Abstract—With the rate at which hardware evolves, predicting application performance on new architectures is extremely valuable. Prior work building frameworks to predict application performance on new architectures has required expert input, considered only a subset of applications, or required source code changes. Our goal is to create a general framework that can automatically predict application performance on a target hardware, given performance metrics from a different hardware architecture, without expert input. In this thesis, we propose such a framework and use it to compare classical machine learning approaches to advanced Deep Neural Networks generated through Neural Architecture Search (NAS). We implement a NAS workflow on the Theta supercomputer at Argonne National Laboratory, and use it to create over 1 million deep learning models that predict performance on a target hardware architecture. When comparing these final models, we see little difference when employing a massively scaled neural architecture search compared to a random forest model. The results suggest that general application cross architecture prediction requires significantly more training data than used in this study data and/or additional feature engineering.

I. INTRODUCTION AND MOTIVATION

In recent years, there has been significant investment in the development of new and innovative computing architectures that diverge from the general-purpose CPU. These new architectures include, among many others, graphical processing units (GPUs), neuromorphic computing chips, and field-programmable gate arrays (FPGAs). This investment has been motivated in part by a growing interest in the application of deep learning methodologies within both industry and scientific research [1]–[7]. The future success of these applications is strongly dependent on how efficiently they will run on future hardware. Understanding how to realize these gains can be a laborious process, taking up to seven years, delaying the time to solution. Direct simulations of computing architectures can involve a clear trade-off between accuracy and performance, with cycle-accurate simulations being prohibitive for most real-world applications [8]–[11]. Mukherjee et al. [8] observe that complex designs can take as long as seven years for performance modeling. They highlight how the increased complexity of target architectures and applications makes performance modeling a daunting task. Consequently, porting an application to new hardware often means optimizing for new hardware. Often times, these applications can have code sizes that exceed 100,000 lines, with optimization taking years of effort [12]. On top of this, the development time needed to understand, benchmark, and model each new architecture can be extensive. This is especially problematic today, as the success of machine learning has sparked investment in a large number of new GPU-like architectures for compute-intensive neural-network training [1]–[3].

Since the ultimate performance of a software application depends so strongly on hardware-specific optimizations, especially for specialized architectures like GPUs, a large body of work (Section III) has already aimed to build reliable performance prediction frameworks. The most obvious motivation for such a framework is in the design of the new hardware itself. In the end, the development of a framework that allows for no code changes will contribute to the understanding of performance of applications on these quickly developing and advancing hardware architectures. Many researchers [13]–[26] have explored application performance prediction. Prediction of execution time is a common metric that is studied [13]–[15]. Often, only a subset of applications are considered. Ipek et al. [19] and Carrington et al. [20] each looked at a single large application, while Konstantinidis and Cotronis [22] and others studied simpler kernels such as DAXPY, DGEMM, FFT, and stencil kernels. There is also work on application prediction within the same architecture [19], [27]. Prior work [20], [23], [28] often requires significant expert input, such as awareness and the ability to pinpoint and extract the most compute-intensive kernels in the application [13], [17], [29] or reliance on a lab generated profiling tool. Additionally, application performance prediction has been used for the benefit of improving efficiency in power consumption, grid job scheduling, and resource management [26], [30]–[32].
quire expert input, a complicated profiling tool, knowledge of specific application kernels, or any source code changes. This framework allows for a comparison of six different methods for doing cross architecture prediction. We implement random forest, deep learning, and a neural architecture search model for application performance prediction. Neural architecture search leverages leadership-scale supercomputing resources to build and train empirical models, with limited profiling data. With the right framework, we were able to explore complicated learning models. Prior work has identified specific performance metrics, such as DRAM utilization, and used these as input features for their models. Additionally, prior work would identify specific compute or memory intensive kernels, which would narrow down the scope of the application being tested. With the insights of prior work, we incorporate the use of a well-documented profiling tool, NVProf, which returns between 116 and 120 performance metrics, depending on the architecture tested.

Although not architecture-agnostic, these metrics give a thorough characterization of the application, similar to specific profiling tools used in other research labs. Additionally, this profiling tool breaks down each application into kernels that make up the majority of the application, allowing for further dissection of application performance. In combination with results given by the profiling tool, we identified key metrics held across both P100 and V100 architectures, creating a database that encompasses unique identifiers for each application and corresponding architecture. The final framework uses either active learning or random selection to create a training set which is then used in the specified learning model. This enables a fair comparison of all models used. The framework does not require the use of expert application input or any code changes.

While prior work shows that machine learning can be used for performance prediction under certain circumstances, such as within a cluster of similar applications, we discovered that deep learning does not perform well when considering prediction across architectures with dramatically different kernels in the training set. Performing an extensive exploration across a million neural architectures yielded results no better than a random forest model. Creating a real world dataset, where data is heavily varied and unevenly weighted posed too much of a challenge, right now. The results still hold even when tripling the amount of data used in the training set, as the accuracy only has a mild improvement (< 1%). These results imply that further research into cluster identification among the applications may achieve better results, as well as curating a much larger dataset.

Our contribution in this thesis is as follows:

- A framework that allows performance prediction, of a target architecture, given performance on a different architecture
- A method for incorporating active learning into DNN design
- Comparing random selection and active learning queried training sets
- A comparison of three modeling methods for the problem of predicting a target architecture’s performance given performance data on a different architecture

II. RELATED WORK

With such strong motivation behind performance prediction, there is much prior work. From single intra-architecture performance prediction [33]–[37] and inter-architecture predictions [13], [15], [16], [20]. Ardalani et al. [13] want to understand the benefit of estimating GPU performance prior to writing a specific GPU implementation of their current CPU implementation. Here, the authors look at the potential benefits of a GPU implementation by looking at corresponding counterparts of the current CPU implementation. They note that since CPU programming is much easier than GPU programming, programmers can implement different algorithms for the CPU and use the CPU-based GPU performance prediction to get speed-up estimations for the different developed algorithms. This tool will then be able to help the developers into choosing and porting the correct algorithm. Specifically, they were able to use program properties, features inherent to the program or algorithm and independent of what hardware the program runs, to create a mapping between these features to GPU execution time. The tool built predicts GPU execution time for CPU code prior to developing the GPU code. Our work looks at a specific metric, IPC, and whether the application in question will become memory bound to obtain an understanding of whether the application is worth porting over to a new architecture. Their final dataset consists of 122 datapoints which was used to test and train 100 different ensemble models achieving an average relative error of 22.4%.

Similar to Ardalani et al., Boyer et al. [14] created a modeling framework, built on top of GROPHECY, that not only predicts kernel execution time, but data transfer time to represent the total execution time of a GPU application. The extended a GPU performance model to include a data usage analyzer for a sequence of GPU kernels, to determine the amount of data that needs to be transferred, and a performance model of the PCIe bus, to determine how long the data transfer will take.

Yang et al. [15] looks at relative performance between two platforms while only needing to observe short partial executions of two ASCI Purple applications. Their method targets performance predictions in guidance for resource allocation. The partial executions require an API where the user must understand where the repetitive phases occur to understand execution behavior across the entire application without the need for full execution. The predictions and evaluations were done across CPUs only and partial execution on the target is required in order to extrapolate and predict whole application performance. Unlike this approach, our work does not require the user to understand the specific partial executions needed to run to implement the workflow.

Marin et al. [16] created a toolkit that semi-automatically measures and models static and dynamic characteristics of applications using the application binaries to predict the L1,
L2, TLB cache miss counts, and execution time. They describe a methodology as a function of how the application exercises the memory subsystem, for constructing models of an application’s characteristics parameterized by problem size or other input parameters. Though Marin et al. created an architecture-neutral model, our work doesn’t require the developer to work on application binaries nor go through a complex workflow to create an initial characterization of the application.

Meng et al. [17] created a GPU performance projection framework, used by Boyer et al., that estimates the performance benefit of using a GPU without requiring GPU programming, but by providing pieces of CPU code that targets for GPU acceleration. The authors defined CPU code skeletons, automated a mechanism to restructure CPU code skeleton and mimic transformations need to tune GPU code, characterized the benefits and side effects of GPU code transformations, projected a CPU kernel’s performance on GPUs without producing GPU code. The authors also allowed the ability to explore future GPU generations and evaluate their performance by varying GPU hardware specifications. The developed code skeletons are transformed to mimic tuned GPU codes where the cost and benefit of GPU development can then be estimated according to this transformed skeleton. Our workflow would not require converting the application into skeleton code and instead would require the user to profile the application on the current architecture using NVIDIA’s NVprof profiling tool to gain the applications characterization.

Hong et al. [18] created a power and performance prediction model (IPP) that predicts the optimal number of active processors for a given application. IPP, takes a GPU kernel as input and predicts both power consumption and performance together. Using these power consumption and performance outcomes, IPP predicts the optimal number of cores that result in the highest performance per watt. Their results show that by using fewer cores based on the IPP prediction, they would be able to save up to 22.09% of runtime energy consumption for the five memory bandwidth-limited benchmarks. In particular, this work characterizes performance prediction in terms of power modeling for the GPU.

Ipek et al. [20] created an easy to use model using one parallel application, SMG2000, to predict performance across two platforms. Similar to our work, they employed a multilayer neural network trained on input data from executions on the target platform capturing full system complexity achieving 5%-7% error across a large, multi dimensional parameter space. In our work, we expand the search beyond one application, to multiple applications creating a complex training set that is not application specific.

Carrington et al. [28] furthered development of their framework to include blind predictions for three systems as well as establishing sensitivity studies to advance understanding of observed and anticipated performance of both architecture and application. Here, the defined that the Machine Profile is measurements of the rates at which a machine can perform basic operations, including message passing, memory loads and stores, and floating-point operations which is collected via low level benchmarks and probes. They specifically look at performance on two applications, Cobalt60 and HYCOM.

Lee et al. [21] looked deeply into parameter space characterization for highly parameterized parallel applications. They construct and compare two classes of effective predictive models: piecewise polynomial regression and artificial neural networks. They look at performance prediction of Semicoarsening Multigrid (SMG2000) and High-Performance Linpack (HPL). Here, a single neural network is developed and was tested using only 100 validation points. We tested over a million neural networks which were each tested with validation sets that had between 400 to thousands of validation points. As noted in the paper, they observed a non-monotonic trend when adding data to their training set illustrating the difficulty of identifying an optimal sample size, which is something we encountered as well.

Konstantinidis et al. proposed a performance prediction of GPU kernels on 4 different computation kernels: DAXPY, DGEMM, FFT and stencil kernels achieving an absolute error in predictions of 10.1% in the average case and 25.8% in the worse case [22]. To achieve realistic results the authors applied three adjustments to the theoretical peak device performance. The adjustments are on the compute and memory transfer peaks and the compute peak of a particular kernel.

Balaprakash et al. [23] present an iterative parallel algorithm that builds surrogate performance models for scientific kernels and workloads on single-core, multicore, and multinode architectures. They developed ab-dynaTree, a dynamic tree model obtained through updates of the first in previous iterations which is then used to choose the next inputs for training. In [29], Balaprakash et al. use their previously developed active learning model, ab-dynaTree to obtain surrogate models for GPU kernels when concurrent evaluations are not possible due to the lack of availability of a GPU cluster. Here, they present an active learning approach for obtaining surrogate models for GPU kernels and an empirical study spanning a diverse set of GPU kernels and graphic cards. In line with our work, Balaprakash built these surrogate models to minimize execution signatures (MetaSim Tracer). They rely on a convolution method that maps this application signature onto a machine profile. Their MetaSim Tracer processes all of the memory addresses of an application on-the-fly. Similar to the Carrington et al. work, the returned NVProf performance metrics used in this work, characterizes the application’s memory and bandwidth utilization on the architecture and is what we use as the input features for our models. Carrington et al. is also predicting execution time and not IPC or whether the application becomes compute bound or not.

Carrington et al. [28] furthered development of their framework to include blind predictions for three systems as well as establishing sensitivity studies to advance understanding of observed and anticipated performance of both architecture and application. Here, the defined that the Machine Profile is measurements of the rates at which a machine can perform basic operations, including message passing, memory loads and stores, and floating-point operations which is collected via low level benchmarks and probes. They specifically look at performance on two applications, Cobalt60 and HYCOM.

Lee et al. [21] looked deeply into parameter space characterization for highly parameterized parallel applications. They construct and compare two classes of effective predictive models: piecewise polynomial regression and artificial neural networks. They look at performance prediction of Semicoarsening Multigrid (SMG2000) and High-Performance Linpack (HPL). Here, a single neural network is developed and was tested using only 100 validation points. We tested over a million neural networks which were each tested with validation sets that had between 400 to thousands of validation points. As noted in the paper, they observed a non-monotonic trend when adding data to their training set illustrating the difficulty of identifying an optimal sample size, which is something we encountered as well.

Konstantinidis et al. proposed a performance prediction of GPU kernels on 4 different computation kernels: DAXPY, DGEMM, FFT and stencil kernels achieving an absolute error in predictions of 10.1% in the average case and 25.8% in the worse case [22]. To achieve realistic results the authors applied three adjustments to the theoretical peak device performance. The adjustments are on the compute and memory transfer peaks and the compute peak of a particular kernel.

Balaprakash et al. [23] present an iterative parallel algorithm that builds surrogate performance models for scientific kernels and workloads on single-core, multicore, and multinode architectures. They developed ab-dynaTree, a dynamic tree model obtained through updates of the first in previous iterations which is then used to choose the next inputs for training. In [29], Balaprakash et al. use their previously developed active learning model, ab-dynaTree to obtain surrogate models for GPU kernels when concurrent evaluations are not possible due to the lack of availability of a GPU cluster. Here, they present an active learning approach for obtaining surrogate models for GPU kernels and an empirical study spanning a diverse set of GPU kernels and graphic cards. In line with our work, Balaprakash built these surrogate models to minimize execution signatures (MetaSim Tracer). They rely on a convolution method that maps this application signature onto a machine profile. Their MetaSim Tracer processes all of the memory addresses of an application on-the-fly. Similar to the Carrington et al. work, the returned NVProf performance metrics used in this work, characterizes the application’s memory and bandwidth utilization on the architecture and is what we use as the input features for our models. Carrington et al. is also predicting execution time and not IPC or whether the application becomes compute bound or not.
times of benchmark problems. We also used an active learning heuristic to reduce and specify the training set required to train the overall models.

From predicting execution time, specific metric predictions, and CPU to GPU mapping, prior work has made many advances in application performance prediction. That said, the work mentioned often requires expert input, lab specialized profiling tools, and/or pinpointing specific compute heavy kernels in the application. Additionally, prior results achieving good performance, error less than 5%, look at only 12 similar applications. Finding an automatically created generalized model without expert input can have significant benefits. A generalized model has an input of any application, from simple computations to machine learning models, and predict performance on a target hardware.

III. BACKGROUND

In this section we provide background information on the following aspects of the work that we present in subsequent sections: the two GPU architectures considered in this work - Nvidia P100 and V100, the main metric we are predicting - IPC, and modeling methods used - random forest, deep learning, neural architecture search, and active learning.

IV. GPU ARCHITECTURES: NVIDIA P100 AND V100

The two architectures we use in this work are NVIDIA’s P100 and V100 provided by Argonne’s Leadership computing facility. In all cases presented, we use P100 as the architecture we are predicting from, and the V100 as the architecture we are predicting to. The P100 metrics are used as features in the training set. The V100 IPC are used as the target value in the case of IPC prediction. For memory bound prediction, a labeling is created for each datapoint regarding whether they are considered memory bound on the corresponding architecture.

Though at face value, NVIDIA’s P100 and V100 may not look that different, especially compared to older models (going from K40 to V100 has a 200% increase in computation capability whereas going from P100 to V100 only 16%), we can see substantial differences when comparing performance across these two architectures. Table I shows the architectural details of present and past NVIDIA GPUs. The last two columns, P100 and V100, are similar although showing an increase in the number of streaming multiprocessors (SMs) and memory cache for the V100. However, Figure 1 illustrates that there is no function that can map a P100 IPC value to a V100 IPC value, because one P100 IPC value can map to multiple V100 IPC values. Therefore, we are not looking a 1-1 mapping between one metric to another, but have an application characterization of multiple metrics. Though they are only one generation apart, the P100 and V100 have fundamental differences that can be seen when executing and profiling a variety of applications between them. The V100 was developed with a core purpose of creating better performance for AI applications, though it is not always the case, as shown here and by other benchmarks [38].

In this thesis, we use NVIDIA’s NVprof to profile all applications, acquiring around 120 metrics, depending on the architecture, to give a detailed overview of an application’s performance on the GPU architecture. In this work, we look at 116 shared metrics between the two GPU architectures, listed in the Appendix, as the P100 did not have the NVLink metrics the V100 has. Metrics such as dram_read_throughput, dram_write_throughput, single_precision_fu_utilization, and flop_count_dp, along with 112 other metrics are used as features to predict IPC on the given architecture. The total metrics used are reduced to 116 as P100 does not have NVLink performance metrics in the profiling result. Similar to how Carrington et al. created and collected machine profiles, these metrics give an overview of the applications performance. Unlike Carrington et al., these profiles are mapped to the specific architecture the application was run on.

V. ARCHITECTURE PERFORMANCE: IPC

Instruction per cycle (IPC) is often used as a simple metric of performance in the development of CPUs. IPC is a good indicator on whether an architecture is optimally performing. A high IPC is not always indicative of a more efficient architecture, but is a start at recognizing the potential of an architecture with a simple metric. Additionally, because of the complexities and differences between CPU and GPU’s, IPC is an easily available metric that can be traced across these two differing chip architectures. In particular, NVIDIA defines the IPC value as the instruction throughput over a period of collection. As we hope this framework will eventually extend to cross-chip (GPU to CPU) predictions, we chose a metric can be carried from one architecture to another [39].

VI. ACTIVE LEARNING

The key idea behind active learning is a machine learning algorithm that can perform better with fewer labeled training instances if it is allowed to choose the data from which it learns [40]–[42]. In our work, we employ the an active learning system due to the fact that we will have much more unlabeled data (P100 performance metrics), than labeled data (V100
### TABLE I

**Key specifications of selected GPUs of different generations. Computation capability is considered numerical label for the hardware version.**

<table>
<thead>
<tr>
<th>CPU</th>
<th>Kepler (K40)</th>
<th>Maxwell (M60)</th>
<th>Pascal (P100)</th>
<th>Volta (V100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IBM Power8 @2.2GHz</td>
<td>Intel Xeon E5-2670 @2.60 GHz</td>
<td>IBM Power8 @2.2GHz</td>
<td>Intel Xeon E5-2699 @2.0GHz</td>
</tr>
<tr>
<td>Computation capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>16</td>
<td>56</td>
<td>80</td>
</tr>
<tr>
<td>Cores/SM</td>
<td>192 SP cores/64 DP cores</td>
<td>128 cores</td>
<td>64 cores</td>
<td>64 SP cores/32 DP cores</td>
</tr>
<tr>
<td>Texture Units/SM</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Register File Size/SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>L1 Cache/SM</td>
<td>Combined L1+Shared 64K</td>
<td>Combined 24KB</td>
<td>Combined 24 KB</td>
<td>128 KB Unified</td>
</tr>
<tr>
<td>Texture Cache</td>
<td>64KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory/SM</td>
<td>Combined L1+Shared 64K</td>
<td>96 KB</td>
<td>64 KB</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1536 KB</td>
<td>2048 KB</td>
<td>4096 KB</td>
<td>6144KB</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>12 GB</td>
<td>8 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Performance metrics). Additionally, Settles notes that labeled instances are difficult, time-consuming, or expensive to obtain. In our case, there was only one V100 GPU with a long queue time, while collecting performance metrics of certain larger applications taking longer than the allotted time cap. Some of examples of the successful use of active learning are in speech recognition [43], [44], information extraction [45], [46], classification and filtering [47], [48]. All examples show how active learning systems overcame the labeling bottleneck querying in the form of unlabeled instances to be labeled by an oracle. With this querying scenario, active learning aims to achieve high accuracy using as few labeled instances as possible and as a result, minimizing the cost of obtaining labeled data.

Active learning has several scenarios in which may pose queries. In this thesis, we will focus on pool-based active learning cycle. Pool-based scenarios have been studied for many machine learning problems [42], [49]–[51]. In this active learning scenario, queries are selected from a large pool of unlabeled instances. An uncertainty sampling query strategy where the where the instances in the pool that the model is least certain how to label are chosen. The learner can begin with a small number of instances in the labeled training set and request labels for low confident instances learned from the querying results. These requested labels are added to the labeled set, and the learner proceeds from there in a standard supervised way.

There are different querying frameworks when using active learning: uncertainty sampling, query-by-committee, and expected model change to name a few. We use uncertainty sampling as our querying framework. Uncertainty sampling is a commonly used query framework where the active learner queries instances about which it is least certain how to label [52], [53]. The unlabeled instances will be ranked based on uncertainty and the learner queries the top most uncertain instance to have them labeled. Uncertainty sampling in regression, and used in this thesis, is the calculation of the sampling variance of the random forest. Wager, et al. developed this method of estimating the variance of bagged predictors and random forests. This variance estimation tells whether the random forest is more confident about certain predictions compared to others [54].

Though there are many potential benefits to employing an active learner, we should note that there are some caveats with active learning. First, the learner measures based on a single hypothesis. The training set returned is very small and as a result introduces potential sampling bias.

### VII. DeepHyper and Balsam for Neural Architecture Search

Neural architecture search (NAS) is the process of automating architecture design for a machine learning model [55]. We employ this powerful process to search across over a million machine learning models predicting performance prediction. Zoph et al. present a Neural Architecture Search, that uses a recurrent network to generate the model descriptions of neural networks. They train the RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set. Elsken et al. categorize methods for NAS according to three dimensions: search space, search strategy, and performance estimation strategy. Figure 2 shows an illustration of the Neural Architecture Search method often referred to and something employed by the version of NAS in DeepHyper. Specifically, the search space defines which architectures can be represented in principle. This stage can still introduce human bias which can in return prevent finding novel architectural building blocks. The search strategy details the exploration of the search space. The Performance Estimation Strategy refers to the process of estimating the predictive performance on resulting NAS architectures on unseen data [56].

DeepHyper and Balsam frameworks developed at Argonne National Laboratory, are vital in conducting a scaled search over millions of deep learning models [57] [58]. DeepHyper is a scalable framework designed to search the hyper-parameter...
space of deep neural-network models. DeepHyper also includes an integrated neural-architecture search (NAS) mechanism, which enables the automated generation and testing of neural-network models. DeepHyper is tightly coupled with Balsam, a workflow manager that uses a persistent job/task database to efficiently utilize leadership-scale distributed supercomputing resources. Balsam dynamically packages tasks into ensemble jobs and manages the end-to-end scheduling life cycle, ensuring fault tolerance along the way. Additionally, Balsam allows for a complex multi-workflow solution with little user configuration.

As validated using a class of representative cancer data [59], DeepHyper-NAS automates the generation of deep-learning models using a reinforcement-learning. To execute an NAS workflow, DeepHyper results are used to dispatch reward-estimation tasks, based on $F^2$, to Balsam. After the architecture search is completed, there are between 15,000 to 30,000 distinct models generated, trained and tested. From this large pool of models, the estimated reward values are used to select the top 50 DNN architectures. The top 50 DNN architectures are then submitted to a post-training sequence, during which each model is thoroughly trained on the full training data.

In this work, we utilize this Deephyper-based NAS workflow to predict performance metrics on NVIDIA GPU architectures. The use of this model-generation pipeline allowed us to test more than one million neural architecture models.

VIII. METHODOLOGY: APPLICATION PERFORMANCE PREDICTION

The following sections will discuss both intra- and inter-architecture performance prediction, as well as the benchmark data used to train and validate the models. For the intra-architecture case, we consider the prediction of application performance (IPC) on a P100 GPU, given profiling metrics from a P100. Our results for intra-architecture prediction confirm that IPC can be accurately predicted, given either complete or partial profiling metrics. For the inter-architecture case, we consider the prediction of application performance (IPC) on a V100 GPU, given profiling metrics from a P100 GPU. We also explore the inter-architecture classification of specific application runs as either memory bound or not memory bound. The memory-bound classification task does not result in a numerical performance prediction, but ideally captures a similar relationship between the alternative architectures. For inter-architecture IPC prediction, we compare various methodologies, including random forest, deep learning, and NAS.

IX. BENCHMARK DATA

Metrics are collected by Nvidia’s NVProf profiling tool. NVProf instruments the CUDA kernels, and collects a variety of useful performance metrics. We acquire 116 NVProf performance metrics for each of the 46,039 application runs (see Appendix for metric list). The target applications include backprop, hybridsort, kmeans, stream, gaussian, and leukocyte; all except stream come from the Rodinia benchmark suite [60].

Backprop, containing two kernels, is a deep learning algorithm used in the training of feedforward neural networks for supervised learning. Hybridsort, containing seven kernels, is a parallel bucket sort that splits the list into enough sublists to be sorted in parallel using mergesort. Kmeans, containing two kernels, is a clustering algorithm that divides an initial cluster of data objects into K sub-clusters. Kmeans represents data by the mean values or centroids of their respective sub-clusters. Iterations of the algorithm compare the data object with its nearest center, based on a distance metric [60]. Srad, containing eight kernels, or Speckle Reducing anisotropic diffusion is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations [61]. Leukocyte, containing three kernels, is an application that detects and tracks rolling white cells [60]. Stream, containing five kernels, is a benchmark designed to measure sustainable memory bandwidth for contiguous, long vector memory accesses [62].

X. INTRA-ARCHITECTURE IPC PREDICTION

As seen in prior work, it is possible to predict performance metrics within the same architecture using a variety of techniques and tools. That being said, we acknowledged that there is no function that can map P100 IPC directly to V100 IPC, as shown in Figure 1, and thus we use the fuller scope of P100 metrics to predict V100 IPC. Here we present a deep learning method that allows for IPC prediction of applications run on the P100 architecture. We consider this step as a proof of concept that a certain scope of metrics can predict IPC.

For our intra-architecture prediction, we use a feed-forward fully-connected deep-learning model that has weights initialized with a normal distribution. The model has 1 input layer and 2 hidden layers, with all layers (excluding the output layer) using ReLU activation functions. We use ReLU because it shows better convergence results compared to using other activation functions. The deep learning model uses the Adam optimizer, along with a mean squared error loss function. The Adam algorithm calculates an exponential moving average of the gradient and the squared gradient [63]. The model is trained for 100 epochs.

Furthermore, to stress the relationship between metrics and IPC, we reduced the input metrics from 112...
to 5. The following definitions for the reduced metrics are provided by CUDA Toolkit Documentation [64]. The reduced metrics include shared utilization, stall other, single precision fu utilization, dram read throughput, and dram write throughput. shared utilization is the shared memory relative to peak memory utilization. stall other is the percentage of stalls occurring due to miscellaneous reasons. single precision fu utilization is the utilization level of the multiprocessor function units that execute single-precision floating-point instructions on a scale of 0 to 10. dram read throughput is the device memory read throughput. dram write throughput is the device memory write throughput.

XI. INTER-ARCHITECTURE MEMORY BOUND CLASSIFICATION

We first look at cross-architecture memory-bound prediction. Similar to the results of comparing IPC between V100 and P100, Figure 3 shows that there is no linear function that can map between V100 dram write, read, and total utilization to P100 dram write, read, and total utilization. In Figure 5, you can also see the differences in dram utilization between the two architectures in both scaled and un-scaled versions. There is no simple linear function that would map all P100 dram values to V100 dram values. In addition to the P100 data points collected for the intra-architecture prediction step, 32,291 V100 data points are collected using the NVProf profiling tool. Only P100 data points that had corresponding V100 data points were used. We explore whether an application was not memory bound on P100, but is on V100. Applications that require processing large amounts of data, such as multiplying large matrices, would likely be memory bound. In Figure 6, memory bound applications on both P100 and V100 NVIDIA architecture are plotted along their IPC value. Here, IPC results are more spread out with a high DRAM total throughput on the V100 compared to the P100.

By NVIDIA architecture standards, an application becomes memory bound on their architecture is an application having a dram utilization of over 75% on the architecture. As shown in Figure 4, there are kernels that become memory bound on the V100 that are not memory bound on the P100. In order to label our dataset, we calculate the ratio of the total DRAM throughput with that of the theoretical memory maximum. Total DRAM throughput is calculated by adding dram read throughput and dram write throughput. We consider all data points with ratios greater than 0.75 to be memory bound. Using a random forest classifier, we were able to predict whether an application run on a P100 would become memory bound on a V100 (given P100 profiling data). We optimize the hyperparameters of the random-forest classifier using a grid search. The classifier is trained on a set of P100 metrics with the applications corresponding to V100 target values. It is then tested on the validation and test sets to confirm that the model is not over-fitted and performs well on untested data.

Fig. 3. Illustration of DRAM read and write throughput and total dram throughput on the V100 and P100. We see that in all three cases, there is no function that can map DRAM read throughput, write throughput, or total utilization from one architecture to another.
Fig. 4. Illustration of memory throughput for Nvidia P100 and V100. Points above the green line are considered memory bound kernels, or having over 75% of memory bandwidth utilization. In comparison, the same applications run on the V100 show kernels becoming memory bound on the V100 that were not memory bound on the P100.

Fig. 5. Dram read and write utilization on both P100 and V100 GPUs

Fig. 6. Memory bound applications vs. IPC of application on both P100 and V100 architectures

XII. INTER-ARCHITECTURE IPC PREDICTION METHODOLOGY

Here, we present the three modeling frameworks tested for IPC prediction. Due to the limited data size available of performance metrics, we employed the use of an active learner to identify if there were application points that best characterized applications performance on the architecture. Essentially, with the use of active learning, we want to avoid collecting more data than necessary. The total data set (32,291 points) is split into a training (22,603) and hold out set (9,688). We used Argonne’s DeepHyper framework to test over a million deep learning models. We used a random forest model and conventionally developed deep learning model as a baseline comparison. All three modeling methodologies are tested with an active learner queried dataset and a training set created by random selection.

XIII. CURATING TRAINING SETS: ACTIVE LEARNING AND RANDOM SELECTION

In real world applications, gathering data can be quite time consuming. This, combined with the fact that new computing architectures are often scarce, means that labeled datasets are likely to be relatively small in practice. We will also not be training to the entire dataset and therefore, using active learning to create refined datasets. We do this for a range of different refined dataset sizes.

First, an initial base set is chosen. Though there are many forms of creating the base set, the original training set batch (250 points) are points randomly chosen from the data set. We use a random forest as the supervised machine learning model in our active learning strategy. We use additions with batch size of 250 data points during each training cycle. 12 different training sets are created by taking sizes of 2.5 to 30 percent in increments of 2.5, from the training dataset. Therefore, the active learning training cycle is run 12 times, once for each size, creating 12 training sets with corresponding sizes.

Fig. 7. Active learning flow chart showing the first batch of 250 points being trained with random forest and the cycle of querying new data points and adding them to the training set. The cycle terminates once the target training set is reached.

We use a pool-based active learning strategy, where the active learner can query from a pool of data points. Figure 7 shows the active learning workflow of creating a queried training set. First, the supervised learning model, we use a random forest model, is initially trained with the base set and predicts on the unlabeled points. The random forest is trained on 250 datapoints with 116 P100 GPU architecture performance metrics as features, to predict IPC (target value) of the given application run on a V100 GPU architecture (inter-architecture IPC prediction). Next, the unlabeled points are then ordered from highest to lowest based on the model’s confidence of the predicted unlabeled points. We use scikit-learn’s implementation of Wager’s uncertainty definition described here [54] to rank the unlabeled data. The batch of points with the highest uncertainty are chosen. These points are considered valuable as the model has low confidence in its accurate prediction of them, and thus these points will be
added to the training set. The model is retrained with the newly formed training set (base set and new points).

One full round of active learning can be seen in Figure 8, where the blue dots are the most uncertain points and the red points are the predicted points on the unlabeled data set. After the learning cycle is completed, the points are once again ranked dependent on their uncertainty score, and a new batch of points are added to the initial training set. After each addition to the training set, the model is retrained and the uncertainty is recalculated until the specified training size and learning cycles are completed. These specified 12 training sets are used in all models, as show in 11.

As a baseline for comparison, equivalent sized training sets are created with randomly chosen data points from the same training set the active learner has access to. Additionally, the random forest model, conventional deep learning model, and DeepHyper are all trained on a the full training dataset, 70 percent of the full dataset (22,603 data points).

The data distribution and percentage breakdown of the application points selected by random and active learner selection can be seen in Figures 9 and 10, respectively. In particular, the random selection process gives a more stratified overlook of the total application data. The active learning selection process, slowly increases the amount of points from other applications, as the training set gets bigger. In particular, the active learning workflow shows a significant focus on the more difficult to predict backprop application. The continuous addition of the backprop data points can be understood by seeing that the backprop application obtains the lowest confidence when predicted. Additionally, the quantity of the backprop data dominates the data set sampling insuring a high amount of backprop application specific points will be ranked high.

XIV. NEURAL ARCHITECTURE SEARCH AT SCALE

Following preliminary success with DeepHyper-based NAS, we further scale the initial search procedure to generate specialized architectures for each of the fine-tuned active-learning datasets. Altogether, we consider 24 training sets, of which half are random and the rest are curated with active learning. The NAS framework utilizes Balsam to test and train a large set of models needed for reinforcement learning based optimization.

For each of the target datasets, we consider the following neural architecture search space. The search space comprises up to 10 architecture cells, with each cell containing a collection of architecture nodes. The nodes can manifest as various neural network features, depending on the actions of the reinforcement-learning agents. For example, the first node in each cell is the *Variable* node, which can be an identity operator (i.e. no layer added), or a dense layer with a range of sizes and activation-functions. In this case, the *Variable* nodes allowed for dense layers between the sizes of 16 and 96 (increment of 16), and activation functions in the set (None, relu, tanh, sigmoid). In order to enable skip connections, each cell also contained an optional connection to each of the

Fig. 8. Results of one round of active learning. Blue points are the points chosen by the active learner and put into the training set. Red points are the predicted points on the unlabeled data set.
Fig. 9. Normalized application percentage breakdown of data points that were chosen at random.

Fig. 10. Normalized application percentage and distribution breakdown of data points created using active learning.

Fig. 11. Illustration of modeling comparison workflow. We start with a pool of data points, create refined datasets with an active learner and have random selection as comparison. These datasets are then used in random forest, deep learning model, and neural architecture search.

Fig. 12. Model throughput results. The graph shows that over 1.4 million models were tested and created in about 1200 hours.

Fig. 13. Illustration of distributed NAS architecture by Balaprakash et al. Balsam runs on a designated node where the launcher is a pilot job that runs on the allocated resources and launches tasks from the database. The multiagent search runs as a single MPI application, and each agent submits model evaluation tasks through the Balsam service API. The launcher continually executes evaluations as they are added by dispatch on idle worker nodes [59].
previous three cells (including the input layer) and an addition operation to combine multiple input layers.

To use the defined search space to select an optimal neural architecture, NAS employs proximal policy optimization (PPO) search strategy. PPO is a policy gradient method for reinforcement learning which requires a reward estimation strategy for the agents. In this work we use $R^2$ for the reward estimation. The overall workflow for the DeepHyper-Balsam NAS is illustrated in Figure 13 for the case of a single multi-agent search. The multi-agent search runs as a single MPI application with each of the agents submitting model evaluation tasks through Balsam. In this work, we leverage multiple multi-agent searches in parallel, allowing for the concurrent execution of multiple MPI applications.

The neural architecture search produces between 15,000 and 30,000 models for each training set. Since a thorough NAS requires large-scale computing resources for even a single target data set, the approach benefited greatly from access to Argonne’s leadership computing facility. As shown in Figure 12, there were over 1.4 million models tested and created in about 1200 hours.

XV. DEEP LEARNING AND RANDOM FOREST

With the use of the same 24 training sets used for the NAS For comparison, we used a classical machine learning approach, random forest for performance prediction. A random forest is an ensemble machine learning method that uses multiple decision trees. Each of the decisions trees are trained on different data sets where sampling is done with replacement [65], [66]. As with the previous models, we used P100 performance metrics as features and the V100 IPC as the target value. The particular random forest tested has 100 $n_{\text{estimators}}$. We use mean squared error as the optimization metric and used the full feature set.

Similar to the intra-node prediction model, we used a sequential fully-connected deep learning model. Deep learning creates models with multiple processing layers to learn representations of data with multiple levels of abstract [2]. Deep learning can discover complicated and intricate structures of large data sets. Our deep learning model uses an Adam optimizer and mean squared error for loss. In neural networks, the activation function transforms the summed weighted input from the node into the activation of the node or output for that input. All layers but the last layer use a $ReLU$ as an activation function as it showed better convergence compared to other activation functions. The results are trained in batches of 250 points and for 1000 epochs. The model created can be seen in Figure 30. The same workflow is used when training with the full training set.

XVI. EXPERIMENTAL RESULTS AND EVALUATION

Here we present the results of intra-architecture IPC prediction, inter-architecture memory bound classification, and inter-architecture IPC prediction. We also present the results of using two types of neural architectures.

XVII. INTRA-NODE ARCHITECTURE IPC PREDICTION

Although the ultimate goal is inter-architecture performance prediction, we use intra-architecture IPC prediction as a preliminary step. That is, we start by confirming that supervised machine-learning methods can be used to predict performance, given profiling data from the same GPU architecture. As shown in Figures 14 and 17, we observe mean absolute percentage errors of 4.11 and 2.96 when trained on 451 and 4,521 data points, respectively. Additionally, we explore the effects of using a reduced feature space (i.e. fewer profiling metrics) to train similar models. As illustrated in Figures 15 and 16, the use of a reduced feature space does not significantly degrade the accuracy of intra-architecture IPC prediction. This overall success implies that, even with a reduced characterization of the application, it is still possible to acquire accurate IPC predictions, thus motivating the more challenging task of inter-architecture prediction.
As shown in the confusion matrix in Table II, both the false-positive and false-negative prediction rates for the inter-architecture memory-bound classifier are below 2%. The goal of the model is to predict if a particular application run will become memory bound on a V100 GPU, given profiling data from a P100 GPU. The task is complicated by the fact that none of the runs are memory bound on the P100 architecture, requiring the model to implicitly capture critical transitions in performance behavior. The results further suggest that there are some applications, such as backprop, that are likely to become memory bound when moved from the P100 to the V100.

<table>
<thead>
<tr>
<th>Memory Bound Classifier Confusion Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual: No</td>
</tr>
<tr>
<td>Actual: No</td>
</tr>
<tr>
<td>Actual: Yes</td>
</tr>
</tbody>
</table>

XIX. INTER-ARCHITECTURE IPC PREDICTION

First, we will look at overall IPC prediction across the three models tested. In Figures 18 and 19, the IPC prediction compared to true IPC across all applications is shown for random selection and active learning selection of 20% of the data, respectively, using DeepHyper. Across these graphs, it is clear that backprop dominates the data set and is a notoriously difficult application to predict compared to the other applications. Additionally, all models have difficulties in predicting leukocyte with good accuracy.

The majority of all training data corresponds to the backprop and stream applications. The results of backprop are further inspected in Figures 26 and 27, showing DeepHyper predictions using an active learning training set and a random selection training set. These results show that the generated models
Fig. 20. Mean square error, common loss metric used in machine learning models, shows a minor decrease for actively queried training sets. According to the MSE score, the DeepHyper returned model using an active learning queried training set shows a 36% decrease in error over a model using random selection.

Fig. 21. Mean absolute percentage error (MAPE) of each framework across applications tested with 20% of the training set used. Each number above the bar is the MAPE for each application and the corresponding model used.

Fig. 22. Mean absolute percentage error (MAPE) of models using full training set in comparison with Old and New IPC. Each number above the bar is the MAPE for each application and the corresponding model used.

Fig. 23. Mean absolute percentage error (MAPE) and error bar of each framework across applications tested with 20% of the training set used. The last set of the columns is the harmonic mean over the applications.

Fig. 24. Mean absolute percentage error (MAPE) and error bar using full training set. The models shown here are the random forest (RFFULL), conventional deep learning model (CFULL), and DeepHyper NAS created model (DHFULL). Active learning results are not shown as the full training set is used. Harmonic mean across applications shows that random forest has better performance compare to both neural network models.

These trends are further supported in the Figure 21, where the mean absolute percentage error (MAPE) is shown for each of the applications. Figure 23 shows the error bar for each application across all models along with a harmonic mean across the applications, as the last set of bars. Among these applications, srad has the highest MAPE error variation across the data. Figure 22 shows the MAPE scores of the models using the full training set and Figure 24 shows the error bar among the same tested applications. Similar to the models trained with 20% of the training data, there is notable variation in the srad application and little to no variation in stream.

For the simplest baseline, we look at mapping the current architecture IPC value to the target IPC value : ‘OldNew’. In other words, we assume that if the IPC is x on P100, then it have trouble adjusting to scenarios when going from high-IPC values on P100 and to low-IPC values on V100, which is the case for a large portion of the backprop data points.
will also be x on V100. Certain applications, such as stream, would not do well with this mapping, but applications such as backprop would not fair any worse. The ‘Random Forest’ bar corresponds to the prediction of V100 IPC using P100 metrics as features. The random forest performance is on par with the deep learning model performance or obtains lower error, such as the error on hybridsort. The ‘Random Forest + AL’ bar corresponds to a random forest model that uses a training set created by the active learner discussed above. This particular model does not do well in applications such as kmeans and hybridsort, in comparison to random forest without a curated training set by an active learner. This could be due to the fact that the active learning selection has a concentrated focus on the backprop and stream application data in comparison to these other applications, and thus does not focus on acquiring points in the training set for these applications.

The ‘Conv_DL’ bar corresponds to a conventionally developed deep learning model. The simple, sequential structure
of this neural network architecture is shown in Figure 30. We explored various network sizes, activation functions, and regularization approaches. We discovered that deeper models tended to overfit the data, while wider models achieved better results. The ‘DH’ bar corresponds to the NAS-generated model using a randomly sampled data set. The ‘DL + AL’ bar corresponds to a NAS-generated model using an actively learned training set. Figure 31 shows the diagram of a neural architecture chosen by NAS. This is the architecture returned by DeepHyper using a training data set created by an active learner, showing skip connections and a variation of activation functions chosen. Every model created by DeepHyper is as complex if not more complex than the one shown in Figure 31. Overall, when looking at Figure 25, which shows the Harmonic mean across applications for both models trained with the full training set and the partial training set, random forest outperforms all other models. Furthermore, tripling the data used to train the random forest shows very little improvement in error, compared to the conventional deep learning model.
Finally, considering all models use mean square error (MSE) as the loss metric, it is only fair to look at the results of mean squared error with respect to training data size and models used. Figure 20 shows a small decrease in MSE for models using an active learning queried training set. There is about a 36% decrease in error when comparing a DeepHyper returned model with an active learn queried dataset to a randomly selected dataset. That being said, though there is a decrease in this error metric, when looking at the MAPE, it is not significant enough to warrant success for any of these models.

XX. DISCUSSION

The initial results of the intra-architecture IPC prediction, using limited data, suggest that the available nvprof-based features are sufficient for accurate intra-architecture performance prediction. In-line with those results, the memory bound cross-architecture prediction, with accuracy of 99%, can be particularly useful when identifying applications that become memory bound from one GPU architecture to another. This information can be useful to application developers and end users. Advanced knowledge of a memory-bound transition allows developers to focus on different performance optimizations and users to avoid the use of certain chip architectures altogether.

Overall the final DeepHyper model prediction does better than some models but still needs improvement. Given that a majority of the data corresponds to the backprop and stream application, shown in Figure 10, we can focus on these applications for our overall evaluation. Unfortunately, as shown in Figure 22, both random forest and NAS-optimized deep learning models fail to outperform a simple old to new mapping. The overall poor accuracy of IPC for backprop data suggests that this particular application is very difficult to predict, and that domain specific feature engineering is most likely required to further improve the model.

In contrast to backprop, all models were successful at predicting performance of the stream application. While a simple old to new mapping results in a 33 percent error, both random forest and deep learning models reduced this error to 3 percent when using the full training set.

When comparing DeepHyper created models, there are some cases where the random selection does significantly better than the active learning selection. There are also cases where active learning out performs random selection. If we say a MAPE of under 5% is excellent, only one application prediction falls into that category - stream. These results show that active learning did not always give beneficial improvements to the model and at times reduced the accuracy tremendously. These results also show that even after training and testing over a million models, a more complicated heuristic and further feature engineering is needed to identify the relationship between these architectures.

XXI. FUTURE WORK

Further work will examine several modifications to the DeepHyper workflow, including wider and shallower neural
network architectures in the search space. Additionally, incorporating the DeepHyper hyperparameter optimization framework has great potential at optimizing the current models for better performance. We would take the top performing DNN architectures and optimize the hyperparameters. DeepHyper has seen significant improvement in accuracy with this hyperparameter optimization.

The current work looks at predicting only IPC. Further work will look into predicting metrics beyond IPC, such as memory bandwidth and other highly looked-after metrics. Additionally, we have done a preliminary study where placing a threshold of the difference in IPC would dedicate whether or not the application can have a good enough prediction. When using this threshold, our accuracy results dramatically increased. To use this, without requiring a significant amount of data from the future architecture, we will look at creating a classifier that labels a small change over a significant change in IPC.

When optimizing and picking the best neural architectures, the DeepHyper search looks at the optimization metric. The current method uses $R^2$ as it’s optimization metric and MSE as the loss metric, which results in MAPE values that aren’t very good. Further steps will be taken into modifying the optimization metric to better capture the error.

Further steps will also be taking at improving the active learning model to have a better overview of the entire data set versus only the points with the lowest confidence.

**XXII. Conclusions**

Developing a thorough understanding of new chip architectures is often a tedious and time consuming endeavor. In contrast, the development of a data-driven model can be much less time consuming in the case that relevant performance data is readily available. In this work, we have highlighted the current opportunities and limitations of a purely data-driven approach. We have shown that classical machine learning (i.e. random forest) can be used to successfully predict if an application will become memory bound when switching between the P100 and V100 GPU architectures. That is, both deep-learning and random-forest classification was used to predict applications that will become memory bound on lightly loaded throughputs. Although our automatically-generated deep-learning models outperformed other notable performance-prediction techniques, the overall accuracy remains insufficient for practical use.

**XXIII. Appendix**

**XXIV. Nvidia NVProf Metrics**

1. inst_per_warp
2. branch_efficiency
3. warp_execution_efficiency
4. warp_nonpred_execution_efficiency
5. inst_replay_overhead
6. shared_load_transactions_per_request
7. shared_store_transactions_per_request
8. local_load_transactions_per_request
9. local_store_transactions_per_request
10. glt_transactions_per_request
11. gst_transactions_per_request
12. shared_store_transactions
13. shared_load_transactions
14. local_load_transactions
15. local_store_transactions
16. glt_transactions
17. gst_transactions
18. sysmem_read_transactions
19. sysmem_write_transactions
20. l2_read_transactions
21. l2_write_transactions
22. dram_read_transactions
23. dram_write_transactions
24. global_hit_rate
25. local_hit_rate
26. glt_requested_throughput
27. gst_requested_throughput
28. glt_throughput
29. gst_throughput
30. local_memory_overhead
31. tex_cache_hit_rate
32. l2_tex_read_hit_rate
33. l2_tex_write_hit_rate
34. dram_read_throughput
35. dram_write_throughput
36. tex_cache_throughput
37. l2_tex_read_throughput
38. l2_tex_write_throughput
39. l2_read_throughput
40. l2_write_throughput
41. sysmem_read_throughput
42. sysmem_write_throughput
43. local_load_throughput
44. local_store_throughput
45. shared_load_throughput
46. shared_store_throughput
47. gst_efficiency
48. tex_cache_transactions
49. flop_count_dp
50) flop_count_dp_add
51) flop_count_dp_fma
52) flop_count_dp_mul
53) flop_count_sp
54) flop_count_sp_add
55) flop_count_sp_fma
56) flop_count_sp_mul
57) flop_count_sp_special
58) inst_executed
59) inst_issued
60) dram_utilization
61) sysmem_utilization
62) stall_inst_fetch
63) stall_exec_dependency
64) stall_memory_dependency
65) stall_texture
66) stall_sync
67) stall_other
68) stall_constant_memory_dependency
69) stall_pipe_busy
70) shared_efficiency
71) inst_fp_32
72) inst_fp_64
73) inst_integer
74) inst_bit_convert
75) inst_control
76) inst_compute_ld_st
77) inst_misc
78) inst_inter_thread_communication
79) issue_slots
80) cf_issued
81) cf_executed
82) ldst_issued
83) ldst_executed
84) atomic_transactions
85) atomic_transactions_per_request
86) l2_atomic_throughput
87) l2_atomic_transactions
88) l2_tex_read_transactions
89) stall_memory_throttle
90) stall_not_selected
91) l2_tex_write_transactions
92) flop_count_hp
93) flop_count_hp_add
94) flop_count_hp_mul
95) flop_count_hp_fma
96) inst_fp_16
97) ipc
98) issued_ipc
99) issue_slot_utilization
100) sm_efficiency
101) achieved_occupancy
102) eligible_warps_per_cycle
103) shared_utilization
104) l2_utilization
105) tex_utilization

References


