DeepFlow: A Cross-Stack Pathfinding Framework for Distributed AI Systems

NEWSHA ARDALANI, Meta, Inc., USA
SAPTADEEP PAL and PUNEET GUPTA, UCLA, USA

Over the past decade, machine learning model complexity has grown at an extraordinary rate, as has the scale of the systems training such large models. However, there is an alarmingly low hardware utilization (5–20%) in large scale AI systems. The low system utilization is a cumulative effect of minor losses across different layers of the stack, exacerbated by the disconnect between engineers designing different layers spanning across different industries. To address this challenge, in this work we designed a cross-stack performance modelling and design space exploration framework. First, we introduce CrossFlow, a novel framework that enables cross-layer analysis all the way from the technology layer to the algorithmic layer. Next, we introduce DeepFlow (built on top of CrossFlow using machine learning techniques) to automate the design space exploration and co-optimization across different layers of the stack. We have validated CrossFlow's accuracy with distributed training on real commercial hardware and showcase several DeepFlow case studies demonstrating pitfalls of not optimizing across the technology-hardware-software stack for what is likely the most important workload driving large development investments in all aspects of computing stack.

CCS Concepts: • **Hardware → Application specific processors;**

Additional Key Words and Phrases: Distributed AI systems, hardware-software co-optimization, design space exploration, performance modelling

ACM Reference format:
Newsha Ardalani, Saptadeep Pal, and Puneet Gupta. 2024. DeepFlow: A Cross-Stack Pathfinding Framework for Distributed AI Systems. *ACM Trans. Des. Autom. Electron. Syst.* 29, 2, Article 30 (February 2024), 20 pages.
https://doi.org/10.1145/3635867

1 INTRODUCTION

Over the last decade, the demand on compute and memory resources for AI workloads has grown by multiple orders of magnitude [1]. As AI models grow in size along with the volume of training data, distributed training on cutting-edge scale-out systems composed of a large number of accelerators and processors has become the norm. However, it has often been noticed that large scale AI training suffers from poor resource utilization. E.g., recent analysis reveals 5–20% utilization across 1000s of GPUs [2]. Such poor utilization of resources is becoming a source of major

Newsha Ardalani is with Meta, Inc. This work was primarily done during her tenure at Baidu Research.
Authors' addresses: N. Ardalani, Meta, Inc., 1 Hacker Wy, Menlo Park, CA 94025, USA; e-mail: new@fb.com; S. Pal and P. Gupta, Department of Electrical and Computer Engineering, UCLA, 420 Westwood Plaza, Los Angeles, CA, 90095, USA; e-mails: {saptadeep, puneetg}@ucla.edu.

1084-4309/2024/02-ART30 $15.00
https://doi.org/10.1145/3635867

concern. Inefficiencies across different layers of the compute stack [3, 4] (from hardware micro-architecture to software parallelization strategies) and the design imbalance across different layers are among a few factors that result in such low system utilization. Different layers of the stack, technology nodes, hardware architecture, network topology, model architecture, and parallelism strategy are designed across different organizations and retrofitted into large-scale systems. The distributed nature of the design makes cross-layer optimization challenging if not impossible. For example, high-level algorithmic choices like batch size, model architecture, and parallelism strategies utilize the underlying hardware components (network, memory bandwidth or compute units) in different ways. As a result, depending on these choices, different chip-level and system-level architectural design decisions (e.g., network topologies, memory technology and the choice of technology node) need to be made to ensure high system utilization.

Despite this, the distributed AI training hardware landscape often focuses on just a small set of parallelism strategies for a fixed hardware design [3]. Exploring the trade-offs between parallelization strategy (e.g., data parallelism and model parallelism) and performance (run-time) is often done in an ad-hoc manner. There is no methodical framework or research that explores the trade-offs between low-level hardware technology details and high-level algorithmic design (such as model architecture, parallelism strategy and batch size) on over performance and utilization of compute and memory resources. As a result, we set out to develop a framework that could enable across-the-stack analysis and allow us to look at the optimal points in the vast technology, system and algorithm design space. Towards that goal, we develop **CrossFlow**, a performance modeling framework that enables "what-if" analysis across different layers of the stack, and **DeepFlow** that builds on top of *CrossFlow* and uses machine-learning based techniques to automate the design space search. CrossFlow is an end-to-end performance modeling tool based on an analytical model which takes the entire system-architecture into account and is more sophisticated than a simple Roofline analysis and less time-consuming than simulation. The framework provides a templatized interface for defining technology (minimum operating voltage, bitcell area, etc), chip (compute cores, memory hierarchy, etc.), system-level architecture (node-level organization, intra-node network, and inter-node network), machine-learning model's compute graph, and parallelization strategies and predicts run-time per iteration step. Key contributions of this work include:

- We develop the first open-source, full-stack pathfinding framework, DeepFlow,[1] for large distributed **deep learning (DL)** training: the driving workload for most future technology, hardware and software development (Sections 3–7).
- We validate CrossFlow performance prediction against measurements on real commercial hardware (NVIDIA P4, V100 and DGX-1) running kernels and DL application in both single and distributed settings, observing near perfect correlation and 10%–16% error. Next we show that large multi-chip integration and waferscale technologies would not be worthy investments for large scale language models (Section 8).
- We conduct a variety of case studies looking at the impact of a variety of high-cost technology innovations on the eventual performance of distributed DL training. We show that future logic technology nodes alone would provide minimal performance gains, and advancement in HBM and inter-node network technologies is needed to provide the next leap in performance. Also, optimal parallelism strategy selection could provide more performance gains than using naive parallelism strategies on next generation hardware (Section 9).

CrossFlow and DeepFlow can be used to bridge researchers across different layers of the stack (often spanning across different industries) to communicate their needs.

[1] https://github.com/nanocad-lab/DeepFlow

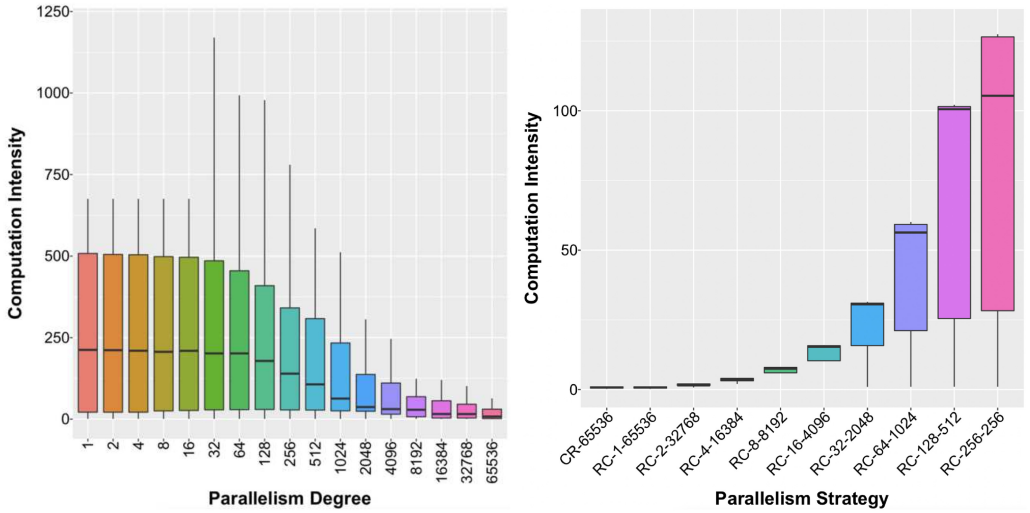

Fig. 1. Impact of parallelism on computation intensity.

2 MOTIVATION

High-level algorithmic design decisions such as batch size, parallelism strategy and degrees of parallelism stress the underlying hardware components in different ways. One important metric that guides a balanced system design is computation intensity. Computation intensity is a workload property defined as the ratio of the number of computation flops to the number of accesses to main memory.

Figure 1 (left) shows the computation intensity distribution across a different number of GPUs. We performed this analysis for a **GEMM (general matrix multiplication)** problem of size $(64K, 64K, 64K)$ distributed across many GPUs. Depending on the parallelism strategy and number of available GPUs, each GPU gets a non-regular matrix shard for compute. Each boxplot shows the spread of computation intensity for a different number of GPUs. For each level of parallelism, we see a large spread of compute intensities, particularly for lower parallelism degrees. This is the result of different parallelization strategies as well as different tiling strategies. It is clear from this figure that computation intensity is much smaller at higher degrees of parallelism, implying the need for a different system design.

There are a myriad of ways to parallelize a model across a large multi-node system. Figure 1 (right) shows the distribution of computation intensity across different parallelization strategies for a fixed level of parallelism (64K GPUs). On the X-axis, we show various parallelization strategies across 64K GPUs. **RC** or **CR** refers to **Row-Column** or **Column-Row** distributed GEMM (a.k.a. kernel parallelism, more details in Section 3.3). As shown, optimal design point is different for different parallelization strategies. Since designing new accelerator architectures (particularly in advanced technology nodes) and developing new integration technologies such as interposers, 3D integration, and the like often costs billions of dollars, it's important to perform thorough design space exploration which encompasses software, hardware design and technology selection. This is essential to guide development of newer technologies and system architectures.

Moreover, large training workloads are rapidly becoming the applications driving large investments in semiconductor technology development all the way down to fabrication equipment, making such a cross-layer pathfinding framework immensely valuable to ML engineers, system architects and technology developers alike. In this work, we developed a cross-layer pathfinding

Table 1. Features of DeepFlow Compared to Other Related Tools

	TimeLoop[5]	Maestro [6]	Mind Mapping [7]	FlexFlow [8]	DayDream [9]	Habitat [10]	Astra-Sim [11]	Astra-Sim2.0 [12]	DeepFlow
Analytical Performance Modelling	Y	Y	Y	N	N	Y	Y	Y	Y
End-to-End DL Network Performance Modelling Support	N	N	N	Y	Y	Y	Y	Y	Y
Scale-out System Modelling	N	N	N	Y	Y	N	Y	Y	Y
Advanced Network and Communication Collective Modelling	N	N	N	N	N	N	N	Y	N
Technology (CMOS nodes, Memory, Interconnect) Modelling	N	N	N	N	N	N	N	N	Y
Automated uArchitecture Generator	N	N	N	N	N	N	N	N	Y
Model-to-Silicon Performance Validation	Y	N	N	Y	Y	Y	N	N	Y
Algorithm-Hardware Co-optimization	N	N	Y	N	N	N	Y	Y	Y
Algorithm-Hardware-Technology Co-optimization and Design Space Exploration	N	N	N	N	N	N	N	N	Y

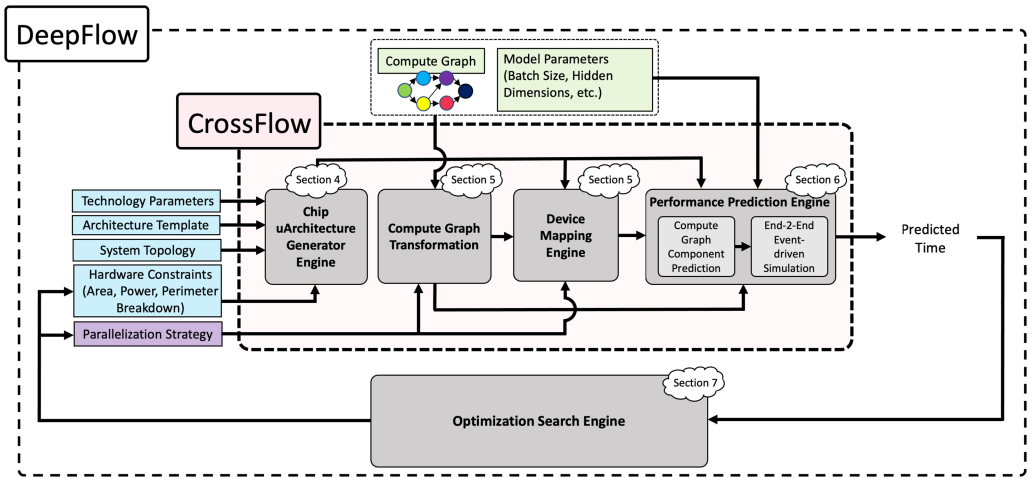

Fig. 2. DeepFlow overview.

framework, which unlike traditional performance simulator and hardware-software co-design frameworks, allows us to evaluate the impact of the choice of technology, micro-architectural design and software parallelization strategies on deep learning workloads. Table 1 shows the features of DeepFlow and compares it to other state-of-the-art hardware-software co-design and performance simulation tools. As shown, DeepFlow allows us to perform cross-stack pathfinding and design space exploration. More details of how Deepflow compares against other related work is provided in Section 10.

3 DEEPFLOW OVERVIEW

Figure 2 shows an overview of the DeepFlow framework. DeepFlow takes the following set of **inputs**: (1) System design hierarchy (e.g., the number of accelerator nodes per device, the number of devices in the system, the network topology connecting nodes within a device and across the devices), (2) Architecture template of each accelerator node which provides a high-level definition of its components and how those components fit together. The purpose of the template is to provide a blueprint for the accelerator without committing to any specific hardware parameters. (3) Technology parameters for each hardware component (e.g., energy per flop), (4) Design budgets for each hardware component (area, power, perimeter), (5) Machine learning model specification in the form of a high-level compute graph, parameters of each compute node (kernel type, tensor dimensions), and (6) Parallelism strategy (data, model, kernel, and/or pipeline parallelism dimensions) which distributes the compute graph across the entire system. (7) Device mapping strategy

which defines mapping of parallel shards onto hardware nodes. Given these inputs, DeepFlow predicts the end-to-end performance of one iteration (i.e., single batch) of the model and finds an optimal hardware-software-technology design point as **output**.

DeepFlow is composed of two major components. CrossFlow which operates in a stand-alone mode and can predict performance for any input configuration; and a **search and optimization engine (SOE)** which enables design space search.

3.1 CrossFlow Building Blocks

Micro-**Architecture Generator Engine (AGE)**. AGE takes the following set of **inputs**: (1) Design constraints (i.e., the power, area and perimeter budget and breakdown across micro-architectural components such as cache, network, compute cores). This breakdown can be provided manually by users or automatically by the Search and Optimization Engine (SOE, Section 3.2). (2) Technology parameters such as energy per flop, energy per data bit transfer for each level of memory and network hierarchy, threshold and maximum gate voltage, integration substrate parameters such as bump/interconnect pitch. We provide a wide range of standard and future technology libraries as baseline. (3) Architecture template which is a blueprint of the underlying accelerator chip without committing to any specific hardware parameters. Given these inputs, AGE performs a frequency-voltage-area scaling optimization to generate the following **output** parameters such that design budgets for all components are met: (1) Compute throughput. (2) Capacity for different levels of memory hierarchy. (3) Bandwidth to each level of memory hierarchy. (4) Inter-node as well as intra-node network bandwidth. These parameters are then utilized by the **performance prediction engine (PPE)** to estimate the execution time of each kernel.

Compute Graph Transformation and **Device Placement Engine (DPE)**. The parallelization strategy and device mapping are critical in deciding the overall execution time. Here, we first transform the model graph to a 'super-graph' to reflect the parallelization strategy provided by the users manually, or SOE engine (Section 3.2) automatically. For example, to apply data parallelism, the model graph is replicated and appropriate edges are added to model the gradient exchange. After generating the transformed graph, DPE assigns the vertices of the transformed graph to the system nodes following a heuristic approach to minimize the communication overhead.

Performance Prediction Engine (PPE). We use hierarchical roofline modeling to predict the performance of each compute node. To calculate the overall end-to-end execution time, while respecting scheduling constraints (e.g., one kernel at a time per GPU, or prioritizing one kernel launch over another) we use event-driven simulation.

3.2 Search and Optimization Engine (SOE)

Co-optimizing micro-architectural parameters and the parallelization strategy that minimizes the overall end-to-end execution time requires navigating a large space of design parameters. Search and optimization engine (SOE) enables the automatic design space search and finds an optimal design point which meets the design constraints and minimizes the overall execution time. SOE takes inspiration from ML-assisted search algorithms, in particular gradient decent search with momentum and builds on top of the CrossFlow modeling engine.

3.3 Parallelism Strategy Space

There are a myriad of ways to parallelize a model across a large multi-node system. Exploring the parallelism space and finding the optimal strategy is critical to overall performance and system utilization. DeepFlow explores kernel, data and layer parallelism. It uniquely identifies each parallelism strategy by following notations: `RC-{KP1}-{KP2}-d{DP}-p{LP}` or `CR-{KP1}-d{DP}-p{LP}`

depending on the choice of kernel parallelism. RC (Row-Column) and CR (Column-Row) refer to different forms of kernel parallelism, i.e., distributed GEMM through inner-product or outer-product implementation. KP1 and KP2 are the parameters of distributed GEMM. For Row-Column (RC) or inner-product, KP1 and KP2 would refer to the number of ways we shard the first matrix across rows and the second matrix across columns. For Column-Row (CR) or outer-product, we would only need one parameter to specify the parallelization strategy; KP1 will refer to the number of ways we cut the first matrix across columns and the second matrix across rows. DP represents the number of model replicas and data shards assigned to each to exploit data parallelism. LP is the number of ways we cut layers into stages to exploit pipeline parallelism.

4 MICRO-ARCHITECTURE GENERATOR ENGINE

The micro-architecture generator engine, AGE, takes three sets of inputs: (1) A technology components library, where the characteristics of each component such as cores, different types of memories, network interfaces, and so on are defined, (2) Architecture template, where the overall high-level chip and system organization (such as compute and memory hierarchies) is provided, (3) Hardware resource allocation, where area, power, and chip perimeter budgets are provided for the different components of the system. Using this information, the AGE generates the final micro-architecture parameters (such as overall compute throughput, memory bandwidths at different memory levels, network bandwidth) as shown in Figure 2.

4.1 Technology Components Library

A system is generally composed of many primitive components or building blocks such as the compute units, SRAM banks, DRAM, interconnect network components (on-chip and off-chip),and the like. A library of these components and their associated technology parameters are provided as input to the tool through a $tech\_config$ YAML file. We classify these components into three primary categories: compute, memory and network.

4.1.1 Compute. Attributes for the minimal compute components such as matrix-multiplier units, vector-matrix multiply units, or a dataflow architecture unit like systolic array are specified under this category. When a compute component is added to the library, the compute attributes listed in Table 2 will have to be defined for that component. The tool user can add any type of compute component in the library ranging from a simple scalar unit to a complex unit comprising of a bundle of systolic arrays and capture the micro-architectural characteristics in the final architecture template file.

4.1.2 Memory. The memory components in a system can be built out of different technologies (e.g., SRAM, DRAM, MRAM, RRAM, 3D-XPoint). Also, these memory components can be used in two ways: on-chip memory and off-chip memory. A library of fine-grained memory components can be created and stored under this category which is utilized to construct different levels of the memory hierarchy. The characteristics of the on-chip components are described at the granularity of a bank because the smallest on-chip memory unit available to a system designer is usually a memory bank. The parameters of a memory bank such as capacity, bit area, periphery overhead, among others, are taken as inputs. On the other hand, we model the off-chip memory components such as DRAM, or 3D-XPoint at device level granularity, e.g., an HBM stack. This is because the off-chip components are usually obtained at a device level granularity. For off-chip memories, other parameters such as memory controller area, I/O bus width per device, and so on, need to be defined. This information is then used to precisely model the capacity and throughput of different levels of the memory hierarchy under the given area and power constraints.

Table 2. Different Technology Components

Compute	Technology Node	Nominal Area
	Nominal Voltage	Threshold Voltage
	Nominal Frequency	Minimum Voltage
	Nominal OP rate	Maximum Voltage
On-chip Memory	Technology	Latency
	Dynamic energy per bit	Static energy per bit
	Area per bit and total area overhead	Bank Capacity
	Controller area overhead per bank	Controller power overhead per bank
Off-chip Memory	Technology	Number of links per device
	Dynamic energy per bit	Nominal Voltage
	Static power per bit	Nominal Frequency
	Device Capacity	Minimum Voltage
	Device Area	Maximum Voltage
	Memory Controller and I/O Area	Access Latency
Network (intra-node and inter-node)	Nominal Voltage	Number of links per mm
	Nominal Frequency	Threshold Voltage
	Nominal Energy per Link	Minimum Voltage
	Nominal Area per Link	Link Latency

4.1.3 Network. The inter-chip network component is either intra-node or inter-node communication link. In the case of a **multi-chip module (MCM)** where multiple compute dies and memory devices are integrated on a 2.5D integration substrate within the same package, the inter-die communication is done using high density and energy-efficient links on the 2.5D substrate. These links are considered as intra-node links. On the other hand, the off-package communication links between nodes are considered as inter-node links. The attributes that need to be defined for inter and intra-die communication network components are provided in Table 2. In case of a waferscale system, the entire wafer could be considered as a single node.

4.2 Architecture Template

Once all system components are instantiated from the technology library, the next step is to hierarchically organize one or multiple components from each category to construct the overall system. Distributed machine learning training is done on scale-out multi-node system, as shown in Figure 3. Such a system consists of multiple individually packaged nodes which communicate through off-package interconnects (such as NVLink, Infiniband, etc.) that form the inter-node network. Inside each package, there can be multiple different accelerator nodes connected using an intra-node network. Each accelerator within the package typically consists of one accelerator die that is connected to its own off-chip main memory components (such as HBM, as shown in the figure). Each accelerator die itself can be composed of smaller compute units.

DeepFlow provides a rich template that can be used to specify the overall architectural organization of such an accelerator system. Next we describe in detail how the template is organized and how different system configurations can be achieved using this template.

4.2.1 Compute Unit. As shown in the accelerator die architecture in Figure 3, compute units are often organized in hierarchies. E.g., in an NVIDIA GPU, multiple tensor cores are bundled in a **streaming multi-processor (SM)** and the SM as a whole interacts with the cache hierarchy. In DeepFlow one can express such hierarchy by defining ***minimal compute units (MCUs)*** and *MCU bundle*. MCU is the smallest compute unit that we expose to the tool user. It defines the dataflow model and layout (e.g., MCU can be a systolic array that its height and width are configurable

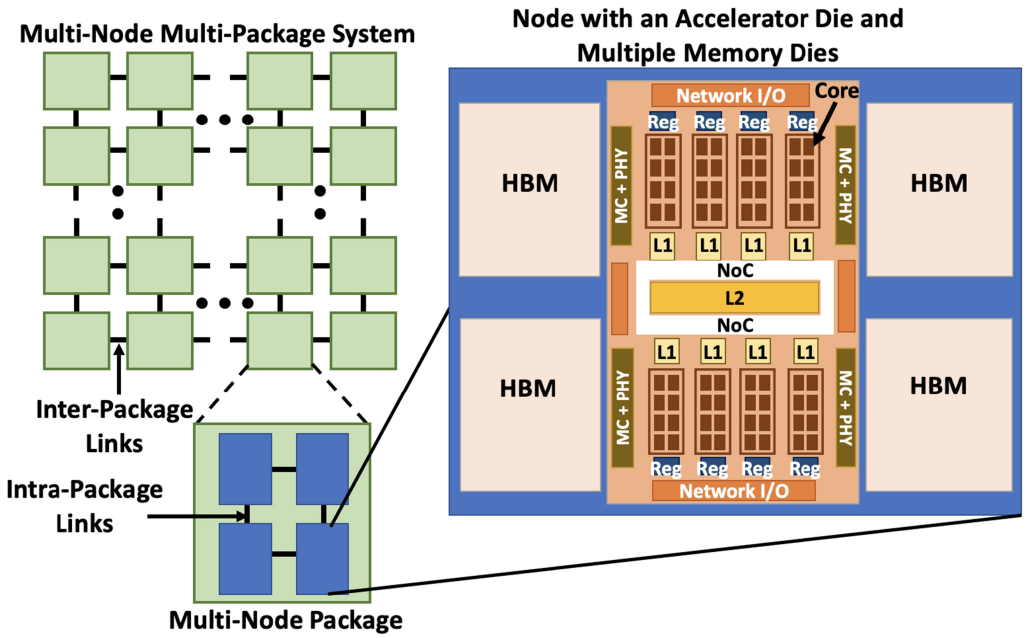

Fig. 3. Architecture template: Overview of a hardware system whose characteristics can be configured in DeepFlow.

as input) and interacts with the first level of memory hierarchy. Meanwhile, MCU bundle defines the number of MCUs that are bundled together and are exposed to the second level of memory hierarchy.

In dataflow architectures such as Eyeriss, TPU, and the like, data can flow directly between different cores. Hence, the tool allows one to define the type of dataflow within an MCU. Currently the performance model supports three types of dataflow: *weight stationary, activation stationary and output stationary*. The tool can also find the best dataflow strategy among the three for any given kernel.

Software runtime, scheduling overheads and the architecture of the cores often restrict the maximum compute utilization. For example, the tensor-cores in NVIDIA V100 incurs fill-drain related under-utilization during tensor loading from the registers and therefore achieves a maximum utilization of 85%. To account for such overheads, a maximum utilization value can be defined which derates the core throughput by that factor.

4.2.2 Memory Hierarchy and Scope. The memory hierarchy is defined by initializing multiple memory levels from the highest to the lowest level (i.e., registers to the main memory) as shown in Figure 3. Each level of memory has two attributes: (1) Memory technology component from the technology component library which defines the physical attributes of the memory as outlined in Table 2, and (2) Scope defines the set of components from the next level of memory hierarchy that are accessible from this level of memory hierarchy. For example, the 'global' scope indicates that the memory level is accessible to all the components.

4.2.3 Network Topology. In DeepFlow , we support two levels of network hierarchy: intra-package and inter-package. For each level, a different topology (e.g., mesh, torus, crossbar) can be defined.

```yaml
area_breakdown:
    node_area_budget: 1230 #mm2
    proc_chip_area_budget: 815 #mm2
    core: 0.35
    L2: 0.14
    L1: 0.1
    L0: 0.2
    DRAM: 0.05
    network:
      intra_package: 0.06
      inter_package: 0.1
```

Fig. 4. Resource breakdown example: This example is showing the area budget allocation and breakdown across all micro-architectural components.

4.3 Hardware Resource Allocation

Hardware design under a limited area and power budget is a fine art of finding the right balance (breakdown of resources) across different micro-architectural components. The area and power allocation for each micro-architectural component, as well as the perimeter allocation for certain components derive the design and specification of that component.

We define resource (area, power, perimeter) distribution across different components of the compute chip, as input parameters. The input definition also includes the total area and power budgets for the entire compute node. The total perimeter is inferred from area. The area budget is usually dictated by packaging constraints. For example, if the compute and memory dies are assembled on a 2.5D silicon interposer-based interconnect substrate, the total area of the node will be limited by the maximum size of the interconnect substrate that can be fabricated. Compare this to a wafer-scale system which houses an entire node on a wafer where the total area budget can be as large as 70,000 mm^2. A node's power budget is determined by the cooling infrastructure that extract heat from the node and the power delivery constraints.

We define budget distribution across different components of the compute graph as a percentage breakdown. As shown in the YAML snippet in Figure 4, fractions of the total area is distributed across cores, levels of memory hierarchy and network components. Similarly, the fraction of the compute chip's power and perimeter gets devoted to different hardware components.

Given the overall resource allocation and distribution, the AGE performs a series of optimizations (voltage-frequency scaling) to find an optimal parameter settings for each micro-architectural component. An optimal parameter setting is one that utilizes the most of the allocated budget. Note that an unbalanced resource allocation may leave some of the budget under-utilized. While we allow users to provide a manual breakdown of resources as input, we highly recommend to use SOE (Search and Optimization Engine) to automatically find the best setting which maximizes the overall resource utilization.

4.4 Micro-Architectural Parameter Generation

Next, the tool generates the micro-architectural parameters for each component of the architecture. Given the architecture template, alongside the resource breakdown across the different components, and the technology parameters, we find the maximum throughput for each component. E.g., we find the maximum number of cores that can fit in the given area allocation and find the voltage-frequency points to maximize compute throughput under the power budget. Similarly, for on-chip caches, we find the memory capacity and memory bandwidth at each level that can fit in the area budget while taking the network and controller overhead into account. For off-chip

memories and network interfaces, we use the energy per bit information along with the physical I/O transceiver area, bump pitch as well interconnect wiring pitch to determine the maximum bandwidth that can be realized on the chip (using a model similar to [13]).

These architectural parameters, throughput, bandwidth, capacity, and so on, are then provided as input to the performance prediction engine. Next, we discuss in detail how we model and calculate these parameters.

4.4.1 Core. For deep learning models, the kernels are usually highly parallel in nature and therefore, our goal is to maximize total compute throughput under the area and power budgets allocated for compute. Given the area budget, we first compute the maximum number of MCUs (minimal compute units, introduced in Section 4.2.1) that can fit within the area allocated. The nominal frequency and voltage for each MCU is an input to the model, therefore the nominal power for each MCU and the entire core can be derived very easily. If the nominal power exceeds the power budget, we scale down the frequency and voltage. If we hit the minimum voltage limit set in the component description, we reduce the number of MCUs till we satisfy the total power budget allocated to the compute units. This explains a case where the core design is power-bound and not area-bound.

Once we determine the total number of cores and the frequency of operation, we compute the compute throughput by appropriately scaling the nominal flop rate, as shown in Equation (1).

$$\texttt{Throughput} = N \times \texttt{flop}_{nominal} \times \frac{f_{op}}{f_{nominal}} \tag{1}$$

where N is the total number of cores, $\texttt{flop}_{nominal}$ is the nominal flop rate of each core, $f_{nominal}$ is the nominal frequency corresponding to the technology node of the core and f_{op} is the final optimal operating frequency. We use standard Voltage-Frequency-Power scaling methodology to obtain the operating voltage and frequency.

4.4.2 Register and Cache Memory. The total area and power budgets allocated to each level of on-chip memory is split between the memory banks and the network circuitry that connects the memory banks at each level to micro-architectural components at the next level that are under its scope. We assume this interconnect to have a crossbar topology. The total number of components under its scope and the number of banks in that memory level determine the area and power overheads of the network. We iteratively determine the total number of banks possible at each level of memory hierarchy such that the total area of the banks and the network at every level satisfies the area budget allocation. Once we determine the number of memory banks, we calculate total static power of all the banks (Equation (2)) and we allocate the remaining power budget to dynamic access energy. The available dynamic energy budget determines the maximum achievable throughput as shown in Equation (3).

$$P_{static} = P_{static-per-bit} \times N_{banks} \times \texttt{Capacity}_{bank} \tag{2}$$

$$\texttt{Throughput} = \frac{P_{on-chip-mem} - P_{static}}{\texttt{Energy}_{dyn-per-bit}} \tag{3}$$

4.4.3 Main Memory. Main memory has two major components that collectively control the overall capacity and bandwidth but are housed in two different places. Memory controller which is placed on the compute chip, and the memory devices are placed outside the compute die within the same package. The area allocation to each component determines the maximum number of memory devices that can be supported, which in turn determines the total memory capacity (see Equation (4)).

$$\#Devices = min\Bigg(\frac{Node\ Area - Processor\ Chip\ Area}{DeviceArea}, \frac{Area\ budget\ for\ Memory\ Controllers}{Memory\ Controller\ Area}, \frac{Perimeter \times \#Links\ per\ mm}{\#Links\ per\ device}\Bigg) \tag{4}$$

Meanwhile power and perimeter allocation dictates the number of links (that can fit along the compute die), and the frequency of each link which collectively determine the overall off-chip memory bandwidth.

4.4.4 Network. The off-chip network links (intra and inter-package) consume both power and area on the compute die. Moreover, the wires need to escape the periphery of the die which gets determined by the interconnect density and the available chip perimeter. The maximum number of links that can be accommodated in the compute die is limited either by the area available to fit in the link I/O cells or the amount of perimeter available for the links to escape the die periphery. Therefore, the tool uses the area per link, the available area budget, wiring density and the die perimeter budget to find the maximum number of links that can fit in the chip. Next, the tool uses the standard voltage-frequency scaling methodology to find the operating point for each link such that the total network-related power is within the power budget allocated. The network bandwidth is then calculated by multiplying the total number of links and the operating frequency of each link. We perform this step for the intra-node network and inter-node network separately.

5 COMPUTE GRAPH TRANSFORMATION AND DEVICE MAPPING

Given the ML model description (in the form of a *compute graph*) and the distributed system topology (in the form of a *system graph*), we find an optimal mapping from vertices and edges in the compute graph to hardware nodes and network links in the system graph. However, before mapping, we transform the compute graph into a *super-graph* to reflect the parallelism strategies specified as input.

5.1 Compute Graph Structure Transformation

Each parallelism strategy is a form of graph transformation where the sub-graph to be replaced is a single node, so essentially all nodes would be replaced with the same replacement graph. For example, to model data parallelism (with the ring-all-reduce implementation) we would need to *replace* each node in the original graph with a ring of length N (for an N-data parallel strategy). The new edges on the ring will be marked as *cross-edge* to capture the fact that they connect compute nodes hosted on separate devices. To capture a kernel parallelism strategy (e.g., `RC-{KP1}-{KP2}`), we would need to *replace* each node in the compute graph with a 2-dimensional torus of $\mathsf{KP1} \times \mathsf{KP2}$ dimension (assuming the reduction algorithm along each dimension is ring-all-reduce). Similarly, new edges on the torus would be marked as cross-edge. To capture a pipeline parallelism, no node transformation is required. The pipeline parallelism slices the original graph into multiple sub-graphs, each hosted on a separate hardware node. Edges connecting sub-graphs would be marked as cross-edge. Figure 5 shows the composition of multiple parallelism strategies applied in sequence (pipeline, data and kernel parallelism, respectively). G_0 is the original compute graph and G_4 is the final transformed graph.

5.2 Device Mapping and Routing Engine

Data parallelism, kernel parallelism and pipeline parallelism would require that each parallel shard to be hosted on a separate physical device. Hence, device mapping happens at the granularity of a parallel shard. We want parallel shards that are close in the parallel space to be mapped onto nodes that are close in the physical space to minimize communication. However, the transformed graph

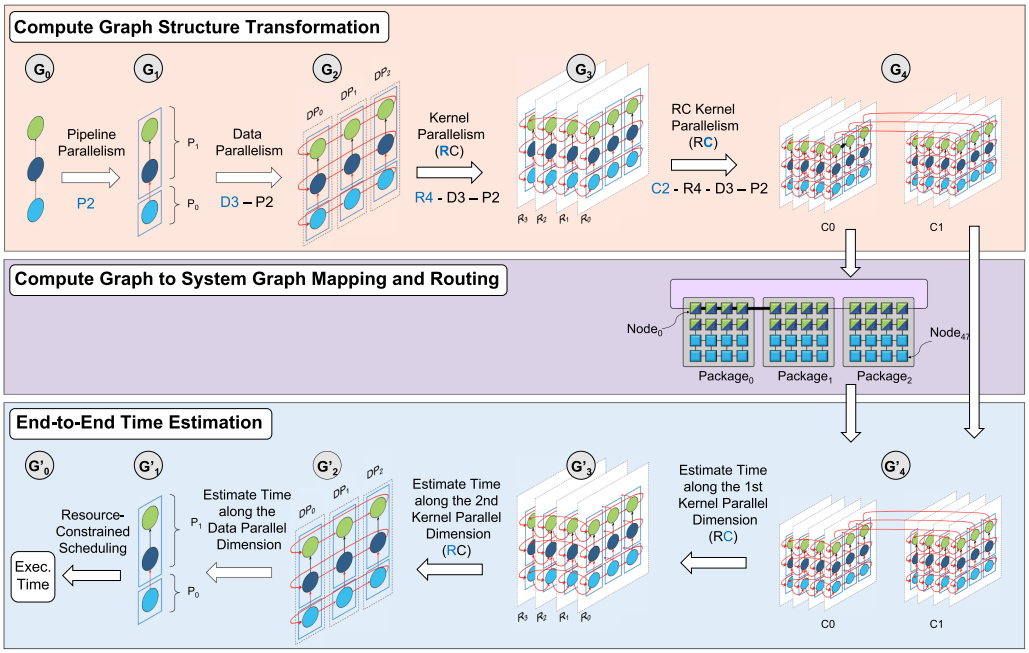

Fig. 5. An example of a compute graph transformation, device mapping and routing, and end-to-end time estimation: (top) Cross-edges are shown in red. To preserve readability, we only show a subset of cross-edges for kernel parallelism. Blue solid borderlines indicates separate hardware nodes. At every parallelization stage, we use black hashed lines to show graph replication along that dimension. A replica is a graph with a similar structure, however the kernel size and/or data size could be different for each replica. For simplicity, the original graph is a simple 3-layer feed-forward neural network that is divided into two sub-graphs (P2). Then for each pipeline stage, batch size is distributed across three workers (D3). Then for each data shard of each pipeline stage, the kernels are distributed in a row-column fashion across a 4×2 torus (RC-K4-K2). **(middle)** Mapping a 4-D hyper-cube into a 2-D mesh: a greedy layout mapped in the following order: kernel(R), kernel(C), pipeline and data. The bolded black edge in G4 is mapped onto a 4-hop path in the system graph. **(bottom)** backward pass time estimation.

usually has higher dimension than the system graph. Figure 5 shows such example, where the final transformed graph (G_4) is 4-D hypercube and the system graph is a 2-D torus. Therefore, it will not be possible to map all adjacent nodes in the compute graph to adjacent nodes in the system graph. We adopt a greedy approach to conduct such mappings: We start with a parallel dimension, map all parallel shards along that dimension to adjacent nodes in the hardware. If the number of shards along the parallel dimension is larger than the hardware dimension we are mapping onto, we wrap-around to the next immediate dimension. We continue this process along other dimensions in a specific order, until all nodes are mapped. The order at which we walk along the parallelism dimensions results in different mappings. For four different parallelism strategies, we explore (4!) = 24 possible orderings to pick the best mapping. Once node mapping is determined, we take a last step to map edges to physical links. An edge that connects to adjacent node in the compute graph may map to a multi-hop path as shown in Figure 5. As a result, one physical link would be shared across multiple edges. The number of logical edges sharing a physical link is an important factor for effective bandwidth estimation. We use $X - Y$ routing to map edges in the compute graph to paths in the system graph. Overall, the whole transformation step followed by device mapping is necessary to find an accurate estimation of *edge* timing.

6 PERFORMANCE PREDICTION ENGINE

Once the mapping is decided for each node and each edge in the transformed graph, performance prediction engine estimates timing for each node and each edge. We then use a resource-constrained scheduling algorithm to find the end-to-end timing.

6.1 Hierarchical Roofline

We use hierarchical roofline analyses [14] to predict the timing of each node in the transformed compute graph. For each node, we estimate the operational intensity ($\texttt{OI}_L$ = $\texttt{\#flops/\#memory accesses}_L$) to each level in the memory hierarchy. We search over the space of possible tiling strategies at each level of memory hierarchy and estimate the number of memory accesses to each level. We explain this in more detail next.

6.2 Memory Hierarchy Modeling

The number of accesses to each level of memory hierarchy is a function of the underlying hardware (memory capacity at each level) and the algorithmic implementation (loop ordering and tiling strategies).

For any given input configuration, we explore N^L random tiling strategies which meet the memory capacity requirement at each level. N is the number of tiling strategies at each level and L is the number of levels of memory hierarchy. Empirically, we found that for $L = 3$, $N \approx 20$ results in a reasonably accurate estimation.

For a given tiling strategy, it is easy to find the number of times each tile needs to be re-streamed from the next level of memory hierarchy. We start from the lowest level (main memory) and walk upward to estimate the number of accesses. The number of memory accesses at each level is dictated by the tiling strategy at current level and the higher level. For the highest level, the number of accesses is determined by the dataflow strategy exploited at MCU units.

6.3 DataFlow Model

The number of accesses to the highest level of memory hierarchy (i.e., register file) will be determined by the number of instructions executed in the execution engine and the dataflow strategy governing mapping and communication between those engines (e.g., weight stationary, activation stationary and output stationary [5, 15]). The execution engine structure dictates how many times a piece of data could be reused internally before accessing the register file. We refer to this number as *reuse factor (K)*. In a 2-D systolic array with size N_x and N_y, and an input GEMM with size $T0_x$, $T0_y$ and $T0_z$ at L_0, each data element could be reused $T0_x/N_x$ or $T0_y/N_y$ or $T0_z/N_z$ times, depending on which matrix is stationary. Given the reuse factor, we estimate the number of accesses to register files as follows:

$$\#RegAccess = \#Flops \times \frac{N_x.N_y + K.N_x + K.N_y}{2.K.N_x.N_y} \tag{5}$$

6.4 Inter/Intra-Package Communication Modeling

As discussed in Section 5, compute graph to system graph mapping captures logical edge to physical link mapping. The effective bandwidth for each link is downrated by the number of logical edges sharing the link.

6.5 End-to-End Time Estimation

We use an event-driven simulation to estimate end-to-end timing. Event-driven simulation is basically a resource-constrained critical path analysis. Since multiple compute nodes can map into the

same hardware node, event-driven simulation is necessary to avoid resource conflicts and respect resource scheduling constraints (e.g., not more than k kernels can run in parallel on each hardware node).

We apply event-driven simulation at the original compute graph where the only parallelism to account for is pipeline parallelism: data parallelism and kernel parallelism would essentially create replicas of the original graph (where the kernel size and/or data size would be different for each node). Given that all replicas by definition are hosted on separate hardware nodes, they can all start and stop at the same time (assuming a homogeneous distribution of data along model replicas and homogeneous distribution of sub-kernels across data replicas) and their timing is deterministic. Hence, there is no need for event-driven simulation at the super-graph granularity.

Figure 5 explains an example of an end-to-end time estimation of a backward pass for a simple 3-layer feed-forward neural network, with 2-level pipeline parallelism ($p2$), 3-level data parallelism ($d3$), and 8-level kernel parallelism (`R4-C2`).

7 DESIGN SPACE EXPLORATION ENGINE

We denote the set of hardware ***parameters*** to explore as $W = \{\{A_i\}_0^{H-1}, \{P_i\}_0^{H-1}, \{R_i\}_0^{H-1}\}$, where H is the number of micro-architectural components in the hardware accelerator node, and A_i, P_i and R_i capture the percentage of the overall area, power and perimeter allocated to each component, respectively.

Our **objective** is to find the optimal W^* that minimizes the total run time, $f(W)$, such that $\sum_{i=0}^{H-1} A_i \leq 1$, $\sum_{i=0}^{H-1} P_i \leq 1$, and $\sum_{i=0}^{H-1} R_i \leq 1$. The objective function f does not have a closed form, but we can calculate it by querying the performance model (CrossFlow). This problem is an example of a *constrained black-box continuous* optimization. Since the objective function evaluation (i.e., querying CrossFlow) is considerably cheap (milliseconds), we use a variation of projected **gradient descent (GD)** optimization to solve for W^* (see 6). Empirically, we found that GD with exponential averaging in the parameter space (rather than gradients) works the best for our problem.

$$
\begin{aligned}
&W_t = W_{t-1} - \eta \mathrm{g_t} \qquad \hat{W_t} = \frac{W_t}{||W_t||} \\
&M_t = \beta M_{t-1} + (1-\beta)\hat{W_t} \\
&W_t = \mathrm{Project}(M_t) \quad \text{onto} \quad C_A, C_P, C_R
\end{aligned}
\tag{6}
$$

Where W_t and $\mathrm{g_t}$ are the input parameters and gradients at time step t, η is the learning rate and β is the discounting factor. We repeat the update steps shown above until convergence or the maximum number of steps (T), whichever conditions happen earlier. The final result is very sensitive to initialization. We repeat the steps above from S different starting points and return the best result. Empirically, we found that $T = 100$ and $S = 10$ are sufficient to find a near optimal solution.

8 VALIDATION

We validate our performance prediction model against execution time measured on real systems (Nvidia P4 with 1 GPU and an NVIDIA DGX-1 system with 8 V100 GPU cards), running distributed GEMM as well as large-scale language models. In particular, we study (2-layer LSTM) **language models (LM)** for validation and case study as it is deemed to be one of the most challenging applications to scale [16], and is very costly to train [17]. All applications are implemented in TensorFlow 2.0. We use CrossFlow to predict the runtime, which can take anywhere from milliseconds to 20 seconds.

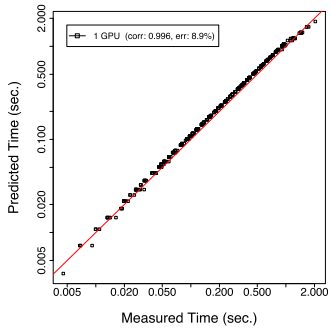

Fig. 6. GEMM Validation on P4.

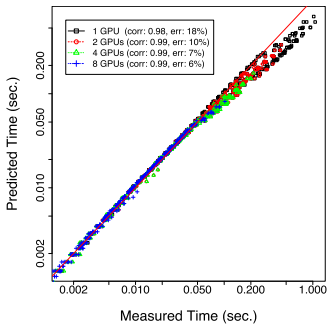

Fig. 7. GEMM Validation on DGX.

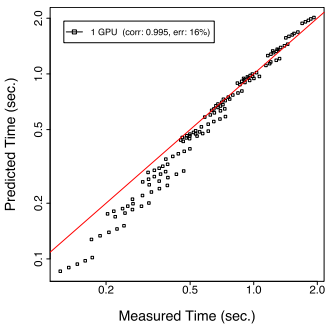

Fig. 8. LM Validation on V100.

For GEMM validation, we look at a space of more than 2,000 GEMM kernels of different shapes and parallelism strategies, where input (`m`), output (`n`) and inner dimensions (`k`) varying from 4K to 32K in steps of 4K, and parallelized across 1, 2, 4, or 8 GPUs, using both Row-Column and Column-Row distributed parallelism strategies. Note that we particularly focus on a variety of GEMM implementations as most modern machine learning workloads (BERT, Transformers, as well as CNNs) are primarily composed (often greater than 85%) of GEMM kernels, and their performance is determined by the performance of GEMMs and inter-kernel communication. For LM validation, we look into a space of 125 configurations, where `Batch Size`, `Hidden Dimension` and `Vocab Size` varying from 2K to 6K in steps of 1K. We report the correlation (corr), and also the mean relative error (err) to quantify the quality of our predictions.

Figure 6 shows the validation results on Nvidia P4 GPU card. On the X-axis, we show the measured time (in log-scale), and on the Y-axis, we show the predicted time (in log-scale). As shown, predictions and measurements are highly correlated (0.996) and the error is small (8.9%). Figure 7 shows that CrossFlow predictions on a DGX-1 system across 1, 2, 4 and 8 V100 GPU cards are well correlated (0.98–0.99) and have low error (6%–18%). Figure 8 shows the performance of LM on V100 GPU card. Similarly, we can predict performance with high correlation (0.996), and low error (16%). A constant pattern visible across all results is the performance prediction deviation from measurement on real hardware for small kernels. This is expected as TensorFlow 2.0 time measurement hooks include all the software stack latency; while this overhead is negligible for large kernels, it accounts for a large portion of total run-time if the kernel is very small. Also control flow overheads in hardware architecture introduces additional latency, which leads to higher overhead and error between our model and hardware performance for smaller kernels. Therefore, the tool outcome would be more reliable for large kernels and large models. Further improvements in accuracy would require careful modelling of control overheads (e.g., SM scheduler, L2 interconnect control flow and memory coalescer, etc.). However, such data is often proprietary and hard to model.

9 CASE STUDIES

DeepFlow is a pathfinding framework with studies and use cases spanning semiconductor technology development, micro-architecture, neural network models, and algorithmic parallelization techniques. In this section, we give a few example case studies for a large-scale language model (hidden dim: 16K, global batch size: 16K, vocab size: 800K, number of layers: 2, sequence length: 20) distributed across 512 hardware nodes. For future technology exploration, we study seven consecutive **logic** technology nodes (from 12nm (N12) to 1nm (N1). Based on the recent scaling trends for logic technologies [18, 19], we assume area and power scale by 1.8× and 1.3× from one node to the

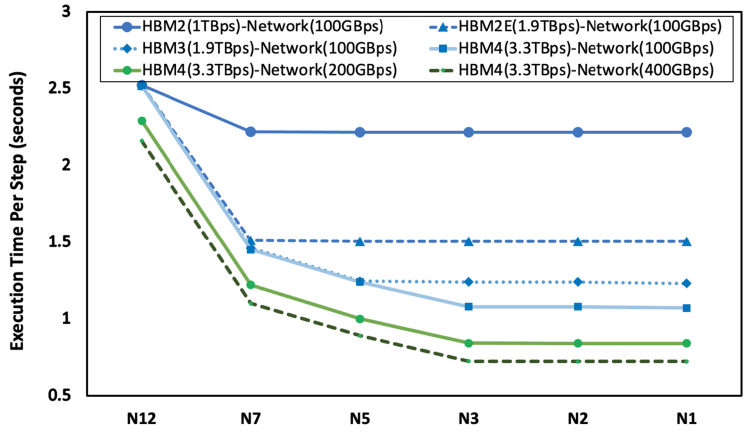

Fig. 9. **Technology scaling:** scaling logic, memory and network technology.

next for iso-performance), four different **memory** technologies (HBM2 (1 TB/s), HBM2e (2 TB/s), HBM3 (projected 2.6 TB/s [20]), and HBM4 (projected 3.3 TB/s)) and three different **network** technologies (Infiniband-NDR-x8 (100 GB/s), XDR-x8 (200GB/s) and GDR-x8 (3.3 TB/s)). The caveat to these results (as with any pathfinding study with DeepFlow) is that if the system architecture or dataflow or neural network is radically different (e.g., this study assumes that same node is homogeneously replicated within the package), the conclusions may change.

9.1 Impact of Technology Scaling

The first question we seek to answer is where the performance bottlenecks are across the stack and which technology could provide the maximum end-to-end performance benefit? Semiconductor technology development decisions are increasingly driven by machine learning as the workload. Many of these decisions trigger large, multi-year investments. Figure 9 shows the impact of scaling logic, memory, and network technology for a large-scale language model using data-parallelism. For these experiments, we assume that power/node = 300W and area/chip = 850 mm^2.

Logic scaling improves compute throughput, and also caching capacity and bandwidth, but only to a smaller extent. Going from N12 to N7, we observe a jump in performance irrespective of memory technology. This is because at N12, the performance of a significant number of kernels are L2 bandwidth bound. At N7, the L2 bandwidth and capacity improve enough for HBM bandwidth to become the new bottleneck. Therefore, with improvement in HBM bandwidth, the balance can shift back again to caches and saturation point can be further improved with logic scaling, hence saturation point shifts further to the right. This trend continues up to N3. Beyond N3, even at very high memory bandwidth (3.3 TB/s) and network bandwidth (400 GB/s) performance stays unchanged as cache capacity and bandwidth are the main bottlenecks. Since the on-chip network connecting MCUs to cache and the cache controller overhead scale along with number of cache banks and the number of MCUs (which scale at ~1.8× per technology node), the cache capacity as well as bandwidth increase only marginally at N2 and N1. These trends are well inline with commercial examples from NVIDIA and AMD, where jump to N7 node provided large performance benefits and then, multiple high-end SKUs of the GPUs with higher bandwidth HBM memories have been released for further performance improvements.

Network technology scaling is another big factor that determines overall end-to-end performance of a distributed deep learning system. As logic and memory technologies scale alongside the size of the models, more inter-node bandwidth is needed to accelerate the inter-node

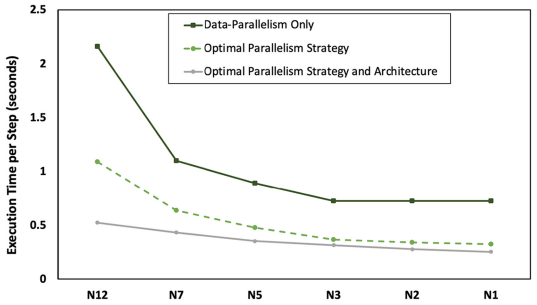

Fig. 10. Co-optimizing parallelism strategy and hardware architecture design.

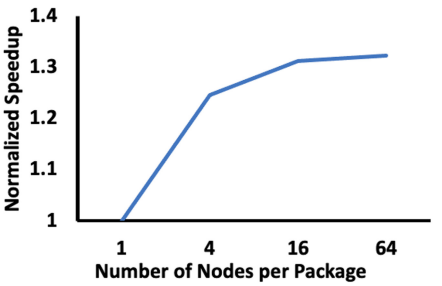

Fig. 11. Performance improvement from multi-node package.

communication collectives. Our analysis (Figure 9) shows that beyond N3, scaling networking technology will provide much larger performance gains as opposed to logic scaling. This trend also aligns with the recent efforts in the industry to push high bandwidth and low energy networking technologies and architectures for inter-node and intra-node communication, targeted towards deep learning systems [21–23].

9.2 Co-Optimizing Technology, Parallelism Strategy and Hardware Architecture Design

Figure 10 shows the importance of co-optimizing technology with parallelism and hardware design in an incremental fashion. As shown: (1) Parallelism strategy optimization alone can offer ~2× performance improvement. (2) Co-optimizing architecture and parallelism strategy offers meaningful benefits for mature technology (12nm and 7nm) nodes. But for more advanced technology nodes, only marginal benefits (20%–30%) can be gained on top of parallelism strategy optimization. (3) For current and near-future technology nodes, co-optimizing for model architecture can provide as much benefit as scaling technology nodes (by almost two generations).

Next, we evaluate the performance improvement that multi-node packaged systems (e.g., MCM-GPU [24], waferscale-GPU [25], Tesla Dojo [26]) can provide in a distributed training setup (see Figure 11). We assumed 2TB/s link bandwidth for the intra-package links and performed both parallelism and architecture search for each case.

A couple of key takeaways from these experiments were: (1) Increasing the number of nodes in a package improves overall performance by roughly 32% at best. (2) Beyond 4-nodes per package, performance improvement is marginal. Since ultra-large packages or waferscale integration dramatically worsens cost, we believe that such technologies may not be worthy investments for scaling large language model training. These conclusions hold across multiple different batch sizes, hidden dimension sizes and intra-node link bandwidths.

10 RELATED WORK AND DISCUSSION

Related work can be broadly categorized into (1) performance modeling frameworks for spatial architectures like TimeLoop and Maestro, (2) performance modeling frameworks for parallelism exploration such as FlexFlow, and (3) algorithm search and analysis tools like DayDream and Habitat.

Similar to TimeLoop [5] and Maestro [6], we use an analytical model to estimate performance, however, the scope of DeepFlow is much broader. TimeLoop and Maestro model a single kernel runtime on the spatial architecture like systolic array or Eyeriss. Similarly, Mind Mapping [7] is a gradient based search tool that finds the best tiling and mapping strategy for a single compute unit and is built on top of TimeLoop. In this regard, all these prior work are similar to analytical

models that goes into DeepFlow's MCU modeling. However, DeepFlow offers more than MCU modeling. DeepFlow allows to capture not only the behavior of an MCU unit but also an entire GPU (through modeling of communication across MCU units through shared layers of memory hierarchy) as well as modeling a data center full of GPUs. Besides, prior work validates against simulators on micro-kernels. We validate our model against SOTA GPU hardware on real-world applications. Furthermore, DeepFlow models an entire compute graph, composed of many kernels mapped and distributed across multiple GPU nodes, and allows the analysis of parallelism at this level, including pipeline, data and kernel parallelism. Moreover, DeepFlow provides four degrees of freedom to explore: model architecture, hardware architecture, technology configuration and parallelism strategy.

FlexFlow [8] is an ML-based model for exploring the best parallelism strategy which relies on the runtime profiling tools to measure kernel timings on the target hardware. While it provides a very rich input for expressing different model architectures, it can only model existing hardware, hence it is not suitable for parallelism-architecture-technology co-design exploration.

DayDream [9] is a what-if analysis tool that enables researchers to evaluate the efficacy of different *algorithmic* optimizations for an *existing* hardware. However, it relies on fine-grain profiling tools to construct dependency graph, hence it lacks the ability to predict individual kernel runtime on non-existing hardware and cannot be used for architecture or technology co-design space exploration. Similarly, Habitat [10] predicts deep learning workloads' run-time across different *existing* GPUs, using a combination of wave scaling and MLP predictors. Wave scaling can only model simple μ-architectural modification, and MLP predictors are μ-architecture specific models that require collecting a large set of runtime data on the baseline and target hardware for model training, hence they cannot be applied to non-existing hardware.

ASTRA-sim [11] is a simulator for hardware-software co-design of distributed deep learning systems. The focus of the paper is on detailed modelling of the inter-node network and they study the effects of network topologies and architecture choices. ASTRA-sim doesn't explore automated technology and architecture exploration and may not be suited for across the stack design space exploration because of the detailed and heavy-weight focus on network effects.

Finally, we discuss how DeepFlow and/or CrossFlow can be used for future architecture research. The framework developed here can be extended to model other Von-Neumann and non-Von-Neumann architectures, since the performance model is based on roofline modeling. On the hardware modeling side, new technologies and nodes can be modeled similar to the existing ones for rapid design-space exploration. For better fidelity of certain aspects, different portions of the framework could be extended to replace the existing modules. We believe that DeepFlow is a foundational framework for early and rapid exploration of technologies across the stack for deep-learning workloads.

11 CONCLUSION

We proposed DeepFlow, a performance modeling framework that enables a cross-stack analysis for hardware-software-technology co-design at-scale. We envision DeepFlow to be used by *ML practitioners* (to decide what hardware to use to maximize their utilization, or simply predict their hypothetical model architecture performance which might not be realizable in today's hardware for many reasons including capacity limitation), by *system designers* (to decide what hardware accelerators they need to acquire or build from scratch to meet their application needs, what new technologies to invest in, etc.), and finally by *technology experts* (to guide future technology development by assessing its impact all the way across the stack, at scale). Our future work plans to extend DeepFlow modeling to other applications beyond language models and GEMM kernels.

REFERENCES

[1] OpenAI. AI and Compute. ([n. d.]). https://openai.com/blog/ai-and-compute/

[2] Kunle Olukotun. 2020. Accelerating Software 2.0. *ScaledML* (2020).

[3] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[4] Amazon AWS Inferentia. (accessed Sep. 10, 2021). Achieve 12x Higher Throughput and Lowest Latency for PyTorch Natural Language Processing Applications out-of-the-Box on AWS Inferentia. https://tinyurl.com/3mbuetmr (accessed Sep. 10, 2021).

[5] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. DOI: http://dx.doi.org/10.1109/ISPASS.2019.00042

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

[7] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: Enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 943–958. DOI: http://dx.doi.org/10.1145/3445814.3446762

[8] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 553–564. DOI: http://dx.doi.org/10.1109/HPCA.2017.29

[9] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately estimating the efficacy of optimizations for {DNN} training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 337–352.

[10] X. Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 503–521.

[11] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 81–92. DOI: http://dx.doi.org/10.1109/ISPASS48437.2020.00018

[12] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. (2023). arXiv:cs.DC/2303.14006

[13] Saptadeep Pal and Puneet Gupta. 2020. Pathfinding for 2.5D interconnect technologies. In *System-Level Interconnect - Problems and Pathfinding Workshop (SLIP '20)*. ACM, New York, NY, USA, 8.

[14] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. DOI: http://dx.doi.org/10.1145/1498765.1498785

[15] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. DOI: http://dx.doi.org/10.1109/JSSC.2016.2616357

[16] Joel Hestness, Newsha Ardalani, and Gregory Diamos. 2019. Beyond human-level accuracy: Computational challenges in deep learning. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 1–14.

[17] Deep Learning's Diminishing Returns. ([n. d.]). https://spectrum.ieee.org/deep-learning-computational-cost Accessed: 2021-10-15.

[18] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. DOI: http://dx.doi.org/10.1016/j.vlsi.2017.02.002

[19] Wikichip: Technology Node. ([n. d.]). https://en.wikichip.org/wiki/\technology_node Accessed: 2021-10-15.

[20] HBM3: Big Impact on Chip Design. ([n. d.]). https://semiengineering.com/hbm3s-impact-on-chip-design/ Accessed: 2021-10-15.

[21] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: Transforming Google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 66–85. DOI: http://dx.doi.org/10.1145/3544216.3544265

[22] Ryohei Urata, Hong Liu, Kevin Yasumura, Erji Mao, Jill Berger, Xiang Zhou, Cedric Lam, Roy Bannon, Darren Hutchinson, Daniel Nelson, Leon Poutievski, Arjun Singh, Joon Ong, and Amin Vahdat. 2022. Mission Apollo: Landing Optical Circuit Switching at Datacenter Scale. (2022). DOI: http://dx.doi.org/10.48550/ARXIV.2208.10041

[23] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/ (2022).

[24] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 320–332. DOI: http://dx.doi.org/10.1145/3079856.3080231

[25] Saptadeep Pal, Daniel Petrisko, Matthew Tomei, Puneet Gupta, Subramanian S. Iyer, and Rakesh Kumar. 2019. Architecting waferscale processors - a GPU case study. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 250–263. DOI: http://dx.doi.org/10.1109/HPCA.2019.00042

[26] Tesla Dojo. ([n. d.]). https://www.nextplatform.com/2022/08/23/inside-teslas-innovative-and-homegrown-dojo-ai-supercomputer/ Accessed: 2022-10-15.

Received 27 June 2023; revised 29 October 2023; accepted 4 November 2023