Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs

Tsung-Wei Huang
Department of Electrical and Computer Engineering, University of Utah
tsung-wei.huang@utah.edu

***Abstract*—Recently, CPU-GPU heterogeneous parallelism has brought transformational performance milestones to static timing analysis (STA) algorithms. As the computing ecosystem continues to proliferate, *performance portability* has emerged as a new challenge when deploying the result to diverse heterogeneous computing platforms. Specifically, the optimal code written on a CPU-GPU architecture may not be optimal for other CPU-GPU architectures, due to various performance, interoperability, and availability constraints. As a result, we introduce in this paper a learning-based framework to enhance the performance portability of a GPU-accelerated STA program. We parameterize important performance parameters and leverage a neural network model to adapt performance optimization to any given computing platforms. We have demonstrated the effectiveness of our framework in real STA applications.**

I. Introduction

The recent research has demonstrated significant success in accelerating static timing analysis (STA) algorithms using CPU-GPU heterogeneous parallelism. For instance, [1] has leveraged GPU to accelerate time-consuming path-based analysis (PBA) up to 45× over 40 CPUs. As new applications continue to harness the power of GPUs, chip vendors have not stopped investing new GPU products (e.g., Nvidia RTX, AMD RX, Intel Arc). While this investment offers researchers various choices to program heterogeneous computing applications, *performance portability* has emerged as a new challenge to overcome. Specifically, *no* single heterogeneous programming model is optimal for all. The same program optimized for a CPU-GPU architecture may not be optimal for another. To enhance the performance portability, there is a need for non-traditional user-level features to adapt performance to various computing platforms.

Consider a GPU-accelerated STA program in Figure 1. The program is written in SYCL [2], a new single-source C++ heterogeneous programming model, to deploy to two different CPU-GPU computing platforms without changing the source. We illustrate the performance in a gray scale (the brighter the better) on a two-dimensional (2D) grid of GPU kernel configurations, threads per block and iterations per thread (number of processed elements per thread). We observe that the optimal configuration on the Nvidia GPU leads to suboptimal result on the AMD GPU, and the variation can be up to 24% (170s vs 211s). While it is possible to count on expert programmers or grid search algorithms to tune kernel configurations, such process is extremely iterative and time-consuming. Moreover, this process cannot handle dynamic performance constraints, such as loads and memory, that depend on the program runtime.

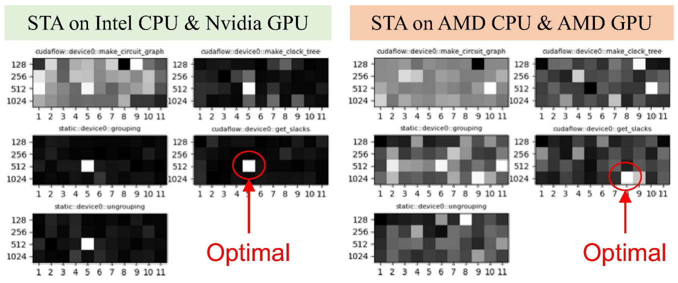

Fig. 1: Performance results of running the same GPU-accelerated STA program on two different CPU-GPU computing platforms.

As a result, we introduce in this paper a framework to enhance the performance portability of a GPU-accelerated STA engine, OpenTimer [3]. Our contribution is threefold: (1) We have identified the need for *parameterizable* kernel designs to achieve performance portability for heterogeneous STA algorithms. (2) We have introduced a deep neural network (DNN) model to learn to optimize the STA program performance in a user's unique computing platform. (3) We have demonstrated the effectiveness of our framework by bridging the performance gap of running a GPU-accelerated PBA algorithm [1] between different heterogeneous computing platforms. While this paper targets an STA workload, the proposed research is framework-neutral and can inspire other heterogeneous design automation applications.

II. Framework

Due to various performance constraints, achieving performance portability is highly parameterizable. This challenge highlights the need for novel learning-based methods to achieve adaptive performance optimization. For the given PBA algorithm [1], our framework learns a DNN model to recommend an optimized scheduling plan for a PBA problem to run on a heterogeneous computing platform. We categorize the feature space into *static features* and *dynamic features*. Static features connect performance with problem-specific parameters, including the circuit graph size, the fanin cone size of a flip-flop (FF), and the number of critical paths to report. Dynamic features connect performance to runtime statistics, including available GPU memory and CPU load. Our scheduling plan consists of three important label categories to infer, kernel execution parameters (block size, iterations per thread), targeted device for each kernel (remain on GPU or switch to CPU), and host execution parameters (number of CPU threads).

Our DNN is trained on a user-specific computing platform. Feature and label selections are based on extensive experiments we have conducted on different computing platforms. To maximize the flexibility of learning, we have decided to implement the PBA algorithm using SYCL [2], which enables a unified programming and execution environment for heterogeneous computing. Specifically, a written SYCL kernel can run on any OpenCL device, including CPU, GPU (Nvidia, AMD), and other accelerators (e.g., Intel oneAPI products). This property allows our model to toggle a GPU kernel to run on a CPU, thereby reducing the GPU traffic in a contending server environment and vice versa. As we integrate our framework into OpenTimer [3], [4], an input PBA problem is represented as a heterogeneous task graph [5], [6]. We leverage the graph embedding techniques of Google's GAP [7] to embed features in nodes as the input for our DNN. Figure 2 shows the proposed framework.

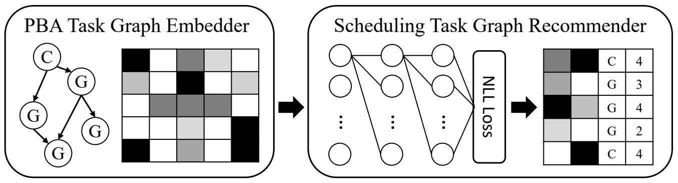

Fig. 2: DNN-based framework to recommend a scheduling plan (e.g., kernel block size, GPU tasks (G) to run on CPU (C), worker count) for an input PBA task graph on a computing platform.

III. Experimental Results

We train our DNN using PyTorch and integrate the result into OpenTimer's PBA module [3]. Our DNN adapts the scheduling performance of a PBA problem to the computing platform from which the DNN is trained. We generated up to 100K different PBA problem instances by augmenting the circuit graphs in TAU Contests [3] to different sizes and configuring each PBA algorithm with different input parameters; 90% for training and 10% for testing. We collect runtime data of these PBA problems on two different computing platforms: (1) 4 Intel Xeon CPU cores with an Nvidia RTX 3090 GPU Ti, and (2) 8 AMD Ryzen CPU cores with an AMD RX 6900 GPU XT. Both machines have 128 GB RAM. Our neural network has 4 fully connected layer; the first layer takes 24 node features and the last layer outputs a per-node scheduling plan. We train our DNN on the AMD machine, which took about 18 hours to finish.

Figure 3 shows the performance results of seven PBA problem instances. The black bar shows the runtime of each PBA problem on the Intel-Nvidia computing platform, for which the program was tuned optimally. On the AMD computing platform, the red bar and the gray bar show the runtimes of the same program with and without our DNN performance adaptor. We can see that the optimal scheduling plan for the Intel-Nvidia platform is not optimal for AMD. For instance, on the circuit netcard, running the same program directly on the AMD machine results in 16.3% performance degradation (61.2s vs 71.2s). With our DNN adaptor, we can close the gap to 3.3% (61.2s vs 63.2s).

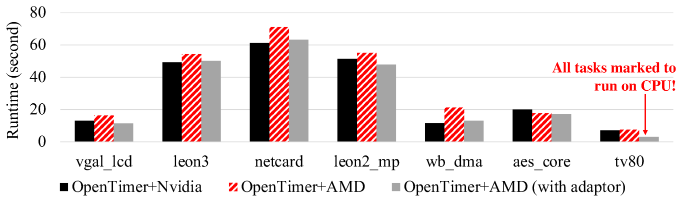

Fig. 3: Performance results of running a GPU-accelerated PBA program on two different CPU-GPU computing platforms.

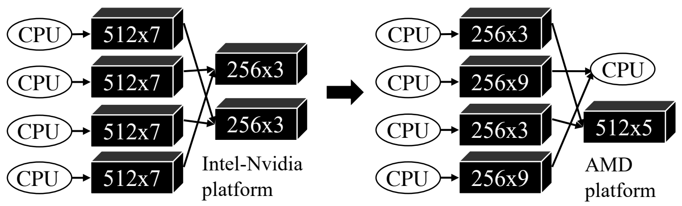

Fig. 4: Inferred scheduling plan for wb_dma on the AMD machine.

Another important finding of our adaptor is the significantly improved performance of tv80 on the AMD machine. Our adaptor recommends running all tasks on CPUs (i.e., placing SYCL kernels on host devices) as the problem size of tv80 is too small to benefit GPU computing. However, such a threshold is difficult to find using general-purpose heuristics as it depends on many complex parameters with non-linear interaction with the computing environment. Figure 4 shows a partial result of the inferred scheduling plan for circuit wb_dma. Our adaptor recommends a different set of kernel execution parameters (block size $\times$ iterations per thread) and a toggled kernel for the AMD machine.

Acknowledgment

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523 (CAREER), and OAC-2209957.

References

[1] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *ACM/IEEE DAC*, 2021.
[2] "SYCL." [Online]. Available: https://www.khronos.org/sycl/
[3] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, no. 4, pp. 776–789, 2021.
[4] T. Huang and M. Wong, "Opentimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.
[5] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," vol. 33, no. 6, pp. 1303 – 1320, 2022.
[6] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE IPDPS*, 2019, pp. 974–983.
[7] A. Nazi, W. Hang, A. Goldie, S. Ravi, and A. Mirhoseini, "GAP: Generalizable Approximate Graph Partitioning Framework," *arXiv*, vol. 1903.00614, 2019.