Performance verification for AI Heterogenous Multicore Systems using Portable Stimuli Standard

Pietro Locci, Hillel Miller

***Abstract*- This paper leverages the advanced features of Portable Standard Stimulus (PSS) to verify performance of highly configurable heterogenous multicore systems for AI applications. The increased complexity of AI SoC and software required to efficiently control the HW resources, demand sufficient test software available from pre-RTL till post silicon validation. A key aspect of this work is addressing the dynamic nature of AI applications, where rapidly emerging applications, with diverse performance demands, require a highly configurable testing infrastructure. In this work, we demonstrate how the modelling capability of PSS can be utilized to map application algorithms ensuring optimal scheduling and HW resources utilization for performance verification and how test portability enable efficient verification reuse.**

I. Introduction and motivation

The rapid evolution of AI applications for edge devices in various domains such as mobile, cameras, automotive, IoT, etc., has driven the development of increasingly complex SoCs, which integrate a diverse set of processing elements.

These elements include general-purpose embedded cores for control logic, vector DSPs for signal processing tasks, and specialized hardware accelerators designed to speed up AI computations. The heterogeneous nature of these systems poses new challenges for performance verification, as each component may have different execution models, data flows, and memory hierarchies.

Traditional verification approaches, such as UVM based testing or direct C tests, often fall short in capturing the real-world use-cases, while full software support may be available only at a later stage of the project. Moreover, the evolving nature of AI algorithms necessitates a more dynamic, adaptable and scalable verification approach. Portable stimuli, which can generate reusable, platform-agnostic test cases, provide valuable solutions for these challenges. By abstracting the specifics of the execution platform and properly modelling the available resources (memories, executors and HW other resources), portable stimuli enable end-to-end project lifecycle performance verification from architecture exploration to post-silicon and across different environment including virtual platforms, simulation, emulation and silicon.

In the following sections, fundamental concepts of AI applications are introduced to establish a foundation for understanding the applied methodology. This is followed by an overview of the (DUT) and the mapping of applications. Subsequently, the use of Portable Stimulus Standard (PSS) for application modeling, scheduling optimization, and the importance of portability are discussed. Finally, results are presented alongside an examination of the challenges across different verification levels, with recommendations for extending the methodology to more complex applications and relevant software considerations.

II. Neural network applications Concept

Before diving into the testing methodology, let's first introduce some basic concepts of Neural Network applications required to understand how tests are defined and optimized.

Neural networks (NNs) are computational models consisting of interconnected layers that transform input data through successive operations to extract meaningful patterns. Taking as reference ResNet50, the network takes as input an image and processes it through a series of layers, identify patterns such as edges, shapes, textures and ultimately classify the image between object classes. A Neural Network **Layer** is a computational block that transforms input data, producing an output called a feature map, which captures specific patterns or characteristics in the data. Layers consist of operations such as convolutions, pooling, and activation functions, each designed to learn different aspects of the input. A **Feature Map** is an essential component in CNNs, representing the output of a layer after applying specific operations to the input data or previous layer's output feature map. A feature map is represented as a three-dimensional tensor. A **Tensor** is a multidimensional array (3D), the dimensions are defined as

- Channels – number of different features, where each channel represents a unique feature
- Height – the vertical spatial dimension
- Width – the horizontal spatial dimension

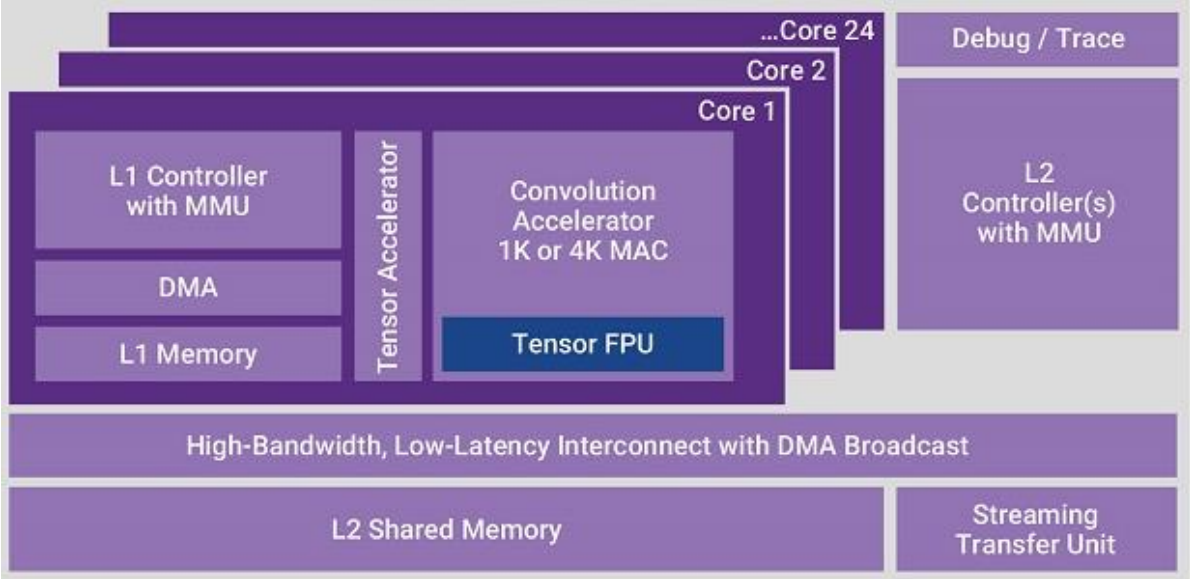

Figure 1: system under test

In the subsequent sections, the channel first notation will be adopted as follow Fmap(C, H, W) represents a tensor with C channels, H on the vertical dimension and W on the horizontal dimension E.G.: Fmap (64, 272, 480).

The fundamental operation in CNNs like ResNet-50 is the **convolution**, a mathematical operation where a small matrix called a filter or kernel slides across the input tensor, performing element-wise multiplications and summing the results to generate the output. Each filter in a layer has associated weights, which are trainable parameters learned during training. The weight values determine the specific features each filter detects**.** The convolutional filter weight are represented with following notation W<nF × inC × fH × fW> where

- nF represent the number of filters applied to the input feature map
- inC the number of channels of the input feature map
- fH and fW represent the heigh and the weight of the filter and are typically 1x1, 3x3 or 7x7 in Resnet50

Ignoring concept like padding, filter dilation or stride, not required for the purpose of this paper and and applying a convolution with filter W <nF × C × H × W> to an input feature map F_in(C, H, W), the results will be F_out(nF, H, W), witg number of channel equal to the num_filters, and spatial dimension equal to F_in.

III. Heterogeneous SoC Architecture for AI Applications

SoCs for AI applications integrate multiple types of heterogenous processing elements in a single system to optimize performance, energy efficiency and flexibility. Unlike traditional homogeneous systems that rely solely on one type of processor, typically a CPU, heterogeneous computing systems combine different processing units each optimized for particular tasks. This combination allows the system to leverage the unique strengths of each processor type to handle a diverse range of workloads more effectively. Fig. 1 shows the architecture of the test-system (Synopsys ARC NPX) with the following processing units:

- General-purpose processors, L1 and L2 cores, handle control and management tasks. These cores manage memory, and orchestrate data movement between different components
- AI Accelerators, Convolution accelerator, Tensor Accelerator and Tensor FPU. These components are responsible for the compute load of the System
- Cores 1 to 24 are homogeneous computing elements capable of executing the same algorithm on different data to leverage data-level parallelism. They can also run different algorithms, either independently or in coordination with other cores in a pipeline, or even combine these approaches.
- DMAs and Streaming Transfer Unit, responsible for the data movement across the system.
- L2 controllers are responsible for the management of the cluster (L2 memory, streaming transfer unit) and the synchronization of the different cores.

- Vector DSPs, designed for high-throughput signal processing, and optimized for various generic tasks commonly found in neural networks.

Another key aspect of the system under test is the memory hierarchy to balance speed and capacity, ranging from high-latency, large-capacity L3 memory to small, high-speed L1 and L2 memories closer to the processing units. L3 (external memory) is used for bulk data storage due to its capacity but incurs high latency. In contrast, L1 memory is

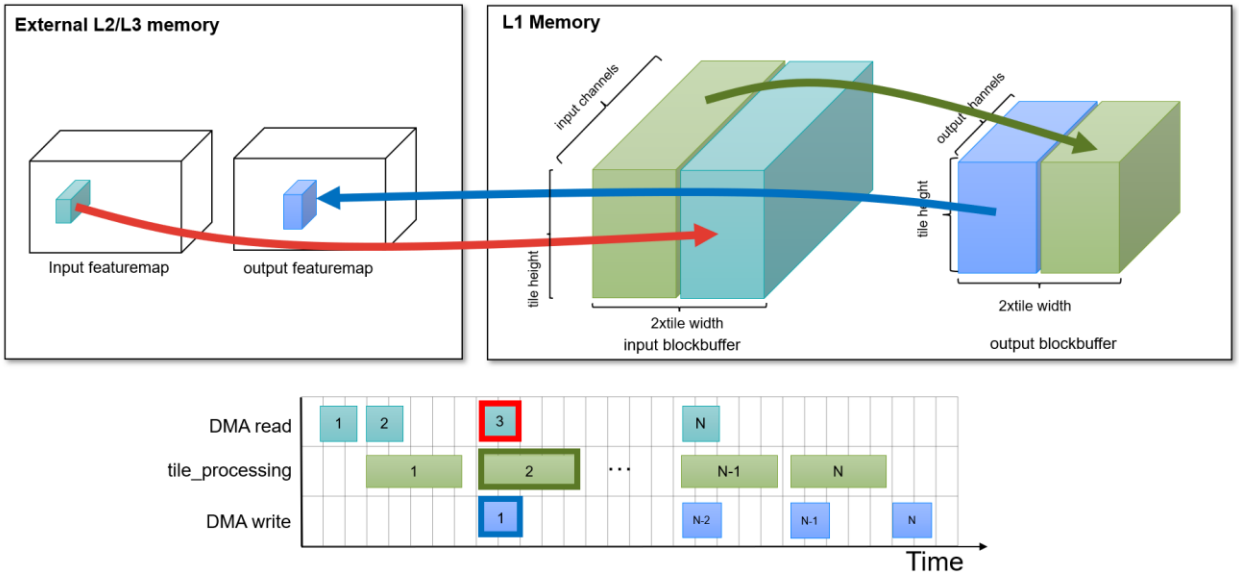

Figure 2: data flow and pipelining

small, private to each core and can directly be accessed by the processing units. L2 memory offers a balance, being larger than L1 and shared across cores with moderate latency, providing buffering capability to overcome high DDR latency and facilitate efficient data transfer between the DDR and processor cores.

All components are configurable to adapt to the area, performance and power requirement of the final chip – e.g.: memories size, Convolution Accelerator number of multiplier and accumulator

IV. Data flow and SW pipelining

In embedded hardware systems for AI applications, efficient data flow and software (SW) pipelining are essential for optimizing performance when mapping neural network computations. Considering a single-core application mapped into the system in Fig. 1. Core 1 subsystem includes two Direct Memory Access (DMA) engines, one input DMA used to fetch data from external (L2 or L3) memory into L1 memory in small chunks (tiles) and one output DMA used to store the results from L1 memory into external. In between these two data move operations the different TPUs (Tensor Processing Units such as convolution engine and generic tensors accelerator) perform a data elaboration. The DMAs and TPUs are controlled by the L1 controller.

To support continuous processing without stalls, double buffering is applied by partitioning the L1 memory into four segments: two for input buffering (allowing one segment to load new data while the other is used in computation) and two for output buffering (allowing one segment to offload completed data to external memory while the other holds intermediate results from ongoing computations). This partitioned setup allows SW pipelining to operate smoothly, with input DMA, compute, and output DMA stages continuously overlapping. Consequently, the system minimizes idle cycles for each component, maintaining a streamlined data flow and enhancing overall processing efficiency.

```
action base_action {
    input   buffer_s in_descr;
    output  buffer_s out_descr;
    input   buffer_s src_b;
    input   buffer_s dst_b;
    output  buffer_s out_b;
}
```

```
action  base_compound_action : base_action {
    base_start_action start;
    base_wait_action wait;
    activity{
        start;
        wait;
        bind start.out_descr wait.in_descr;
        bind in_descr start.in_descr;
        bind out_descr wait.out_descr;
        bind start.out_b wait.dst_b;
        bind wait.out_b out_b;
    }
}
```

```
schedule{ // Schedule the runtime application
    { // IDMA
        replicate(i: NUM_ITER){
            idma_move[i] with {…
            }
            if (i > 1){
                bind idma_move[i -1].start.out_descr idma_move[i].in_descr;
                bind fmap_in.out_buf idma_move[i].src_b;
                bind idma_move[i].dst_b comp_ker[i-2].out_b;
            }

        }
        bind fmap_in.out_buf idma_move[0].src_b;
        bind fmap_in.out_buf idma_move[1].src_b;
        bind idma_descr.out_buf idma_move[0].in_descr;
        bind idma_move[1].in_descr idma_move[0].start.out_descr;
        bind l1_buf_in_even.out_buf idma_move[0].dst_b;
        bind l1_buf_in_odd.out_buf idma_move[1].dst_b;
    }
    {  // COMPUTE
        replicate(i: NUM_ITER){
            comp_ker[i] with {
                stream_id == {i%2}+1;
            }
            if (i > 1){
                bind comp_ker[i -1].start.out_descr comp_ker[i].in_descr;
                bind comp_ker[i].dst_b odma_move[i-2].out_b;
            }
            bind comp_ker[i].src_b idma_move[i].out_b;

        }
        // bind the input buffers for the first two compute kernels
        bind comp_ker[0].in_descr comp_descr.out_buf;
        bind comp_ker[1].in_descr comp_ker[0].start.out_descr;
        bind comp_ker[0].dst_b l1_buf_out_even.out_buf;
        bind comp_ker[1].dst_b l1_buf_out_odd.out_buf;

    }
    {  // ODMA…
    }
}
```

Figure 3: PSS actions interfaces and runtime scheduling

V. TEST IMPLEMENTATION APPROACH

The Portable Stimulus Standard (PSS) model combining the HW configuration parameters of the system, along with external factors such as memory bandwidth, to facilitate the generation of tests and optimize the scheduling of various actions. As shown in Figure 2, the PSS model can be expressed using three types of actions with standardized interfaces across components (see Figure 3). These interfaces enable the buffer_s flow object to enforce dependencies between actions, ensuring correct execution for each solution. The key components of the model are:

1. base_action: This is the primary action class for all atomic actions required for testing (e.g., idma_start, idma_wait, compute, and odma). In "_start" actions, the Level 1 (L1) controller sets the descriptor and sends it to the accelerators. In "_wait" actions, the L1 controller waits for an event response from the accelerator. “_wait” and “_start” action deferred for the random members and the body.
2. base_compound_action: This action defines the data flow between start and wait actions for each component (i.e., idma, compute engine, and odma), managing dependencies within each individual component.
3. Application Action: This action schedules and manages dependencies across all actions, enabling functional testing with a range of legal schedules. Figure 3 illustrates the runtime scheduling process, which can generate multiple valid test schedules, though only one will be optimal.

To illustrate, consider a simplified scheduling problem involving four actions, idma_start, idma_wait, compute_start, and compute_wait, where idma and compute operate on independent data.

Figure 4 displays three possible schedules. The L1 controller executes _start actions at predictable intervals to dispatch jobs to the accelerators, while _exec actions depend on the job duration on each accelerator (with comp_exec time typically longer than idma_exec). _wait actions occur as the accelerator completes each job and signals the controller.

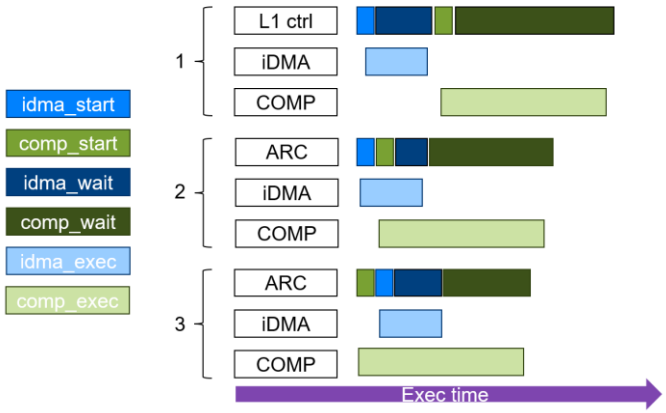

Figure 4: scheduling optimization

- **Solution 1**: idma_start is scheduled first, followed by idma_wait, then comp_start, and finally comp_wait. This sequence is inefficient, as idma and compute tasks are executed sequentially, underutilizing parallelism.
- **Solution 2**: In this case, comp_start follows idma_start, allowing both accelerators to operate in parallel, which improves efficiency compared to Solution 1.
- **Solution 3**: This optimal solution schedules idma_start first, followed by idma_wait and comp_wait, fully exploiting parallelism between idma and compute.

From this example we can extract three simple rules:

1. Start functions should have scheduling priority over wait functions.
2. Among start functions, those for accelerators with higher expected execution times, referred as cost functions, should be prioritized.
3. For wait actions, prioritize those associated with accelerators with lower cost function.

The first step to validate these rules is determining the cost function of a large number of compute kernel and determine in which cases the priority scheduling can be applied. Specifying action priority for PSS schedule activity operator is beyond the PSS standard and was done using an underlying tool mechanism

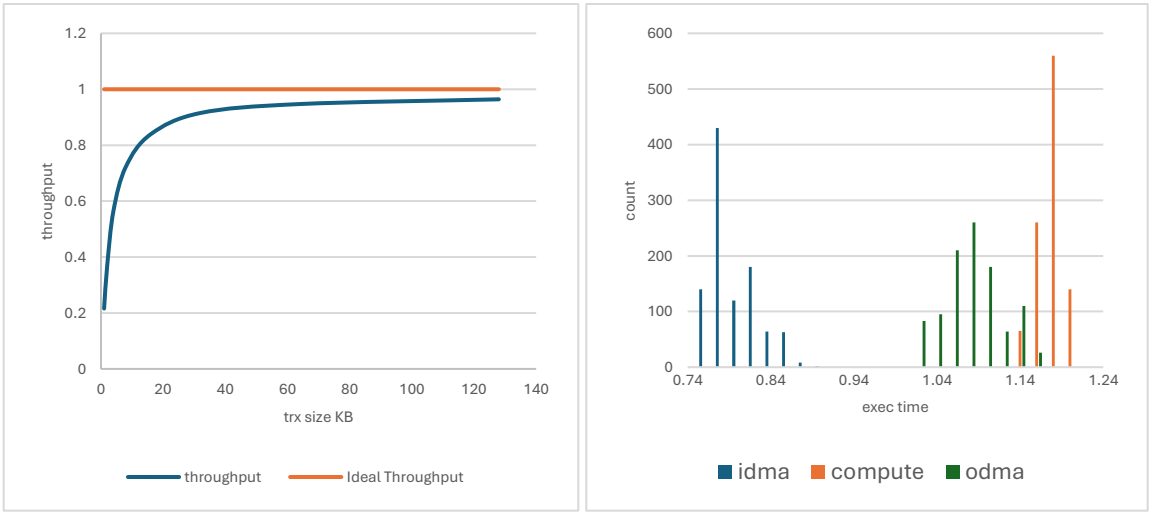

Figure 5: .1 DMA throughput analysis – .2 execution time histogram

V. COST FUNCTIONS CALCULATION

The process to determine the cost functions begins by assessing performance metrics at the module level isolating components such as DMA and the various compute components. Each module is tested independently under different operational conditions and parameter configurations, tests are generated with PSS reusing the same actions applied at system level to determine the individual contribution to the overall cost.

This approach allows for a systematic evaluation of performance variability, contributing to an accurate definition of the cost function based on actual performance measurement of the single component in isolation.

The cost function is defined as the estimated execution time correlated to a particular data transfer or kernel. It could be generally represented by $C=K0+F(p0,p1,\ldots,pn)$, where $K0$ is a constant representing fixed system overhead, and F is a function dependent on multiple parameters, p0,p1,…,pn, specific to each module. Fig. 5.1 illustrates the results of DMA module testing, displaying variations in performance as transfer size changes. These results establish baseline performance for each component and clarify parameter sensitivity.

Following the cost calculation for all components in the application, an optimized scheduling algorithm is developed by incorporating these costs. This optimized schedule undergoes validation through post-execution analysis to verify alignment with the anticipated cost function values. Post-analysis includes generating histograms for each component to provide insights into their distributional characteristics, as well as identifying any interactions or conflicts that arise during simultaneous operation within the system. This analysis uncovers the influence of component interactions on the cost function, revealing potential areas for further optimization.

The cost function is then refined by introducing an interaction factor that accounts for system conflicts or synergies, represented either as an additional constant or as a random variable as $C=K0+F(p0,p1,\ldots,pn)+ K1+ rand\_dist(x)$. This interaction factor may be modeled using a uniform or normal distribution based on observed interaction patterns. Incorporating these refined interaction terms enhances the accuracy of the cost function, ensuring it better reflects the real-world behavior of the AI application, thereby providing a reliable framework for effective scheduling optimization across various scenarios.

VI. RESULTS

The performance verification based on the Portable Stimuli Standard (PSS) methodology has demonstrated substantial advantages for optimizing AI applications on both single-core and multi-core systems. The structured approach, beginning with single-core application optimization, established a robust foundation that was later extended to multi-core scenarios.

A key advantage of this methodology is its portability, with cost functions determined at the module level. The module approach enabled us to explore a wide number of scenarios and parameters exploiting the quick turnaround and simulation time. The next step is applying those learning to single-core and multi-core use cases, enhancing flexibility across diverse processing configurations.

PSS randomization random constrain resolution engine is a powerful tool to quickly adapt the use cases to the different configuration e.g.: the size of L1 tiles depends on the L1 memory size, the cost function of the DMA depends on the latency in combination with the tile size.

The performance results for single-core applications were collected from over 146 different tests, each based on a different family of NN layer. The presented methodology was used to iteratively optimize scheduling and tiling size parameters. With modular cost functions, this phase sought to reduce execution times while maximizing resource utilization, systematically tuning each parameter to achieve optimal performance. Measurements taken after each iteration were compared against product-level KPIs to ensure alignment with performance goals. Results showed that over 95% of the NN layers met or exceeded KPI targets, demonstrating the methodology's effectiveness. For the remaining layers, further manual analysis was needed to resolve specific bottlenecks, requiring waveform debugging and adjustments in both software and hardware. Software optimizations included resolving memory bank conflicts in Level 1 (L1) memory, implementing cache warm-ups, and inlining C functions. On the hardware side, improvements in Level 2 (L2) memory access and Network-on-Chip (NoC) configurations were essential to address constraints of certain memory access required for specific kernels.

After establishing a successful single-core optimization framework, the methodology was extended to multi-core use cases. The modular cost function approach proved highly portable, allowing efficient scaling to multi-core architectures. Data-level parallelism was employed, where NN layers, particularly convolutional layers, were split across multiple cores by dividing the feature map into sub-feature maps, enabling concurrent processing across cores. Alternatively, independent NN layers could be distributed to different cores, increasing parallel throughput. Here, the modular cost functions could be refined using a histogram-based analysis to account for the bandwidth reduction for each core and shared resources conflict.

Kernel name	Fmap input	number of cores	(exec_time-KPI)/KPI%
Conv W<128×512×1×1> + Relu	(512, 135,240)	1	2.43%
Conv W<256×256×3×3> + Relu	(256, 68, 120)	1	-4.83%
Conv W<1024×256×1×1> + Add + Relu	(256, 66, 117)	1	-1.01%
Conv W<64×3×7×7> + BatchNormalization + Relu + Maxpool	(3, 1920, 1080)	1	1.49%
Matmul B<2048×1001> + Add + Identity + 2 * (Softmax + Identity)	(2048, 1, 1)	1	-0.19%
Conv W<192×32×1×1> + Clip + Conv W<192×1×3×3> + Clip	(96, 14, 14)	1	0.26%
SpaceToDepth + Conv W<32×128×3×3> + Relu +DepthToSpace	(32, 128, 512)	1	-1.88%
Conv W<64×64×3×3> + Relu	(64, 180, 320)	2	-3.18%
Conv W<128×128×1×1> + Relu	(128, 535, 127)	8	2.12%
Conv W<64×64×3×3> + Relu	(64, 180, 320)	16	0.39%

Table 1: performance results

Tab. 1 reports some of the results obtained through the testing and the relative results compared to the system KPIs for some of the single and multi-care test cases (the name of the kernels is defined using ONNX standard convention). The performance of all the tests is meeting or exceeding the system KPIs defined with a certain margin of error.

The PSS methodology was instrumental not only in verifying performance at the hardware sign-off stage but also in defining an optimization strategy for production software.

VII Conclusion

In this work, the Portable Stimuli Standard (PSS) methodology is demonstrated to be a powerful framework to optimize, analyze and verify performance of AI applications. The portability of PSS enabling efficient and scalable testing from single to multicore systems exploiting the data-level parallelism and workload distribution. This structured approach established a clear and efficient path for performance sign-off, meeting the hardware performance criteria required for production readiness. Additionally, the results were used to improve the optimization strategy for production software, ensuring it was aligned with the hardware's capabilities.

The combined insights from this methodology provided a holistic view of performance optimization, effectively bridging hardware and software requirements guarantying a smooth transition from verification to deployment and a strong foundation for continuous performance improvements and efficient AI application deployment.

Beyond AI, this approach is also well-suited to other computationally intensive domains where data processing takes precedence over control logic, such as radar, image and audio processing.

REFERENCES

[1] Portable Test and Stimulus 1.0a Language Reference Manual Accelerate Systems Initiative USA June 2018.
[3] https://accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf
[4] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", *CVPR 2016*.
[5] https://onnx.ai/onnx/intro
[6] https://onnx.ai/onnx/operators
[7] Challenges & Solutions for AI SoC Semiconductor Testing - National Instruments. 2024-08-14