiles, or Configurable Logic Blocks (CLBs). The contents of the LUTs and the interconnects between the tiles can be configured using a highly parallel interface. Byte-addressable I/O buffers (registers) act as interfaces between the PL region and the rest of the GPU. Fig. 1 (c) shows a typical workflow on a GPU enhanced with SoLs (GPU-SoL). For operations that would benefit from a custom dataflow-based acceleration, a design and its corresponding configuration data (bitstream) is generated, which is loaded into the SoL core ahead of time. GPU kernels incorporate new SoL instructions that trigger the pre-programmed SoL core to perform its operations, in a much more efficient fashion compared to their native GPU execution. For designs that involve multiple layers of custom operations that cannot simultaneously fit on the SoL core, the architecture enables dynamic reconfiguration of the core on a different stream while standard operations continue to be executed. This

removes the SoL configuration latency from the critical path of execution, ensuring optimal utilization of the GPU resources.

Goal. The primary intent of this work is not to increase the compute-density of GPUs with computational motifs common to all workloads, but rather to provide a customizable fabric to provide special function units on a workload-specific basis. The SoL core complements the GPU, accelerating irregular and special functions within larger workloads, while leveraging the high compute density and parallel memory access strengths inherent to GPUs.

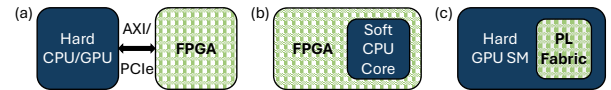


Fig. 2: Prior work: (a) externally interfaced FPGA-CPU/GPU heterogeneous systems, and (b) soft CPU core inside an FPGA. (c) Proposed reconfigurable programmable logic (PL) fabric integrated inside the GPU SM (this work).

Prior Work. The idea of integrating FPGAs and processors has been explored in the past, offering flexibility, high performance, and customizability for specific tasks [4], [5], [62], [72]. These prior heterogeneous designs generally fall into two categories: (i) External FPGA acceleration as in Fig. 2(a), and (ii) Soft processors integrated into FPGAs as in Fig. 2(b). In the former category, “hard” (dedicated silicon) processors are interfaced with an external FPGA through an interconnect, such as AXI or PCIe bus, like the Zynq series of Xilinx FPGA boards having ARM-based embedded CPUs interfaced with the logic fabric [5], [71], [74]. Platforms such as Project Brainwave from Microsoft have also been developed to accelerate ML inference by augmenting CPUs with an FPGA-like Neural Processing Unit (NPU) [14]. This category suffers from high communication overheads and reprogramming latency that prevent fine-grained interleaving of tasks across platforms. In the latter category, “soft” processors are implemented in the FPGA’s logic. FPGAs have had such integrated embedded processors, as seen in AMD/Xilinx MicroBlaze [72] or Intel Nios V [4]. Due to the limitations of the programmable logic fabric, these processors typically operate at low-frequencies (50-250 MHz). In contrast, we propose a fine-grained programmable logic (PL) fabric with a sea of LUTs, integrated within the

GPU, offering a large interface bandwidth, fast reprogramming, and potentially high operating frequency (Fig. 2 (c)).

To the best of our knowledge, ours is the first work that integrates a fine-grained programmable logic fabric inside the GPU fabric. DySER [26], a prior CGRA work has some conceptual similarities to SoL due to its tight integration with the processor core. However, there are several important distinctions: (i) Target domain: DySER is designed for CPU pipelines, whereas SoL is designed for GPU SIMT execution and SM integration, which introduces different architectural constraints and opportunities. (ii) Functionality model: DySER operates as a CGRA-like fabric with a fixed set of functional units and configurable interconnects to accelerate common computational motifs. In contrast, SoL enables fine-grained, programmable-function LUT-based hardware tailored to the specific needs of each workload. (iii) Purpose: DySER primarily increases compute density for common patterns. On the other hand, SoL enables GPUs to efficiently accelerate irregular, control-flow-intensive, or non-linear operations that are currently inefficient on GPUs, complementing its high-throughput compute units. Thus, while DySER shares the general idea of tight integration with a processor core, the architectural goals, programmability model, and targeted workload characteristics differ substantially.

Overall, we make the following contributions in this paper:

- We augment GPUs with an integrated Sea-of-LUTs (SoL) core to offer GPU-SoL, accelerating irregular and custom operations through fine-grained dataflow architectures. This complements the computational strengths of the GPU with a flexible programmable region for custom compute, supporting any unconventional operations in the future.
- We develop a software-hardware workflow supporting GPU-SoL, with dedicated SoL instructions that integrate well with the GPU execution model. The newly introduced SoL programming instruction also enables dynamic reprogramming with minimal overheads.
- We evaluate our proposal using a combination of FPGA architecture modeling (COFFE & VTR) tools and GPU performance evaluation tools (Accel-Sim & AccelWattch). With only 6% area overhead, the SoL core offers $1.34\times$ end-to-end performance improvement and $1.22\times$ energy reduction on a set of ML, HPC, and general workloads, demonstrating the wide applicability of this work.

II. BACKGROUND & MOTIVATION

A. GPU Architecture Evolution : TCs & SFUs

With the popularization of deep learning in the last decade, matrix multiplication on GPUs became mainstream, leading to the incorporation of *Tensor Cores* (TCs) into the GPU architecture [7], [17], [21], [30], [43], [56]. However, many workloads, including elementwise kernels, FIR filters, and several non-ML applications primarily use CUDA cores, and do not benefit from Tensor Cores at all – either because they lack GEMM operations or because their datatypes and operations map better to CUDA cores [27]. Therefore, accelerating non-GEMM operations continues to be valuable even in modern

GPUs with TCs. In this work, our focus is to enable efficient execution of irregular or special non-GEMM functions which are commonly seen across workloads, yet map poorly to current GPU cores and execution paradigms.

Recent GPU architectures also include *Special Functional Units* (SFUs) within each SM, dedicated to perform transcendental math operations such as *sin*, *cos*, *exp*, and *log* using approximate computing algorithms [38]. Operations like Softmax in LLM inference use these SFUs to accelerate *exp* computations [9]. However, architectural improvements such as these are based on current workloads and are implemented for specific operations. Consequently, any introduction of unconventional operations in future workloads would trigger a redesign of the architecture. While these units deliver significant performance gains for the narrow classes of operations they target, their utility is limited by their fixed functionality. Essentially, programs that do not benefit from such operations will have an unavoidable section of hardware idling throughout the execution. As workloads diversify, designing a dedicated hardware unit for each emerging function is impractical. As a result, such units are often only introduced to cater to the most common operations across workloads, while the rest of the operations remain unoptimized and fall back on the general functional units. This motivates the need for a general purpose programmable logic unit to implement workload-specific SFUs.

Programmable Logic Blocks within GPU for future!

Functional units on GPUs are optimized for present workloads. Having a programmable logic block within the GPU will enable any new custom operations in future workloads to be accelerated as well.

B. Limitations of FPGA Architecture

FPGAs involve logic slices and interconnects that are programmed to the design being implemented, with the bitstream specifying the contents of the configuration SRAMs. Reconfiguration is achieved by updating the SRAM contents while “streaming” in the bitstream through dedicated configuration ports on the FPGA. This can be done using a scan-chain-based method, with a single JTAG port for serial loading of the bitstream, or with a frame-based reconfiguration, with prior work also proposing memory-based dynamic reprogramming [45], [69]. However, reconfiguration in commercial FPGAs continues to be a slow process, and limits their usage in scenarios where designs have to be swapped to support different functions or contexts [13], [15], [66]. In addition, FPGAs also consist of dedicated DSP slices, BRAMs (memory), and peripherals connected to the programmable logic, with the former two accounting for over 30% of the area [8]. These additional elements also result in a low logic density as compared to other processors, and end up necessitating a lower operating clock frequency. Finally, I/O ports which facilitate data movement to the FPGA logic fabric are often limited in number, and bottleneck the overall latency [61]. In this work, we retain the high compute density of a GPU but achieve reconfigurability as

in FPGAs while alleviating the limitations of external FPGAs using a programmable logic unit within the GPU.

III. THE SEA-OF-LUTs (SoL) CORE

Based on the insights gathered in the previous sections, we propose GPU-SoL: a GPU enhanced with a programmable logic block called the Sea of LUTs (SoL) core (Fig. 3 (a)). In addition to the typical CUDA/functional units (FUs) and Tensor Cores, a SoL core is included in each Streaming Multiprocessor (SM), to implement custom dataflow architectures for realizing specific functions.

A. Design Choices

We make two key design choices in the SoL core architecture. *First*, we choose to only include logic slices (LUTs) in the programmable logic, and omit DSP slices/ BRAMs typically seen in modern FPGAs, in order to limit the area overhead of the SoL core. The idea behind this is that existing tensor cores and single-precision/double-precision functional units in GPUs are efficient in floating point math – and operations that are heavily dominated by such computations should remain on the SM and utilize these units for better throughput. Further, LUTs themselves are functionally complete, implying they could realize these floating point operations albeit with reduced throughput. Therefore, operations that are better suited for SoL, but involve limited interspersed floating point computations can still be supported on the SoL while achieving an overall performance improvement. Our tight integration enables such fine-grained offloading of operations within a single kernel.

Second, we opt for a distributed design, with small SoL cores within each SM, rather than a larger centralized core outside the SM. This distinguishes our work from other architectures, such as the XDNA architecture in NPU [57], where a large centralized Neural Processing Unit is integrated to the processors, with a shared main memory. Our distributed design choice is driven by the GPU architecture and execution model. GPUs are designed to parallelize execution across SMs, with each of them accessing data and performing computations in parallel independently – thereby enabling high memory bandwidth and computational throughput. By adopting a distributed architecture, we align with this native execution model, and ensure data movement to/ from the SoL continues to exploit the inherent bandwidth advantages of the GPU memory hierarchy. A decentralized SoL core also avoids the overheads and challenges associated with synchronization across thread blocks/ SMs, when trying to offload operations within the kernel boundary. While introducing SoL cores at the SM Sub-Partition (SMSP)-level would have offered a simpler hardware-integration, it hinders user adoption as warp distribution across SMSPs is hardware-managed at runtime, without user transparency – severely constraining the programming model. We taxonomize this design scenario in Section V-H.

B. SoL Core Architecture

Fig. 3 (b) shows the architecture of the proposed SoL core.

Programmable Logic (PL): At the heart of the SoL core is the programmable logic segment, which includes logic tiles comprising lookup tables (LUTs) and flip-flops (FFs) that are organized with configurable interconnects. As shown in Fig. 3, the PL consists of $M \times M$ tiles, each having N **Basic Logic Elements (BLEs)**. The BLE in turn comprises of a K -input LUT and an FF with multiple output drivers. As illustrated, Routing MUXes connect the input wires in the tile to the LUT inputs. Connection-box MUXes are responsible for selecting inputs to the tile from the routing channels, and switch-box MUXes connect tile outputs to the routing channels. The LUT contents and the MUX select lines together form the configurable elements of the design, with dedicated SRAM cells for each of these holding their configuration data. Table I summarizes the parameters of the SoL core architecture. The number of tiles, number of BLEs, size of LUT, input and channel capacity, etc. can be adjusted at design-time according to the needs of the specific GPU-SoL. The logic slices are realized using custom-sized transistors, with SRAM cells holding the programmed configuration data. We use COFFE2 [76], an FPGA architecture modeling tool to model this (elaborated in Section IV).

TABLE I: SoL Core Configuration Parameters

Parameter	Description
$M \times M$	Number of tiles in the core
N	Number of BLEs in each logic tile
K	Number of inputs to each LUT
I	Number of input wires to each tile
W	Number of routing wires (channels)

Bitstream Loader: The bitstream loader programs the SoL core to the specific logic circuit being implemented based on the bitstream, by loading the bits to the appropriate configuration SRAM cells in the tiles. In order to support high-speed dynamic reconfiguration of the PL, we propose multiple scan-chain links that concurrently program different regions of the PL. Each scan-chain link connects the SRAM configuration cells of a tile in a serial single-bit link. During PL programming, the configuration data for each of these cells is streamed in through the scan chain, and each cell is programmed in a serial fashion. Our design includes dedicated scan-chains for each tile, thereby allowing them to be programmed in parallel at a high-speed. Note that in case of typical FPGA boards, the PL interfaces with off-chip peripherals through I/O ports, imposing constraints on the number of pins available to program it using JTAG interfaces. However, in our design, the SoL core being on the same GPU die allows us to create multiple scan-chains through the PL, without requiring external I/O ports.

Input and Output Buffers: The input and output buffers provide the interface between the register file in the SM and the PL in the SoL core. These buffers are essentially staging registers typically seen in functional units that can be addressed or indexed from the GPU threads and PL.

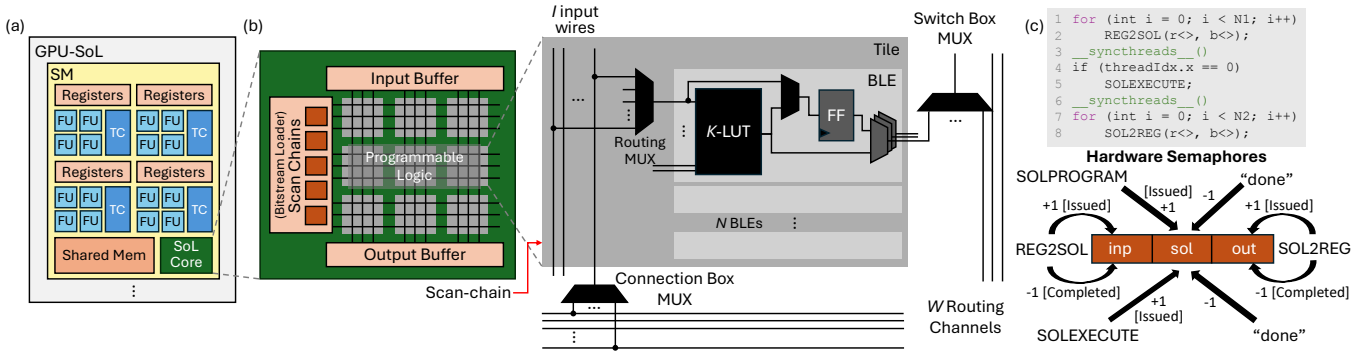


Fig. 3: SoL Core Architecture. (a) SoL core within each GPU SM. (b) The core comprises programmable logic (PL) of logic tiles and interconnects that can be rapidly programmed using a highly parallel interface, and input/output buffers through which data movement occurs to/from the PL. (c) Typical structure of SoL instructions and semaphores for hazard scoreboarding.

C. SoL Instructions

Our design introduces four new hardware SASS instructions to support the SoL core:

- `REG2SOL r_data, r_bufidx`: moves data from a GPU register into SoL's input buffer at the specified index.
- `SOL2REG r_data, r_bufidx`: moves data from the SoL's output buffer at the specified index, into the GPU register.
- `SOLPROGRAM`: reconfigures the SoL core with the bitstream data currently in the input buffer.
- `SOLEXECUTE`: executes the SoL core with the input data currently within the input buffer. After execution, the outputs are available in the output buffer.

Prior to execution on the SoL core, the `SOLPROGRAM` instruction is issued to program the PL with the bitstream or configuration data. The bitstream is loaded to the SoL input buffer through a series of `REG2SOL` instructions before issuing the `SOLPROGRAM` instruction. In case the bitstream exceeds the size of the input buffer, multiple `SOLPROGRAM` calls can be issued serially. Once programmed, a sequence of `REG2SOL`, `SOLEXECUTE`, and `SOL2REG` instructions are used with other SASS code to manipulate the SoL core. Once the data is moved to the SoL buffer with `REG2SOL` instructions, a `SOLEXECUTE` instruction is issued, starting the pre-programmed operations in the SoL core. In most scenarios, programming of SoL happens very infrequently, and is usually followed by several calls of `SOLEXECUTE` for a series of inputs.

D. SoL-SM Synchronization

1) *Clock Frequency Domains*: Owing to the nature of the programmable logic fabric, the SoL core might be constrained to run at a lower clock frequency than the GPU. To support this, the SoL core operates at a divided clock compared to the rest of the SM. The operating frequency of the SoL core is determined at the time of bitstream generation, and the clock dividers to the SoL are configured to that factor. The clock domain crossing of data between the GPU SM and the SoL is inherently supported by the input/ output buffers, with the read/ write ports on either side of the buffers operating at

their respective clock frequencies. For instance, the actual data movement/ writes from the SM to the SoL input buffer occur entirely at the SM clock domain, and such writes to the input buffer can occur on consecutive cycles. In our execution model, the SoL does not access data from the buffers at this time, ensuring stability. Only the control signals (`SOLEXECUTE` in this case) actually cross the clock domain, and signals to the SoL that the data is ready to be read by the SoL at its clock frequency. This clock domain crossing of the control signals is facilitated by flip-flop synchronizers, adding a 2 cycle penalty before the SoL can actually begin processing from the time a `SOLEXECUTE` signal is issued. Our evaluation takes these aspects into account, by specifying the `SOLEXECUTE` latency in terms of equivalent GPU cycles, and by adding an extra 10 GPU cycle latency penalty to the `SOLEXECUTE` instructions.

2) *Hazard Scoreboarding*: Fig. 3(c) describes the hazard scoreboarding mechanisms in GPU-SoL. When a `REG2SOL` instruction is issued, it increments the *inp* hardware semaphore (counter), and it decrements the same when it is completed. The warp scheduler also tags the hardwareID of the block accessing the SoL to ensure that other thread blocks do not attempt to access it meanwhile. The scheduler/ scoreboard for the `SOLEXECUTE` instruction tracks the *inp* (as well as *out*) semaphore(s), and issues it only after all previous `REG2SOL` instructions have completed (*inp* semaphores on all SMSPs should be 0). Issuing `SOLEXECUTE` increments the *sol* semaphore, and it waits for a "done" signal from the SoL – which then decrements the *sol* semaphore. This is tracked before issuing the `SOL2REG` instruction, which similarly updates the *out* semaphore.

3) *Memory Consistency*: The SoL core does not initiate any additional memory requests on its own. The SoL core only accesses data in its dedicated input and output buffer, and this data is moved to/ from the buffer by the SM through the `REG2SOL` and `SOL2REG` instructions. As is the convention in GPU programming models, software synchronization barriers ensure that while the SoL is operating on the data within its buffer, the rest of the SM is not accessing that data. With this, GPU-SoL maintains the same overall memory consistency model as a regular GPU.

4) *Programming Model*: The SoL core follows a programming model similar to a regular GPU, as shown in Fig. 3(c). The SoL buffers are indexed with physical addressing. Importantly, threads concurrently issue their respective REG2SOL and SOL2REG instructions for parallel data movement, but the SOLEXECUTE instruction is only called by a single thread in the thread block. Synchronization instructions are inserted before and after it to ensure that the data movement instructions (REG2SOL / SOL2REG) are issued in the correct order. Additional synchronization instructions between the SoL instructions and other regular instructions are only required if the rest of the program accesses data across warps, as intra-warp dependencies are natively handled by the scoreboard. During program execution, the SoL core and the contents of its buffers are private to a thread block: similar to the conventional GPU programming, the SoL core cannot directly access data from a different thread block. However, techniques like atomics can still be used around the SoL instructions in the same kernel to facilitate such communications.

E. Workflow

Fig. 4 shows the system-level overview and the overall compile-time workflow for program execution on GPU-SoL.

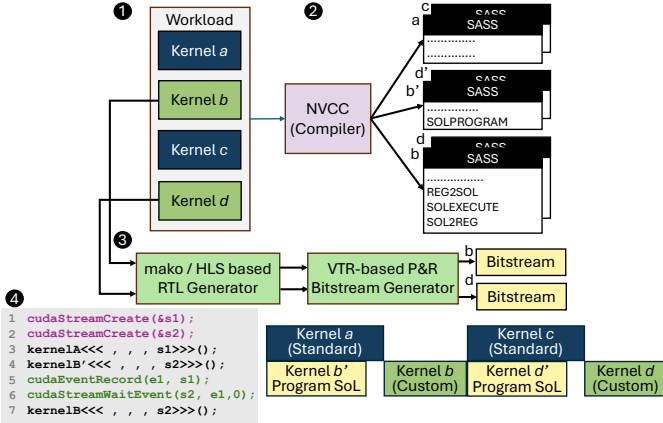


Fig. 4: System Overview. The SoL programming kernels are overlapped with non-SoL kernels through CUDA streams and events, with the default CUDA scheduler.

1 Analyze Workload. The workload is first analyzed to identify operations that could benefit from execution on the SoL cores. Kernels that are comprised of standard operations (such as Kernel *a* and Kernel *c*) are left untouched. Kernels that involve custom operations are modified by inserting SoL instructions. These kernels are marked as those utilizing SoL cores, and are processed differently in subsequent steps. This step can be performed by a compiler that parses through the workload and automatically detects common operations that can be accelerated on SoL cores. Alternatively, these could be manually added by a user through our exposed API.

2 Compilation. The kernels—including those modified in the previous step—go through the compilation flow with the NVCC compiler to create SASS instructions corresponding to each

kernel. The kernels that involve SoL operations include the SoL SASS instructions described in Section III-C. In addition, for each of these kernels involving SoL operations (Kernel *b* or *d*), a separate SoL programming kernel (Kernel *b'* or *d'*) is created consisting of SOLPROGRAM instructions for configuring the SoL core. These programming kernels have a very consistent structure across different kernels, making it straightforward for a compiler to automatically generate them.

3 Bitstream Generation. Kernels involving SoL operations (Kernel *b* and Kernel *d*) pass through two stages: RTL generation and bitstream generation. In our framework, for ML workloads, we employ mako [11], a templating tool to automatically generate RTL for these layers, starting from the trained PyTorch models. For non-ML workloads, we employ Xilinx Vitis HLS to generate RTL from the high-level language code. Alternatively, handwritten templates or other tools like hls4ml [20] or Siemens Catapult [60] could be employed for this. The RTL generator parses through the layers and instantiates SystemVerilog modules implementing the datapath architecture to realize the custom operations. A VTR-based tool then places and routes this RTL targeting the SoL core architecture, with the bitstream to configure the core subsequently being generated. We note that recent advancements in High Level Synthesis (HLS), which automatically converts high-level programs to their bitstream implementations makes this process practically feasible. These tools support programs written in high-level languages such as C/C++, while also having dedicated support for ML models written using frameworks such as PyTorch.

4 Scheduling. SoL kernels must be programmed before they are executed, but this can occur in parallel with non-SoL computations by using two CUDA streams. Some programs [18] feature long latencies between SoL compute kernels and allow the programming latency to be completely hidden, though the small SoL reconfiguration latency means a non-hidden programming kernel is not a significant source of overhead. The dependencies between SoL and regular kernels are handled using CUDA events, as shown in the pseudocode. As the SoL program kernels are issued on the same stream as the SoL execution kernels, the configuration is guaranteed to finish before the SoL execution begins. Further, in most workloads, reprogramming is not very frequent, and therefore any non-hidden programming latencies would get amortized over time.

Finally, the kernels are launched on the proposed GPU-SoL. The regular kernels operate as usual, like any typical GPU, while the SoL kernel instructions specifically target operations on the SoL cores within each SM. Note that while we illustrate a typical workflow with separate kernels for SoL execution and programming, this is not necessary. A programmer could choose to write fused kernels with the SOLPROGRAM and SOLEXECUTE instructions alongside other regular instructions, by inserting appropriate synchronization instructions (as described in Sec. III-D4), and the 'sol' hardware semaphore will ensure that SOLPROGRAM finishes before the next SOLEXECUTE instruction is issued (Fig. 3(c)).

IV. EVALUATION METHODOLOGY

A. Tools

Fig. 5 provides an overview of our design simulation and evaluation methodology. We employ a combination of standard tools, including COFFE2 [76], VTR [58], Accel-Sim [34], and AccelWattch [32] for performance modeling and evaluation of our proposed design. In this section, we provide details of our evaluation framework.

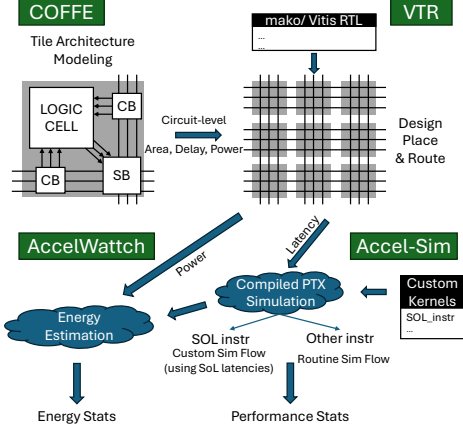


Fig. 5: Overview of the Evaluation Methodology.

1) *COFFE*: As our SoL core architecture closely resembles standard FPGA programmable logic (PL) blocks, we use COFFE2 [76], a standard FPGA architecture modeling tool to model the same. COFFE is a transistor sizing and modeling tool developed for FPGA architecture research, which performs SPICE simulations using a 22nm Predictive Technology Model [40] to optimize area and delay of the FPGA tile. It also models the interfaces, crossbars, switch boxes, connection boxes, and their corresponding SRAM configuration cells, thereby accounting for associated overheads in the tile design. We pass the PL design parameters (Table I) and routing flexibility factors as inputs to COFFE. This generates area reports, and circuit-level delay and power characteristics for the primitives (LUTs, interfaces, switches, etc.), which are used to update the FPGA architecture description file for VTR as explained next.

2) *VTR*: The Verilog-to-Route (VTR) [58] toolflow provides an end-to-end framework for evaluating FPGA architectures, from logic synthesis through placement, and routing. It takes in the RTL and FPGA architecture description file and proceeds with the design synthesis and place-and-route flow, targeting the specified SoL core PL architecture. In our evaluations, we provide the generated RTL corresponding to the custom architectures being implemented in the SoL core for each of the workloads. With this, we obtain latency and power statistics of the synthesized, placed, and routed design from VTR, which is then used in other tools as described next.

3) *Accel-Sim*: We use Accel-Sim [34], a popular cycle-level GPU simulation framework for performance evaluation of GPU-SoL. Accel-Sim allows us to model architectural changes to

GPUs from a performance viewpoint and provides detailed kernel-wise performance reports. We model the PL, input buffer, and output buffer within the SoL core each as distinct functional units in the simulator, each of which uses the same pipeline stages as other standard GPU instructions (i.e. instruction fetch, decode, issue, operand collection, execute, and writeback). REG2SOL and SOL2REG both have execution stages with latencies of one cycle, representing the movement of data from the register file into or out of the input/output buffers in the SoL Core. The execution latencies of SOLPROGRAM and SOLEXECUTE are obtained from the reports generated by VTR for the specific workload and architecture.

For our evaluations, we instrument the baseline program and replace the operations that we wish to perform on the SoL core with kernel calls comprising SoL instructions. We accomplish this by inserting sentinels into the CUDA code representing the new instructions and using a compiler to transform these into their intended SoL instructions after the programs are traced. This transformation preserves the correctness of the original program while allowing typical NVCC optimizations to still be performed, modeling what it would look like if a real version of NVCC is released with access to our SoL instructions and SoL cores in hardware.

4) *AccelWattch*: AccelWattch [32] is a GPU power modeling framework that is tightly integrated with Accel-Sim, enabling a fine-grained cycle-level power estimation. We update the GPU power configuration in AccelWattch to incorporate the power corresponding to the SoL core functional units based on the power reports obtained from VTR, and run simulations for the programs with the new SoL instructions.

5) *RTL Generation*: For ML models, we employ mako [11] template scripts that automatically generate RTL for the custom operations being accelerated on SoL cores. For other workloads, we use Xilinx Vitis HLS [6] to generate RTL from the high-level functions, subject to the resource constraints imposed by the SoL core.

B. Workloads

To demonstrate the efficacy of GPU-SoL, we consider a set of ML and other diverse workloads, that involve operations that could benefit from a custom dataflow accelerator. These workloads are representative of common real-world problems in ML inference, medical imaging, physics simulations, neuroevolution, and genome sequencing. For these workloads, custom operations are accelerated on SoL cores, while traditional operations continue to be implemented using existing cores on the GPU, as described in Fig. 4. Table II provides an overview of these workloads and their operations that are scheduled on the SoL core. This analysis demonstrates that there are certain kernels or program phases that are better suited for custom logic, while certain others may be better suited for regular GPU SMs. These findings are also corroborated in prior work [16].

In long-context LLM inference [29], [51], [65], [68], non-linear operations like Softmax often bottleneck the overall latency, and performing these operations on SoL could reduce the overall latency. Vision transformers also show similar

trends, with some variants like LL-ViTs [48] having additional low-precision lookup (LUT) layers which perform poorly on native GPUs, while also necessitating reprogramming between layers. While benchmarks like kmeans and particlefilter from the Rodinia [12], and tsp (genetic algorithm) from the HecBench suite [31] have dense math computations suited for GPUs, the reduction and cumulation operations in these suffer from synchronization bottlenecks – these could be performed on the SoL much more efficiently. The keyword spotting models, which are often run on edge GPUs in robotics and IoT applications, also involve several bitwise operations and irregular data accesses [10] that are ill-suited for the native GPU architecture. In the needle benchmark, the dynamic programming component involves a wavefront dependency, which can be more efficiently addressed in a custom dataflow architecture in the SoL core, while the GPU facilitates scheduling and parallel data movement.

TABLE II: Overview of Workloads

Workload	SM-friendly parts	SoL-friendly parts
LLMs: llama2 [65], mistral [29], BERT [18]	Linear layers (GEMM, GEMV)	Softmax (custom pipeline)
LL-ViT [48]	Linear layers (GEMM, GEMV)	Low-precision LUT layers, Softmax
Clustering (kmeans/ Rodinia [12])	Distance computation (dense math)	Center reduction (synchronizations)
particlefilter (Rodinia [12])	Likelihood evaluation (dense math)	Cumulation (syncs, random data accesses)
Genetic Algorithm (tsp/HecBench) [31]	Distance computation (dense math)	Random initialization, reduction
Keyword Spotting (kws/DWN [10])	Scheduling, parallel data movement	Bitwise operations, irregular data accesses
Genome Sequence Alignment (needle, Rodinia [12])	Scheduling, parallel data movement	Dynamic programming (wavefront dependency)

C. Experimental Setup

SoL Core Configuration. For our evaluations, we assume the following default layout for the programmable logic: we consider a 44×44 grid of tiles, each of which has $N = 10$ BLEs with $K = 6$ -input LUTs. This contains $I = 40$ input wires to each tile, and $W = 320$ routing channels. We consider the SoL input and output buffers to each have a size of 1KB for our experiments, and each SM to have one SoL core. We analyze other configurations in Section V-G.

Accel-Sim Configuration. To evaluate the performance of our workloads on Accel-Sim, we use two variants of performance configurations – an embedded grade version along the lines of a Jetson Orin [52] and a server grade version along the lines of an A100. The configuration details are summarized in Table III. In our evaluation, the experiments with llama2 and mistral LLMs use the server grade configuration, while the others including Rodinia/HecBench benchmarks use the embedded grade version. We also modify AccelWattch with power values from VTR runs and integrate it with Accel-Sim to give detailed power metrics.

TABLE III: Accel-Sim Configurations

	Embedded Grade	Server Grade
Architecture	Ampere (sm87)	Ampere (sm80)
# SMs	16	108
# Registers / SM	65536	65536
L1 Data Cache + Shared Memory	192KB	192KB
# Exec Units / SM	4 SPs, 4 DP, 4 SFUs, 4 INTs, 4 TCs, 1 SoL core	4 SPs, 4 DP, 4 SFUs, 4 INTs, 4 TCs, 1 SoL core
L2 Cache	4MB	20MB
GPU Core Clock	1300 MHz	1410 MHz
SoL Core Clock	260 MHz	282 MHz
Memory Bandwidth	83.5 GB/s	1.95 TB/s

Implementation Details. In our evaluations involving machine learning models, we use baseline PyTorch [53] based GPU implementations. For the SoL instructions, we define custom CUDA kernels consisting of these instructions and integrate them with the PyTorch model implementations.

Figures of Merit. We report the speedup in end-to-end inference latency, and improvements in energy efficiency (energy per inference) as the key figures of merit. In order to evaluate the benefits of SoL cores in accelerating latency-critical operations and analyze its end-to-end performance implications, for ML workloads we primarily consider single-batched model inference scenarios.

V. RESULTS

A. Performance Improvements and Breakdown

Fig. 6 shows the end-to-end performance improvement achieved with GPU-SoL on a range of workloads, along with a breakdown of the segments or kernels offloaded to the SoL core. Overall, the proposed GPU-SoL achieves $1.34\times$ geomean end-to-end speedup across workloads compared to the baseline GPU by accelerating specific custom operations on the SoL cores. As seen in Fig. 6, these offloaded operations range between 15-90% of the baseline execution latency, indicating the diverse mix of our benchmarks.

We observe that language models (llama2, mistral, BERT) achieve around $1.25\times$ speedup owing to Softmax acceleration on SoL, with softmax achieving a speedup of $3\times$ on an average through a custom dataflow implementation [37]. The LL-ViT models achieve additional speedups owing to the low-precision lookup operations in them being implemented in the SoL core, which get accelerated by roughly $6\times$. The KWS workload sees the highest speedup, as this model is dominated by several bitwise operations, all of which are accelerated on the SoL cores. In the kmeans and tsp benchmarks, the reduction operations get accelerated by about $3\times$ to achieve an overall speedup of $1.2\times$ and $1.15\times$ respectively. In the particlefilter benchmark, the cumulation operations achieve a $7\times$ speedup to offer an overall $1.2\times$ speedup, and the needle benchmark achieves a $1.33\times$ speedup by accelerating the wavefront-based dynamic programming algorithm. We note that even if we omit the highly optimized KWS workload, GPU-SoL offers a $1.25\times$ geomean speedup on the other diverse set of benchmarks.

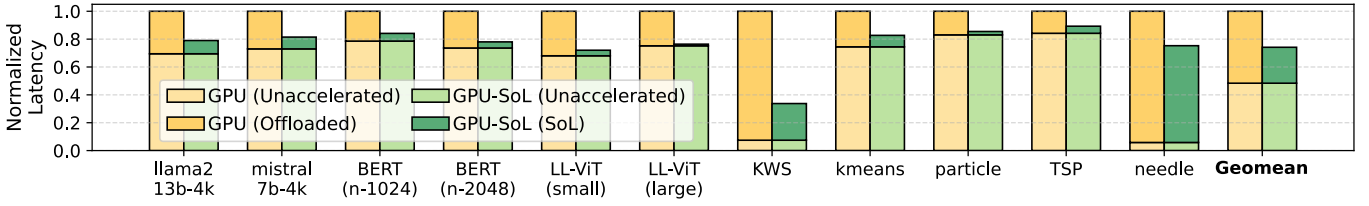


Fig. 6: End-to-end performance improvements offered by GPU-SoL against a regular GPU on a range of workloads.

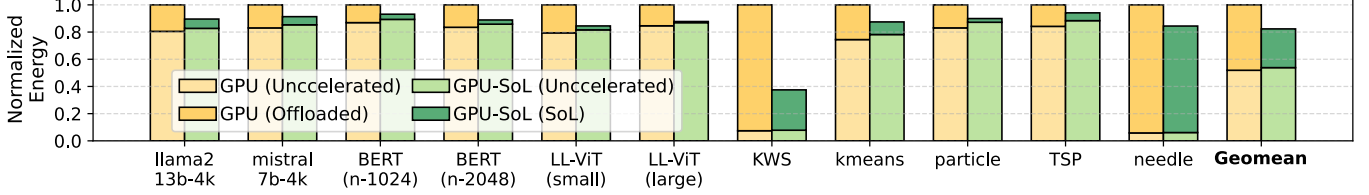


Fig. 7: End-to-end energy efficiency offered by GPU-SoL against a regular GPU on a range of workloads.

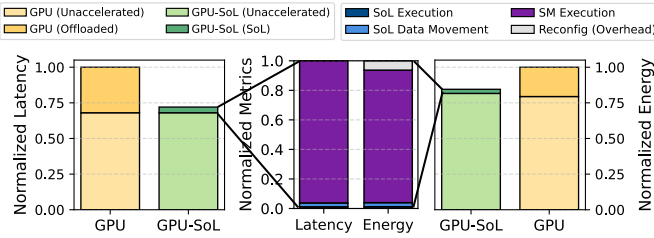


Fig. 8: Latency-energy breakdowns for LL-ViT workload.

B. Energy Efficiency

Fig. 7 demonstrates the energy efficiency offered by GPU-SoL on end-to-end inferences for the same set of workloads. Overall, SoL cores enable a $1.22\times$ geomean energy reduction as compared to baseline GPUs without SoLs. The lower execution time on GPU-SoL translates to energy reduction, though the SoL cores add approximately 6% area and some communication and reprogramming overheads which offset some of the benefits. We also observe that the idle/ leakage power from the SoL core is a relatively small fraction of the total GPU power – as a consequence, the energy increase of the unaccelerated portion is minimal. Although the energy reduction is less than the latency reduction, the $1.22\times$ energy reduction positions GPU-SoL as an energy-efficient accelerator. We note that the fraction of latency and energy spent within a SoL kernel on SoL operations and regular SM operations (loads, stores, etc.) could vary across workloads. For instance, Fig. 8 shows the breakdown of latency and energy spent in the kernel accelerated for the LL-ViT workload. In this scenario, the majority of latency and energy costs of the SoL kernel can be attributed to the regular SM operations (loads/ stores) within it.

C. Area Overhead and Frequency

Table IV shows the area breakdown of individual components of a tile within the SoL core for a few different configurations in the SoL design space. As anticipated, scaling the number of input wires and routing channels considerably increases the area associated with the switch block. For all of our evaluation,

we chose the $(N=10/K=6/I=40/W=320)$ configuration, primarily because it resembles the characteristics of a commercial Stratix III FPGA, thereby validating the feasibility of implementing and routing designs on our SoL core. We note that the choice of LUT-6 ($K=6$) and 10 BLEs per tile ($N=10$) is a standard practice in commercial FPGAs. For our chosen tile configurations, we observe the area overhead of the SoL core to be about 1.8 mm^2 per SM on 22nm technology (from the per tile area of $933.19 \mu\text{m}^2$ indicated in Table IV), including all the area contributions from the I/O buffers, reconfiguration circuitry, crossbars, among others. To understand the relative area penalty of adding SoL cores to a current GPU, we adjust this area to the 7nm technology node using scaling factors established by a prior work on FPGA tile area scaling [3]. Considering a pessimistic (15nm to 7nm) scale-down factor of $3.7\times$, we observe this area overhead to be 0.48 mm^2 per SM on the 7nm technology node, with prior work noting a 2% standard deviation in this scaling methodology across architectures [3]. Compared to the areas reported for an A100 GPU (826 mm^2 , 108 SMs) [49], this overhead is roughly 6% of the normalized per SM die area – a relatively small overhead. We also note that these scaling factors are also consistent with a widely established prior work quantifying technology scaling relations for CMOS devices [63].

Table IV also shows the critical path delay within a single tile, ranging about 150-200 ps – this includes the delays of the LUTs, switch, MUX and interconnects within a single tile. Typically, critical paths in the implemented designs involve about 5-20 such logic stages [1]. This impacts the operable frequency of the programmable logic fabric, as reported in Table IV (adjusted for technology scaling [63]). We note that these frequencies are higher than the operating frequencies seen in commercial FPGAs due to (i) the simplicity of our PL fabric and (ii) a modern technology node. However, just for the purpose of a more rigorous evaluation, we consider a pessimistic operating frequency of **260-282MHz** for the SoL core (aligning with commercial FPGAs). We also note that all of the implemented designs meet these timing requirements.

TABLE IV: SoL Core Area & Frequency for each tile. N BLEs; K input to each LUT; I input wires; W routing channels.

	N:10/K:6 I:40/W:160	N:10/K:6 I:40/W:320	N:10/K:6 I:80/W:320
Tile Area (μm^2)	721.18	933.19	1230.78
Breakdown			
LUT	271.77	278.20	286.88
FF	8.02	8.02	8.02
BLE Output	25.26	24.06	25.27
Local MUX	128.81	128.86	197.11
Connection Block	114.39	169.69	344.12
Switch Block	172.91	324.33	369.36
Tile Critical Path	159.42ps	169.44ps	201.48ps
Tile Max Freq.	6.27 GHz	5.90 GHz	4.96 GHz
Fabric Freq. (7nm)	1.1-4.1GHz	0.9-3.8GHz	0.8-3.3GHz
Evaluated Freq.	–	260/282MHz*	–

*This configuration was used in our evaluations

D. Reconfiguration Overheads

As mentioned in Section IV-B, for workloads like LL-ViT, their model inference on SoL cores involves intermediate tile reconfiguration. The highly parallel configuration interface in our design loads the reconfiguration data (bitstream) to the SRAM cells at a high speed, taking less than 10,000 GPU cycles, with a power overhead of $\sim 3\text{W}$. This latency is effectively hidden with the execution of model layers that do not utilize the SoL cores, thereby incurring no penalty on end-to-end performance. This pattern can also be seen in Fig. 9.

E. GPU Architectural Analysis

We present a range of analysis with the LL-ViT workload to demonstrate the efficacy of our approach, and to position our work in the context of current GPU microarchitectures.

1) *SoL Utilization and Active-Warp Stats*: For the kernels involving operations accelerated on SoL, the peak SoL utilization reaches 100% across workloads, while the average utilization varies considerably depending on the nature of operations offloaded. For instance, while compute-intensive operations like softmax see an average utilization upto 80%, the low-latency LUT kernels in LL-ViT show an average utilization of less than 10% (in this case, data loads/ stores to the cache by the SM account for almost 90% of the idle time of the SoL). We note that a low utilization of SoL core does not indicate poor performance, as the SoL could potentially finish its operations very efficiently, at which point the data loads from cache/ memory to the registers become the bottleneck.

As noted in Section III-D4, only one thread contains the SOLEXECUTE instruction, and a good implementation should not have multiple CTAs with SoL instructions mapped to the same SM. Therefore, in all our implementations, none of the active warps stall due to SoL unit contention. For instance, the kernel accelerated with SoL in the LL-ViT workload, exhibits nearly 70% SM occupancy, and scoreboard stalls account for over 92% of its active warp stalls (primarily data dependencies due to pending memory fetches), with barriers and structural hazards contributing to less than 3% of warp stalls. This analysis indicates that our programming model ensures the SoL

core is occupied as intended, and regardless of the low average SoL utilization, GPU-SoLs offer a $1.34\times$ overall speedup.

2) *Resource Utilization and Timeline*: Fig. 9 compares the SM utilization of GPU-SoL against that of a default GPU, of a single pair of layers in the LL-ViT workload which involves SoL reprogramming. As expected, the SM utilization for the parts that are not accelerated on SoL cores remains unchanged. Furthermore, the programming latency is well below the latency on the critical path, and for the highlighted region that is accelerated on SoL cores, the latency is considerably reduced.

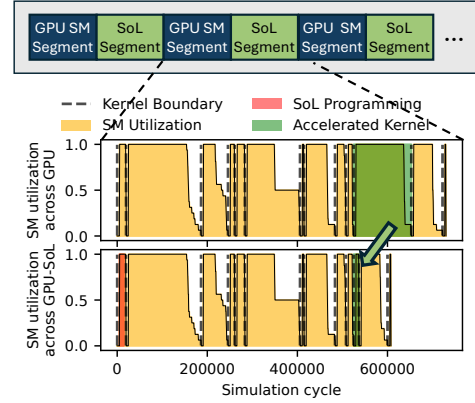


Fig. 9: SM utilization and programming latency hiding.

In most of the workloads considered here, there are repeating patterns of direct data dependencies between SoL-friendly and the SM-friendly operations, and these cannot be logically overlapped. For instance, reduction and cumulation operations (kmeans, particle, etc.) can only begin once all the previous operations have finished, and the next iteration of these workloads can resume only after these reduction operations have finished. In addition, many of these operations primarily rely on CUDA cores, further limiting any potential for latency hiding. Therefore, these operations are on the critical path, and the SoL core speeding up these operations reduces the end-to-end latency, without introducing any additional synchronization penalties. In case of the LLM workloads, while softmax operations could potentially be overlapped with some GEMM operations on the tensor cores, the data dependencies between the attention and MLP layers limit the extent of the overlap. As exponents are long-latency and tensor cores offer high-throughput, softmax operations could be exposed on the critical path and become the bottleneck, as highlighted in prior work [33], [55]. Therefore, accelerating softmax operations on the SoL core is valuable.

3) *Impact on memory accesses*: Accelerating operations on SoL cores reduces memory accesses over the course of these operations. With a custom datapath being implemented on the SoL core, data is streamed through wired interconnects. A byproduct of this is many of the memory accesses becoming coalesced. Further, in many cases we observe constant operands in the program being embedded within the SoL core, minimizing the need to fetch these constants from memory, especially with model parameters in ML inferences. Fig. 10

shows the reduction offered on the workload where lookup operations are offloaded to SoL cores. SoL cores offer up to 77% reduction in dynamic GPU loads.

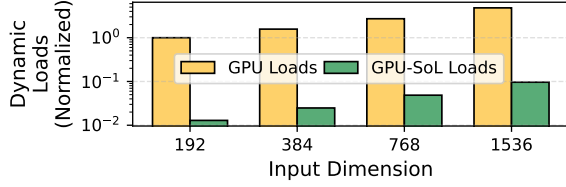


Fig. 10: Normalized number of load instructions in a feed-forward block (from the LL-ViT workload) accelerated on SoL. Significant reduction is observed with GPU-SoL.

4) *Comparison against fast high-capacity cache:* We note that LUTs in the SoL core are not merely offering a closely integrated fast memory block. Instead, these offer programmable logic tiles that implement a datapath realizing custom operations. In order to demonstrate this, we consider a scenario of enhancing current GPUs with an L1 cache that has a hypothetically low latency of 5 cycles, as opposed to the baseline L1 latency of 38 cycles. As seen in Fig. 11 (a), even with such a cache, the achieved latency of custom operations in the workload is 12 $\times$ worse than a GPU-SoL that has the conventional L1 cache. Increasing the size of the L1 cache for such a GPU improves latencies up until the working set is fully resident in L1 cache; though even at that point, its performance is 10 $\times$ worse than a GPU-SoL using the default memory hierarchy.

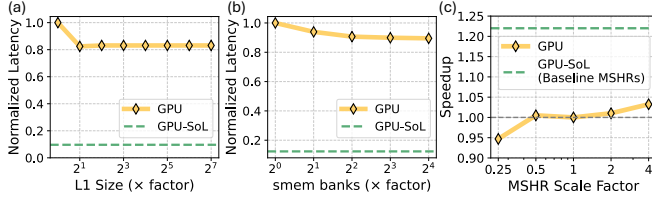


Fig. 11: (a) Latency of operations accelerated on SoL cores against increasing size of a hypothetically fast L1 cache. (b) Latency of operations with increasing shared memory banks. (c) End-to-end speedup impacts of varying the number of MSHRs for the regular GPU.

5) *Comparison against increased shared memory bandwidth:* Fig. 11 (b) shows the latency of custom operations in the workload that are accelerated on the SoL cores as we increase the number of shared memory (smem) banks, thereby reducing bank conflicts and improving shared memory bandwidth. The performance improvement offered by increasing smem banks in a regular GPU still does not match up to that offered by GPU-SoL with the baseline smem.

6) *Comparison against increased MSHRs:* Increasing the number of MSHRs would often improve performance due to better memory access coalescing. Fig. 11 (c) shows the end-to-end speedup of the workload. As seen, despite the fact that more MSHRs offer a speedup for the execution on GPUs,

the performance improvement still does not match up to that offered by GPU-SoL with the baseline MSHRs.

F. Sensitivity: SOLEXECUTE & REG2SOL latencies

Fig. 12 (a) shows the increase in overall latency for three workloads accelerated on the SoL cores with increasing SOLEXECUTE latency. As seen, initially the latency is not impacted as data movement to the SoL dominates latency. Eventually, the latency increases consistent with expectations, but even so, it remains significantly lower compared to the execution latency of the same workload on a regular GPU – as indicated in Fig. 12(a), the latency when SoL performs worse than the baseline is significantly high. Therefore, we ascertain that even if the execution latency of the operations in SoL core is much higher – potentially due to multi-staged RTL or longer critical path delays – GPU-SoL will continue to offer advantages compared to baseline GPUs.

Similarly, Fig. 12 (b) shows the sensitivity of the overall execution latency to the REG2SOL. Even at high latencies of over 50 cycles, SoL offers a 5 $\times$ better performance for the LL-ViT workload. This suggests that even if our latency estimates were overly optimistic, SoL would offer performance advantages under any practical scenario.

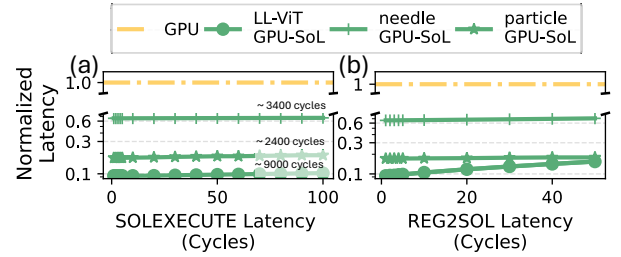


Fig. 12: Sensitivity to SOLEXECUTE and REG2SOL latencies, along with latencies when SoL gets worse than the baseline.

G. SoL Core Area-Latency Tradeoffs

Fig. 13 shows the area-latency tradeoffs of GPU-SoL on the needle benchmark, by varying the number of tiles in the SoL core. The default version (44 $\times$ 44 tiles) has about 19,360 LUTs, and supports a systolic array of BLOCK-SIZE=16 for the dynamic programming operations. As we scale up and down the number of tiles, the BLOCK-SIZE of the design that can be supported changes, thereby impacting the execution latency. With a much smaller 16 $\times$ 16 version (2560 LUTs), the SoL can only support a small design (BLOCK-SIZE=8), performing worse than the GPU baseline. This is to be expected, since 2560 LUTs provide a relatively small amount of logic resources. Therefore, we note that it is important to consider reasonably-sized SoL cores to leverage the true potential of dataflow acceleration.

H. SoL Cores within SMSPs

As mentioned in Section III-A, introducing SoL cores at the SMSP level (instead of SM) would introduce software programming challenges as warps get assigned to SMSPs non-deterministically. A potential way to program such a system

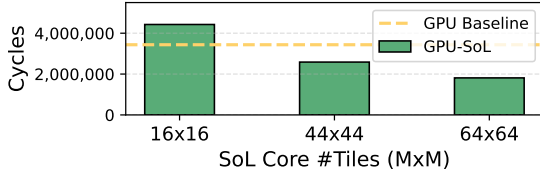


Fig. 13: SoL core area-latency tradeoffs (needle workload).

would be to enforce single-warped thread-blocks. Analyzing this with the needle benchmark suggests that imposing such a restriction causes a $1.2\times$ slowdown in spite of the additional SoL units available in each SMSP. To efficiently realize this, the hardware-schedulers will have to be software-transparent – this exploration is left for future work.

I. Roofline Analysis

We perform roofline analysis with respect to the L2 cache (LLC) on an edge-grade GPU, for a set of representative kernels that are accelerated on the SoL core, and compare them with their performance on the baseline GPU, as shown in Fig. 14. In order to make meaningful comparisons against the baseline, we define “operations” in case of SoL execution to be the equivalent number of operations performed on the GPU under the baseline execution, for the offloaded portion of the workload.

Compute Bound Workloads. We consider one of the most computationally intensive kernel launches from the needle (Rodinia) workload, which lies in the compute bound region of the CUDA Core (CC) roofline (this workload does not use Tensor Cores). As seen in Fig. 14, the workload moves upwards – accelerating the dynamic programming algorithm on the SoL core offers an increase in the performance due to efficient wavefront execution with a custom datapath architecture.

Memory Bound Workloads. The memory bound LUT-layer kernel from the LL-ViT workload sees significant improvement in the achieved performance – while also improving the arithmetic intensity by effectively reducing the memory accesses from the LLC. The improvement can be attributed to two factors: (i) the dataflow execution model on the SoL core ensures intermediate data is staged in internal buffers and streamed; such an execution model also reduces randomized memory accesses, therefore eliminating repeated data movement from the LLC (ii) the serial instructions for each LUT access in the original kernel get replaced by a dataflow execution within the SoL core, reducing front-end pressure and instruction fetch/ decode overheads. Therefore, the SoL core is not merely offering compute acceleration, but also realizing a customized dataflow execution based logic fabric that improves the overall performance of the workload. Further, we also observe that the kernel performing cumulation operations from the particlefilter benchmark, which lies in an intermediate region, also sees an improvement in performance and arithmetic intensity. Therefore, we demonstrate the efficacy of SoL cores in improving performance across a diverse range of workloads, regardless of where they lie under the roofline.

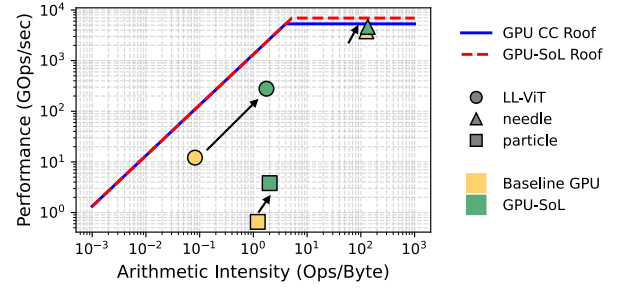


Fig. 14: Roofline Analysis for kernels accelerated on SoL.

J. External FPGA systems

Fig. 15 compares GPU-SoL against externally interfaced GPU-FPGA systems on the BERT (small) workload, with the softmax operation accelerated on the FPGA – involving back and forth data movement. For the GPU-external FPGA system, we consider a small FPGA (Zynq Ultrascale+) and a large FPGA (Virtex Ultrascale+), with highly efficient PCIe-3 and PCIe-4 interconnects, respectively. In spite of such efficient interconnects, we observe that the data movement latencies dominate the execution time on GPU-FPGA systems. In contrast, GPU-SoL avoids the communication bottleneck, offering improved performance. We also note that this gap widens in case of workloads requiring dynamic reprogramming on smaller external FPGAs.

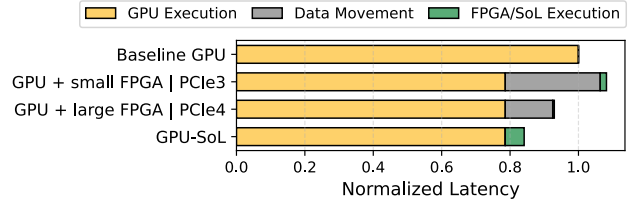


Fig. 15: GPU-SoL vs. external FPGA on a BERT workload.

VI. DISCUSSION AND FUTURE WORK

Applicability to highly regular workloads. As the SoL core is an additional functional unit, highly regular workloads that do not utilize it would not experience a slowdown, although there is a 6% area penalty. Alternatively, for a constant area, GPU-SoLs would experience $\sim 6\%$ degradation for such workloads. However, in such scenarios the SoL core can be potentially configured to function like typical GPU functional units to increase the compute density.

Future Compiler Optimizations. In our current design framework, users and library developers specify the regions of the program to be accelerated on SoL cores through our exposed APIs. In the future, compilers could be enhanced to parse through any workload, identify operations that could benefit from execution on the SoL core, and automatically rewrite kernels to utilize them (Fig. 4). We note that though the SoL configuration bitstream generation adds an overhead to the compilation time of the order of a few minutes [47], this gets amortized over long program executions.

Supporting larger designs on the SoL core. While the size of the PL on the SoL core inherently limits the unrolled sizes of designs implemented on it, larger datapaths can be supported by one of the two mechanisms: (a) the design could be folded in time, and the processing elements could be reused for the next set of operations; or (b) for designs where the datapaths are not identical, the SoL can be rapidly reprogrammed for the next part of the datapath, and the intermediate data can be streamed in. As the `SOLPROGRAM` instruction can be issued from within the same kernel, the intermediate data remains in the register-file or L1 cache, minimizing data movement overheads. By virtue of its fast dynamic reconfiguration, the SoL core can support design sizes exceeding those supported on commercial FPGAs, alleviating conventional bottlenecks.

Fabric Granularity. In this work, we propose adding a fine-grained PL fabric to the GPU, in order to accelerate a wide-range of custom, irregular, or control-flow intensive operations. While a CGRA-like fabric would have offered an increased compute density at a faster clock, the coarse-granularity of its reconfigurability limits its utility for several applications. CGRAs have fixed-function processing elements that are “baked-in” at manufacturing, and consequently can only be designed to cater to the most common operations seen in workloads – with the configurability limited to routing and opcodes. For instance, a CGRA-like core introduced to cater to softmax operations would primarily be designed with floating-point functional units. However, such a CGRA would be largely unutilized for applications such as encryption or LUT lookups, involving low-precision or bitwise operations. These applications could otherwise benefit from a fine-grained configurable fabric.

Modern Technology Push. While GPU and FPGA architectures have independently existed for a while, their tight fine-grained integration has not been explored presumably due to challenges with realizing such integrations. However, modern processors contain up to 18 metal layers, making it feasible to realize the highly parallel interconnects needed to interface with and reprogram the logic blocks. Emerging trends in 3D integrated circuits also make such rapid PL reconfiguration more viable [13], making now an appropriate time to explore such architectures. Such vertical stacking technologies also make larger SoL cores (beyond the 6% area considered in this paper) more practical.

VII. RELATED WORK

In the recent past, many works have proposed architectural enhancements to GPUs leveraging workload-specific performance characteristics. LUT-Tensor Core [46], a recent work, introduced dedicated Tensor Cores for lookup operations, targeting inference of mixed-precision LLMs. However, it can only support look-up based mpGEMMs, and cannot support the other wide range of applications such as reduction and dynamic programming that can be supported by the SoL. Avant-Garde [25] proposes changes to GPUs to support scaled numeric formats. While support for such numerics can be natively realized within the SoL core, such approaches could be

orthogonally considered to leverage the full throughput offered by Tensor Cores. However, approaches such as these target a single operation, and lack generalizability to any workload.

Other works have explored developing new functional units to accelerate some applications [2], [38], [41], though our work shows that reprogrammable devices can improve efficiency while being more generally applicable. Microarchitectural changes to GPUs are expensive to realize in practice due to long drawn out design cycles and lead times. Therefore, practically useful changes must be applicable to a range of workloads and applications, and should ideally stand the “test of time”. In contrast to these prior approaches, our proposed microarchitectural changes are general purpose, and can be repurposed to cater to the demands of any workload.

Going beyond GPUs, PICACHU [55] introduces a coarse external plug-in CGRA to handle non-linear operations in LLMs. However, it has the same communication limitations as earlier discussed in case of externally interfaced FPGAs, and cannot be efficiently used to offload operations at a fine-grained level. Tandem [24] similarly introduces an NPU co-processor for non-GEMM workloads, and Gemini [23] offers a CPU co-processor for DNNs. Both of these approaches involve external, centralized accelerators which cannot leverage custom operation acceleration at the granularity of functional units. The proposed SoL architecture differs fundamentally from all of these approaches. It enables per-workload specialization inside the SM, allowing the system to accelerate irregular or control-flow-dominated portions of GPU workloads.

VIII. CONCLUSION

Though GPUs excel at high-throughput data-parallel applications, their performance on operations that are latency-critical or involving scattered memory accesses is sub-optimal. FPGAs perform well on such operations, with custom datapath architectures implementing these much more efficiently. In this work, we propose to introduce an FPGA-like Sea-of-LUTs (SoL) core within the Streaming Multiprocessor of the GPU in an effort to offer a programmable logic block alongside the existing GPU cores. By integrating SoL cores on the same die, we alleviate the pinout and communication limitations of existing FPGAs and offer a programmable logic region that can be reconfigured with minimal latencies. This enables support for any new custom operations in future workloads, without having to redesign GPU architectures for each emerging workload. Our evaluations using FPGA architecture modeling tools (COFFE and VTR), and GPU performance evaluation tools (Accel-Sim and AccelWattch) on a range of workloads show more than $1.34\times$ performance improvement and $1.22\times$ energy reduction with about 6% area overhead.

ACKNOWLEDGMENTS

This work was supported in part by compute resources through the Center for Generative AI (CGAI) and Texas Advanced Computing Center (TACC) at UT Austin. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the funding agencies.

REFERENCES

- [1] Advanced Micro Devices, Inc., *UltraFast Design Methodology Guide for the Vivado Design Suite (UG949)*, Advanced Micro Devices, Inc., section: Improving Logic Levels. [Online]. Available: <https://docs.amd.com/r/en-US/ug949-vivado-design-methodology/Improving-Logic-Levels>
- [2] A. Agarwal, P. Harsha, S. Vasishta, and S. Sivathantham, "Implementation of special function unit for vertex shader processor using hybrid number system," *Journal of Computer Networks and Communications*, vol. 2014, 10 2014.
- [3] M. Al-Qawasmī and A. Ye, "Scaling the area of synthesizable fpga tiles across semiconductor process nodes," *Microelectronics*, vol. 1, no. 2, 2025. [Online]. Available: <https://www.mdpi.com/3042-5344/1/2/6>
- [4] Altera, "Nios ii embedded processor background." [Online]. Available: <https://www.intel.com/content/www/us/en/support/programmable/support-resources/fpga-documentation-index.html>
- [5] AMD Xilinx, *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*, AMD, Nov. 2022, version 1.10, November 7, 2022. [Online]. Available: https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf
- [6] AMD/Xilinx, *Vitis High-Level Synthesis User Guide (UG1399)*, AMD, 2022, version 2022.2. [Online]. Available: <https://docs.amd.com/r/2022.2-English/Vitis-Tutorials-Getting-Started/Vitis-HLS>
- [7] J. Anderson, M. Garland, J. Nickolls, J. Hardwick, S. Morton, Y. Zhang, E. Phillips, V. Volkov, and S. Le Grand, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 04, pp. 13–27, Jul. 2008. [Online]. Available: <https://doi.ieeeecomputersociety.org/10.1109/MM.2008.57>
- [8] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, "Comefa: Compute-in-memory blocks for fpgas," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9.
- [9] K. Aubrey and N. Stam. (2025, Aug.) Inside nvidia blackwell ultra: The chip powering the ai factory era. NVIDIA Developer Blog. [Online]. Available: <https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/>
- [10] A. T. L. Bacellar, Z. Susskind, M. B. Jr, E. John, L. K. John, P. M. V. Lima, and F. M. França, "Differentiable weightless neural networks," *2024 41st International Conference on Machine Learning (ICML)*, 2024. [Online]. Available: <https://icml.cc/virtual/2024/poster/34511>
- [11] M. Bayer, "Mako templates for python," 2021. [Online]. Available: <https://www.makotemplates.org/>
- [12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.
- [13] N. K. Cherukuri, S. Nag, P. S. Nalla, A. K. Kola, C. S. Gadireddi, K. Dai, J.-s. Seo, Z. Fang, J. Zhang, and Y. Cao, "vfpga: Towards sub- μ s reconfiguration via 3d fpga and packaging co-design," in *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 189–200. [Online]. Available: <https://doi.org/10.1145/3748173.3779206>
- [14] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Hussein, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnn in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.
- [15] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, p. 171–210, Jun. 2002. [Online]. Available: <https://doi.org/10.1145/508352.508353>
- [16] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 93–96.
- [17] A. M. Devices, "Amd cdna architecture," Advanced Micro Devices, Tech. Rep., 2020.
- [18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: <https://aclanthology.org/N19-1423/>
- [19] A. Eassa and S. B. Eryilmaz, "Nvidia blackwell architecture sweeps mlperf training v5.1 benchmarks." [Online]. Available: <https://developer.nvidia.com/blog/nvidia-blackwell-architecture-sweeps-mlperf-training-v5-1-benchmarks/>
- [20] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgreni-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," in *Proceedings of TinyML Research Symposium (TinyML Research Symposium '21)*. ACM, 2021, pp. 1–10. [Online]. Available: <https://par.nsf.gov/servlets/purl/10300178>
- [21] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004. [Online]. Available: <https://api.semanticscholar.org/CorpusID:370998>
- [22] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on gpus," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 12–22.
- [23] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.
- [24] S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. H. Ahn, D. K. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. S. Kim, and H. Esmailzadeh, "Tandem processor: Grappling with emerging operators in neural networks," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1165–1182. [Online]. Available: <https://doi.org/10.1145/3620665.3640365>
- [25] M. Gil, D. Ha, S. B. Harna, M. K. Yoon, B. Falsafi, W. W. Ro, and Y. Oh, "Avant-garde: Empowering gpus with scaled numeric formats," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 153–165. [Online]. Available: <https://doi.org/10.1145/3695053.3731100>
- [26] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.
- [27] B. Hanindhito and L. K. John, "Accelerating ml workloads using gpu tensor cores: The good, the bad, and the ugly," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 178–189. [Online]. Available: <https://doi.org/10.1145/3629526.3653835>
- [28] M. Hosseinabady, M. A. B. Zainol, and J. Nunez-Yanez, "Heterogeneous fpga+gpu embedded systems: Challenges and opportunities," 2019. [Online]. Available: <https://arxiv.org/abs/1901.06331>
- [29] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: <https://arxiv.org/abs/2310.06825>
- [30] H. Jiang, "Intel's ponte vecchio gpu : Architecture, systems and software," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–29.
- [31] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the sycl programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327.
- [32] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "Accelwatch: A power modeling framework for modern gpus," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 738–753.

- [33] H. Kang, S. Bharadwaj, J. Hensman, T. Krishna, V. Rühle, and S. Rajmohan, "Turboattention: Efficient attention approximation for high throughputs llm," in *Proceedings of Machine Learning and Systems*, M. Zaharia, G. Joshi, and Y. Lin, Eds., vol. 7. MLSys, 2025. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2025/file/f4f55846501f3336f293fd8b6de10770-Paper-Conference.pdf
- [34] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.
- [35] R. Kim, D. Lee, J. Kim, J. Park, and S. M. Lee, "Hardware accelerator for approximation-based softmax and layer normalization in transformers," *Electronics*, vol. 14, p. 2337, 06 2025.
- [36] S. Kim, C. Hooper, T. Wattanawong, M. Kang, R. Yan, H. Genc, G. Dinh, Q. Huang, K. Keutzer, M. W. Mahoney, Y. S. Shao, and A. Gholami, "Full stack optimization of transformer inference: a survey," 2023. [Online]. Available: <https://arxiv.org/abs/2302.14017>
- [37] N. A. Koca, A. T. Do, and C.-H. Chang, "Hardware-efficient softmax approximation for self-attention networks," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.
- [38] A. Li, S. L. Song, M. Wijtvet, A. Kumar, and H. Corporaal, "Sfu-driven transparent approximation acceleration on gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2925426.2926255>
- [39] J. Li, G. Shen, D. Zhao, Q. Zhang, and Y. Zeng, "Firefly: A high-throughput hardware accelerator for spiking neural networks with efficient dsp and memory optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 8, p. 1178–1191, Aug. 2023. [Online]. Available: <http://dx.doi.org/10.1109/TVLSI.2023.3279349>
- [40] X. Li, W. Zhao, Y. Cao, Z. Zhu, J. Song, D. Bang, C.-C. Wang, S. H. Kang, J. Wang, M. Nowak, and N. Yu, "Pathfinding for 22nm cmos designs using predictive technology models," in *2009 IEEE Custom Integrated Circuits Conference*, 2009, pp. 227–230.
- [41] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.
- [42] S. Liu, Z. Wan, and B. Yu, *Robotic Computing on FPGAs*, ser. Synthesis Lectures on Computer Architecture Series. Morgan & Claypool Publishers, 2021. [Online]. Available: <https://books.google.com/books?id=sGOLzgEACAAJ>
- [43] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "Gpgpu: general-purpose computation on graphics hardware," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 208–es. [Online]. Available: <https://doi.org/10.1145/1188455.1188672>
- [44] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," 2024.
- [45] U. Malik and O. Driesel, "A configuration memory architecture for fast run-time reconfiguration of fpgas," in *International Conference on Field Programmable Logic and Applications*, 2005., 2005, pp. 636–639.
- [46] Z. Mo, L. Wang, J. Wei, Z. Zeng, S. Cao, L. Ma, N. Jing, T. Cao, J. Xue, F. Yang, and M. Yang, "Lut tensor core: A software-hardware co-design for lut-based low-bit llm inference," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 514–528. [Online]. Available: <https://doi.org/10.1145/3695053.3731057>
- [47] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, Jun. 2020. [Online]. Available: <https://doi.org/10.1145/3388617>
- [48] S. Nag, A. T. L. Bacellar, Z. Susskind, A. Jha, L. Liberty, A. Sivakumar, E. B. John, K. Kailas, P. M. V. Lima, N. J. Yadwadkar, F. M. G. Franca, and L. K. John, "Ll-vit: Edge deployable vision transformers with look up table neurons," 2025, 24th International Conference on Field Programmable Technology 2025 (FPT 2025). [Online]. Available: <https://arxiv.org/abs/2511.00812>
- [49] NVIDIA Corporation, "Nvidia ampere gpu architecture," NVIDIA, Tech. Rep., 2020. [Online]. Available: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf>
- [50] —, "NVIDIA H100 Tensor Core GPU Architecture," NVIDIA, Whitepaper, 2022. [Online]. Available: <https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c>
- [51] OpenAI, "gpt-oss-120b and gpt-oss-20b model card," 2025. [Online]. Available: <https://arxiv.org/abs/2508.10925>
- [52] J. Pan and T. G. Rogers, "Crisp: Concurrent rendering and compute simulation platform for gpus," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2024, pp. 1–14.
- [53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: <https://arxiv.org/abs/1912.01703>
- [54] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of machine learning and systems*, vol. 5, pp. 606–624, 2023.
- [55] J. Qin, T. Xia, C. Tan, J. Zhang, and S. Q. Zhang, "Picachu: Plug-in cgra handling upcoming nonlinear operations in llms," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 845–861. [Online]. Available: <https://doi.org/10.1145/3676641.3716013>
- [56] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 79–92.
- [57] A. Rico, S. Pareek, J. Cabezas, D. Clarke, B. Ozgul, F. Barat, Y. Fu, S. Münz, D. Stuart, P. Schlangen, P. Duarte, S. Date, I. Paul, J. Weng, S. Santan, V. Kathail, A. Sirasao, and J. Noguera, "Amd xdna npu in ryzen ai processors," *IEEE Micro*, vol. 44, no. 6, p. 73–82, Nov. 2024. [Online]. Available: <https://doi.org/10.1109/MM.2024.3423692>
- [58] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeters, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The vtr project: architecture and cad for fpgas from verilog to routing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 77–86. [Online]. Available: <https://doi.org/10.1145/2145694.2145708>
- [59] H. Sharma, J. Park, B. Samynathan, B. Robatmili, S. Mirkhani, and H. Esmailzadeh, "From tensors to fpgas: Accelerating deep learning," in *Hot Chips Symposium*, 2018, hotChips 2018. [Online]. Available: <https://cseweb.ucsd.edu/~hadi/doc/paper/2018-hotchips-dnnweaver2.pdf>
- [60] Siemens EDA, "Siemens catapult high-level synthesis (hls) and verification," <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/>, 2022, accessed: 2025-11-17.
- [61] G. Singh, M. Alser, D. S. Cali, D. Diamantopoulos, J. Gomez-Luna, H. Corporaal, and O. Mutlu, "Fpga-based near-memory acceleration of modern data-intensive applications," *IEEE Micro*, vol. 41, no. 4, p. 39–48, Jul. 2021. [Online]. Available: <http://dx.doi.org/10.1109/MM.2021.3088396>
- [62] B. Sputh, O. Faust, and A. Allen, "Portable csp based design for embedded multi-core systems," in *Communicating Process Architectures 2006*, ser. Concurrent Systems Engineering, P. Welch, J. Kerridge, and F. Barnes, Eds., vol. 64. IOS Press, 2006, pp. 123–134, 29th WoTUG Technical Meeting ; Conference date: 17-09-2006 Through 20-09-2006.
- [63] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74–81, 2017. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167926017300755>
- [64] A. Tirumala and R. Wong, "Nvidia blackwell platform: Advancing generative ai and accelerated computing," in *Proceedings of the 36th IEEE Hot Chips Symposium (HC2024)*, Aug. 2024, paper No. 64, NVIDIA Corporation. [Online]. Available: https://hc2024.hotchips.org/assets/program/conference/day1/64_HC2024.NVIDIA.TirumalaWong.pdf
- [65] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra,

- I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023. [Online]. Available: <https://arxiv.org/abs/2307.09288>
- [66] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018. [Online]. Available: <https://doi.org/10.1145/3193827>
- [67] W. Wang, S. Zhou, W. Sun, P. Sun, and Y. Liu, "Sole: Hardware-software co-design of softmax and layernorm for efficient transformer inference," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.
- [68] X. Wang, M. Salmani, P. Omid, X. Ren, M. Rezagholizadeh, and A. Eshaghi, "Beyond the limits: A survey of techniques to extend the context length in large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2402.02244>
- [69] Y. Wang, Y. Wang, and J. Lai, "An fpga configuration circuit used for fast and partial configuration," in *2007 7th International Conference on ASIC*, 2007, pp. 157–160.
- [70] Z. Wang, Z. Que, W. Luk, and H. Fan, "Customizable fpga-based accelerator for binarized graph neural networks," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 1968–1972.
- [71] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed, "A heterogeneous platform with gpu and fpga for power efficient high performance computing," in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 220–223.
- [72] Xilinx, "Microblaze risc 32-bit soft processor datasheet," 2002. [Online]. Available: https://www.ee.torontomu.ca/~courses/ee8205/Data-Sheets/sopc/MicroBlaze_DataSheet.pdf
- [73] *7 Series FPGAs Configuration User Guide*, Xilinx Inc., San Jose, CA, USA, 2015, describes JTAG-based serial FPGA configuration and related programming latency limitations. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf
- [74] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, G. Huang, and X. Liu, "The case for fpga-based edge computing," *IEEE Transactions on Mobile Computing*, vol. 21, no. 7, pp. 2610–2619, 2022.
- [75] F. Yang, X. Yang, H. Wang, Z. Wang, Z. Zhu, S. Zeng, and Y. Wang, "Glitches: Gpu-fpga llm inference through a collaborative heterogeneous system," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 2024, pp. 1–7.
- [76] S. Yazdandshenas and V. Betz, "Coffe 2: Automatic modelling and optimization of complex and heterogeneous fpga architectures," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3301298>
- [77] C. Yin, Z. Bai, P. Venkatram, S. Aggarwal, Z. Li, and T. Mitra, "Tereffci: Highly efficient ternary llm inference on fpga," 2025. [Online]. Available: <https://arxiv.org/abs/2502.16473>