Analyzing Machine Learning Workloads Using a Detailed GPU Simulator

Jonathan Lew*, Deval Shah*, Suchita Pati**, Shaylin Cattell*, Mengchi Zhang†, Amruth Sandhupatla*,
Christopher Ng*, Negar Goli*, Matthew D. Sinclair**, Timothy G. Rogers†, Tor Aamodt*

*Electrical and Computer Engineering, University of British Columbia
{jonathan.lew, shaylin.cattell, cng123}@alumni.ubc.ca
{devalshah, amruth, negargoli93, aamodt}@ece.ubc.ca
**Computer Science, University of Wisconsin-Madison
{spati, sinclair}@cs.wisc.edu
†Electrical and Computer Engineering, Purdue University
{zhan2308, timrogers}@purdue.edu

Abstract—Most deep neural networks deployed today are trained using GPUs via high-level frameworks such as Tensor-Flow [1] and PyTorch [2]. This paper describes changes we made to the GPGPU-Sim simulator [3], [4] to enable it to run PyTorch by running PTX kernels included in NVIDIA's cuDNN [5] library. We use the resulting modified simulator, which we plan to make available publicly with this paper, to study some simple deep learning workloads. With our changes to GPGPU-Sim's functional simulation model we find GPGPU-Sim performance model running a cuDNN enabled implementation of LeNet for MNIST reports results within 30% of real hardware. Using GPGPU-Sim's AerialVision performance analysis tool we observe that cuDNN API calls contain many varying phases and appear to include potentially inefficient microarchitecture behavior such as DRAM partition bank camping, at least when executed on GPGPU-Sim's current performance model.

Index Terms—GPGPU-Sim, Simulator, CNN, CuDNN, GPU, PyTorch

I. INTRODUCTION

Machine learning is being employed to tackle a rapidly growing set of problems. In recent years deep neural networks (DNNs) have made striking advances in accuracy. Training DNNs requires massive amounts of computational power, which is currently predominantly done with graphics processor units (GPUs). While industry has rapidly introduced changes to GPU architectures to support machine learning training, such as Tensor Cores and NVLINK introduced in the NVIDIA Volta architecture [6], academic researchers have largely focused on designing inference accelerators. Although the focus of academic researchers is to exploit the strong potential for neural network deployment in mobile platforms (e.g., iPhone X, Huawei) and small embedded devices [7]–[10], another reason for the lack of academic research on optimizing GPUs for machine learning may be the lack of support in current architecture simulators for running these workloads. This paper takes an important step towards addressing this shortcoming.

Popular machine learning frameworks such as TensorFlow and PyTorch typically expose a high-level python application programming interface (API) to developers. Calls to this API invoke computation on a GPU via specialized precompiled libraries such as cuBLAS [11] and cuDNN [5]. To achieve the highest levels of performance these libraries are typically provided by hardware vendors. These libraries take advantage of the vendor's detailed knowledge of their product's microarchitecture, which is typically not fully described in publicly available documentation. As a result, popular open source GPU architecture simulators such as GPGPU-Sim [3], [4] are unable to run applications that make use of these precompiled libraries. Indeed, we confirmed with the maintainers of GPGPU-Sim that a key limitation of the currently available version of GPGPU-Sim is the lack of support for applications that use precompiled libraries. In this paper, we focus on enabling support for cuDNN as cuDNN enables the highest performance on NVIDIA GPUs via implementation of specialized algorithms such as Winograd [12].

One limitation of this work is a lack of support for NVIDIA's tensor cores which is a consequence of the fact that the intermediate-level PTX assembly code [13] embedded within NVIDIA's cuDNN library does not include tensor core operations. Instead, the cuDNN library appears to contain hand tuned machine-level SASS assembly code for supporting tensor cores. This is a limitation because the current version of GPGPU-Sim only supports executing SASS code for older generation GPUs. We believe that the updated GPGPU-Sim framework we provide is still of significant value as the limited dynamic range of 16-bit floating-point provided in NVIDIA Tensor Cores can result in convergence issues unless special steps are taken [14]. As a consequence, many machine learning researchers still use cuDNN APIs that avoid using Tensor Cores. While NVIDIA's CUTLASS [15] enables use of Tensor Cores, it does not implement highly optimized kernels such as Winograd which can provide large performance gains for convolutional neural networks (CNNs) that have small filter sizes.

Overall, we make the following contributions in this paper:

- We modify GPGPU-Sim to enable running cuDNN. In turn, this enables us to run PyTorch and should enable running other high-level frameworks such as TensorFlow.
- We introduce a new methodology to identify bugs in the functional simulation implementation of GPGPU-Sim.
- As the runtime of architecture simulators is many orders of magnitude slower than hardware and machine learning workloads can run for days, we introduce checkpointing support to GPGPU-Sim.
- Using our modified GPGPU-Sim we analyze one of NVIDIA's cuDNN application samples modeling LeNet [16] trained with the MNIST dataset.

II. BACKGROUND

This section provides background on machine learning frameworks and their implementation as well as GPU simulators.

A. Machine Learning Frameworks

Enthusiasm for employing machine learning in practice followed AlexNet [17] achieving an impressive 15.3% top-5 test error rate on image classification, far out-performing state-of-the-art models at that time. AlexNet was trained for several days on two GPUs, although it was observed that the amount of GPU memory and training time limited the network's size. Follow on work proposed more sophisticated approaches such as VGGNet [18], GoogleNet [19], Residual Networks [20] and DenseNets [21], which have surpassed humans in classification accuracy. This result was achieved by combining huge datasets and GPUs. Prior research has shown that GPU can be $4\times$ to $50\times$ faster than CPUs [22]–[25]. Thus, GPUs play an important role in accelerating the execution times of CNNs [26].

Subsequently, companies have provided more optimized hardware and software for running machine learning workloads on GPUs. For example, NVIDIA has introduced specialized cores known as Tensor Cores, high bandwidth NVLINK for communication between GPUs, and optimized their software stack (e.g., CUDA, cuBLAS, and cuDNN). Matrix multiplication is the key underlying operation behind most of the neural network computations and a highly optimized GPU code to implement traditional matrix multiplication operation has a time complexity of $O(n^3)$ [26]. Faster alternatives include the Winograd and Strassen [27] algorithms. Here, the Strassen's algorithm has a time-complexity of $O(n^{2.81})$ whereas Winograd has a complexity of $O(n^{2.38})$. As Winograd is faster than Strassen's algorithm, it is used in libraries such as cuDNN, which are exploited by frameworks such as Tensorflow and PyTorch.

B. GPU Performance Simulators

1) NVProf: The most closely related tool to GPGPU-Sim is NVProf [28], NVIDIA's command-line profiler for CUDA programs. NVProf and GPGPU-Sim give many similar statistics, including instructions per cycle and the number of instructions executed for certain types of instructions such as loads and stores. They also track basic memory and stall information. NVProf is useful in many cases since it provides fast, accurate results from the hardware itself. Several recent papers have used tools like NVProf to profile machine learning workloads [29]–[31]. However, since these papers use profilers, unlike our work they can only provide higher-level analysis about the behaviors of the applications. In comparison, GPGPU-Sim provides detailed information on memory usage, power, efficiency, can easily be extended to provide additional statistics, and can output useful plots of relevant statistics using AerialVision [32].

2) Simulation: Some prior work has also simulated machine learning workloads, but these papers used private simulators [33]–[36]. Since these simulators are not publicly available and few details are available, it is difficult to compare their approaches to ours. In comparison, we simulate machine learning workloads at high fidelity in the widely used, publicly available GPGPU-Sim. Moreover, the fact that other papers use disparate architectural simulators for machine learning workloads makes it crucial to provide better, publicly available tools for simulating machine learning workloads.

III. IMPLEMENTATION

This section describes the modifications that were required to simulate cuDNN and PyTorch applications in GPGPU-Sim. We use regression tests and NVIDIA's cuDNN MNIST example to verify the functional correctness of our changes. Some of the the key changes we discuss are: (1) adding support for precompiled GPU kernel libraries, (2) implementing some missing CUDA Runtime and CUDA API functions, (3) finding and fixing bugs in the existing functional simulator that prevented correct execution, and (4) adding support for checkpointing simulation. Figure 1 shows the control flow of GPGPU-Sim with our modifications.

A. Support for kernels in external CUDA libraries

The existing version of GPGPU-Sim first extracts all PTX code embedded within an application from the binary using an NVIDIA supplied program called `cuobjdump`. Next, GPGPU-Sim combines the extracted PTX into a single PTX file that is then parsed by GPGPU-Sim's program loader. Unfortunately, this approach causes two issues when trying to run cuDNN enabled applications:

First, we found that cuDNN programs and programming frameworks that use cuDNN are typically dynamically linked to the cuDNN library. However, current versions of `cuobjdump` do not resolve dynamic linked libraries before searching for PTX code. Thus, GPGPU-Sim fails to launch kernels contained in dynamically linked libraries. There are two potential solutions to this: either modify GPGPU-Sim to search through any dynamically linked libraries (using `ldd`), or rebuild the CUDA application and statically link against the external library. In this paper, we followed the latter approach, as shown in Figure 1 (❶).

Second, cuDNN includes code and variables with the same names in multiple source files. After combining all the PTX
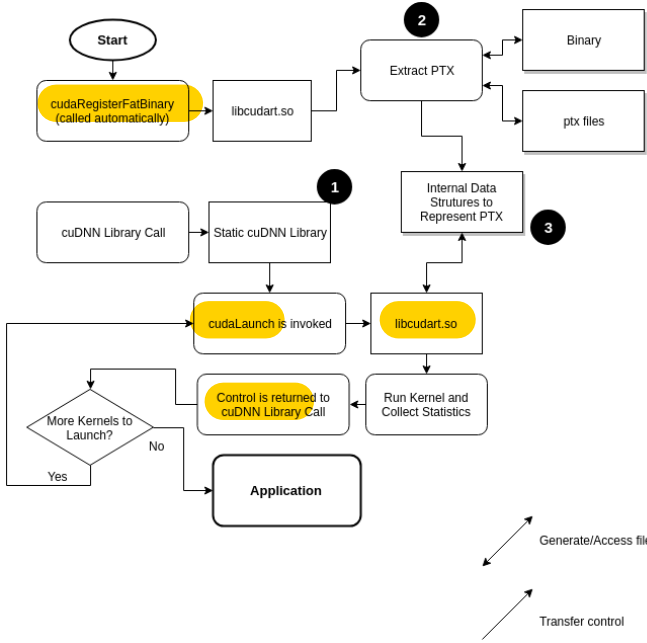
Fig. 1. GPGPU-Sim's Control Flow with modifications for cuDNN.

extracted from the application binary into a single file, these multiple definitions resulted in errors when parsed by GPGPU-Sim's program loader. Thus, we modified GPGPU-Sim to extract and process each embedded PTX file separately (❷).

B. Additional CUDA Language Support

NVIDIA's CUDA enables overlapping memory copies from CPU to GPU with computation on the GPU via a construct known as streams (similar to a command queue in OpenCL). We found that cuDNN uses multiple streams to overlap memory transfers with computation. Although GPGPU-Sim already supports streams, we found it did not support all the required API functions. Thus, we added support for `cudaStreamWaitEvent`, an API call that allows a stream to wait for an event to occur before continuing execution.

Additionally, we also added support for PTX instructions that were not implemented in the current version of GPGPU-Sim, but are used by cuDNN. For example, we found that cuDNN uses the bit reverse instruction (`brev.type d, a;`), which was introduced in PTX version 2.0, for FFT-based convolutional kernels [5], [37]. Thus, we added an implementation for this instruction, which is used to output the bits of its input in reverse order.

Moreover, in the process of developing our debugging tool (discussed further in Section III-D), we found that we had to add an alternative CUDA API call for launching kernels: `cuLaunchKernel`. The CUDA Runtime API equivalent is `cudaLaunch`, which was already supported by GPGPU-Sim.

C. Texture References

To represent textures, GPGPU-Sim uses a system of texture names, texture references (texref), cudaArrays, textureInfos,

and textureReferenceAttrs. A given texture name maps to a texture reference, and a given texture reference maps to a set of cudaArray, textureInfo, and textureReferenceAttr. This aligns with the APIs __cudaRegisterTexture to map a name to a texref; cudaBindTextureToArray to map a texref to a set of cudaArray, textureInfo, and textureReferenceAttr; and unbindTexture to unbind a cudaArray from a texref. A texture instruction in CUDA kernels accesses this data by looking it up with the texture name.

Although textures were already supported in GPGPU-Sim, MNIST registered multiple texrefs to the same name. This caused conflicts in the map and as a result, data was lost. Consequently, some texture instructions would fail because they could not find the cudaArray they were looking for. To resolve this problem, we mapped the texture names to a set of texrefs and also mapped texture names directly to their associated cudaArrays, textureInfos, and textureReferenceAttrs. Thus, texture instructions now use texture name to look up cudaArrays, textureInfos, and textureReferenceAttrs.

We also encountered another problem with textures where the program called bindTextureToArray on the same texref with different cudaArrays multiple times. To resolve this, we assume the program meant to first unbind the existing cudaArray from the texref, and then bind the new cudaArray to the texref.

D. Debugging Functional Simulation Errors

Although the existing GPGPU-Sim simulator correctly runs a large number of CUDA applications such as those found in Rodinia [38], machine learning workloads presented additional complexities. After the changes discussed in Sections III-A–III-C, GPGPU-Sim could run MNIST to completion but generated incorrect results. This presented us with a big challenge: How to find which instruction(s) out of billions of executed instructions was incorrect?

The developers of GPGPU-Sim gave us some help by explaining the process they followed to getting GPGPU-Sim working [4]: first, they validated individual instructions one-by-one by comparing execution on real GPU hardware with execution on GPGPU-Sim. Then, once each instruction appeared to work, they started with getting smaller applications (e.g., `template` from the CUDA SDK) running and progressively ran larger applications. While adding applications, they used GNU's `gcov` tool to compare the coverage analysis of the simulator for correctly simulated applications and new, incorrectly simulated applications. By comparing these, performing "differential coverage analysis", they were able to narrow down which part of the functional simulator had a bug.

Thus, we first attempted to solve the functional correctness issues by employing a similar differential coverage analysis. We compared the coverage of the functional simulator when running the regression tests on GPGPU-Sim's Github page with the results obtained running MNIST. Using this approach, we identified that GPGPU-Sim's implementation of the bit
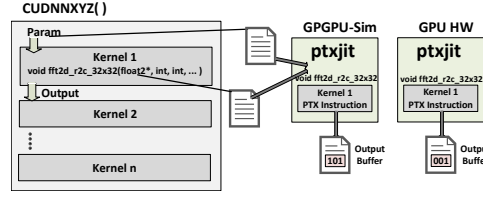
Fig. 2. Identifying the first incorrectly executing kernel within a multi-kernel library function call
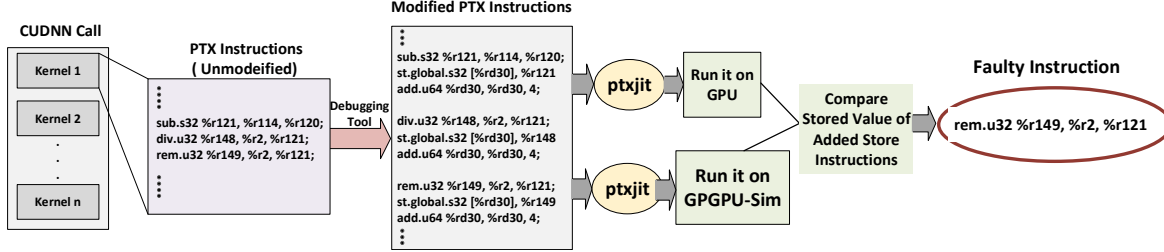

Fig. 3. Identifying the first incorrectly executing instruction within the first incorrectly executing kernel

field extract instruction (`bfe.type d, a, b, c;`) had subtle errors for signed inputs. Thus, we modified GPGPU-Sim's bit field extract instruction implementation to correctly handle signed 32-bit and 64-bit integer inputs. However, after this change, GPGPU-Sim's result for MNIST was still incorrect. We found no other other lines that were exercised by cuDNN and not exercised by the regression tests that appeared to involve incorrectly executing instructions.

We then developed a new approach to debugging functional simulation errors in GPGPU-Sim that was ultimately successful in finding the remaining error. We believe this approach and the resulting debug tool, which we plan to make available with this paper, will be useful to other researchers encountering incorrect results when running new applications in GPGPU-Sim. At a high level, we compare the execution of every instruction executed by GPGPU-Sim to the result obtained from executing that instruction on hardware, then flag the first instruction with an error. An important practical complication we encountered is that every high level API call in cuDNN launches several kernels onto the GPU. Thus, in practice, we followed a three-step process: first identify which cuDNN API call results in incorrect results, then identify which GPU kernel launched within that API call is executing incorrectly, and finally identify the first instruction in that kernel that executed incorrectly.

To identify which cuDNN API call was incorrect, we compare the result buffers on GPGPU-Sim versus the hardware by adding calls to cudaMemcpy to MNIST. However, having identified an incorrect API call, finding which specific kernel of our cuDNN enabled application was responsible for an incorrect result is non-trivial because we do not have source code for cuDNN. Thus, we changed to GPGPU-Sim to optionally capture and save all relevant data to a file. As shown in Figure 2, this data corresponds to the data which is being copied to the GPU before a kernel is launched, along with the parameters passed into the kernel as GPGPU-Sim runs.

Armed with this data, and using our debugging framework, the extracted PTX, and a version of the ptxjit CUDA SDK example, we systematically launch each kernel from the failing cuDNN API call onto both GPGPU-Sim and a real GPU. We assume that any kernel parameter that is a pointer may point to an output buffer. We also modified GPGPU-Sim to obtain the size of any GPU memory buffers pointed to by these pointers. Then, after the extracted kernel executes we use cudaMemcpy to transfer all buffers back to the CPU so they can be output to a log and compared. By comparing the buffers after each kernel, we can identify which kernel executed incorrectly.

Next, as illustrated in Figure 3 we instrumented the extracted PTX for just the incorrectly executing kernel so that the results of each executed instruction that writes a value to a register is saved into a new global array in GPU memory. At the end of the kernel execution, this array is transferred to CPU memory and written to a log file. Comparing GPGPU-Sim execution of this modified kernel versus GPU hardware helps the user to identify the first instruction that executed incorrectly. To help automate the process of adding store instructions to a kernel, we developed an LLVM-based tool to modify a kernel.

Using the above approach we found that the first kernel in `cudnnConvolutionForward` had an error when executing a remainder instruction. Specifically, the remainder instruction "`rem.u32 %r149, %r2, %r121;`" inside the kernel "`fft2d_r2c_32x32`" generated a different result in GPGPU-Sim versus hardware. In the existing GPGPU-Sim this instruction is implemented by the function `rem_impl` using the code:

```
data.u64 = src1_data.u64 % src2_data.u64;
```

In GPGPU-Sim, `data`, `src1_data`, and `src2_data` are C/C++ "union" type called `ptx_reg_t` which, among others, contains fields named "`.u32`" for holding 32-bit unsigned

values and ".u64" for unsigned 64-bit values. Thus, this code is incorrect in some cases because it does not take into account signed vs. unsigned operations and 32- vs. 64-bit values. To resolve the problem, we added a switch statement take account of the type specifier and signed operations, e.g., .u32 and instead use the code such as:

```
data.u32 = src1_data.u32 % src2_data.u32;
```

when type is ".u32" and

```
data.s32 = src1_data.s32 % src2_data.s32;
```

when type is ".s32". After making this change, cuDNN was able to run 32-bit floating-point applications correctly.

1) FP16 Support: Much research on hardware support for deep learning, particularly for inference, focuses on reduced precision. NVIDIA supports 16-bit floating-point on both their regular ALUs and in their Tensor Cores and cuDNN has 16-bit (FP16) versions of the algorithms it supports for deep learning. Accordingly, we added FP16 support in GPGPU-Sim, including instructions that convert FP32 to FP16 and back using an open source library.

However, when we ran MNIST in FP16 mode, it produced incorrect results. We traced this problem back to a subtle issue with multiply instructions, followed by either a subtract or an add, being optimized by the NVIDIA assembler into fused-multiply-add (FMA) SASS instructions. The FMA instruction retains additional precision between multiplication and addition, which results in a mismatch between GPGPU-Sim and execution on GPU hardware. Thus, correctly simulating code with 16-bit floating-point instructions is left to future work. We expect our debugging strategy mentioned above will be of help in this regard but will need to be modified to account for rounding errors.

2) Timing-Model Deadlocks: We also fixed bugs in the memory model and in GPUWattch code that caused cuDNN enabled programs to deadlock GPGPU-Sim's timing model.

E. PyTorch and TensorFlow

After successfully running 32-bit MNIST with correct outputs, we turned our attention to supporting PyTorch and TensorFlow. PyTorch's calls invoke functions in its shared library, _C.so. A regular PyTorch build uses rpath to link this _C.so dynamically to a hard-coded path to the CUDA Runtime Library installed on the machine. We removed all these rpath links, so that it would forced to look for a shared library at run time. Then we changed the search paths in our environment so that it would find GPGPU-Sim's libcudart.so. Finally, we used cuDNN's shared library libcudnn.so to get the source of the corresponding PTX.

When an application imports Torch, the library libcudart.so is loaded, which invokes a series of initialization functions in GPGPU-Sim and GPUWattch. A thread_exit in GPUWattch caused another library load, which created a deadlock. We solved this by removing the thread_exit.

We took a similar approach to try to get Tensorflow to run in GPGPU-Sim. We managed to get TensorFlow

to call CUDA Runtime API, but unfortunately TensorFlow tries to launch PTX that is not in libcudnn.so. To get around this, we attempted to use TensorFlow's _py-wrap_tensorflow_internal.so's PTX. However, this file it uses syntax that is not supported by GPGPU-Sim to initialize arrays using curly braces ({}). Thus, adding this support is left to future work.

F. Checkpointing support

GPGPU-Sim can be run either in the Performance simulation mode or in the Functional simulation mode. The Functional simulation mode only executes the application and gives functionality correctness of the application, but doesn't give any performance statistics. The Performance simulation mode collects all statistics and gives an estimate of the number of GPU clock cycle on actual hardware. However, the Performance simulation mode is generally 7-8 times slower than the Functional simulation mode. Run-time of architecture simulator is significantly slower than the actual hardware and accordingly a typical cuDNN/Pytorch application might take very long time to run on GPGPU-Sim. For example, MNIST takes ~1.25 hours on GPGPU-Sim's Performance mode to classify three images. This is much longer than a real GPU takes, and the user may only be interested in the performance analysis of a particular part of the program rather than the entire program. Thus, we added checkpoint-resume functions to GPGPU-Sim. The flow of checkpointing is explained in the Figure 4 and the flow of the implementation is explained in the Figure 5.
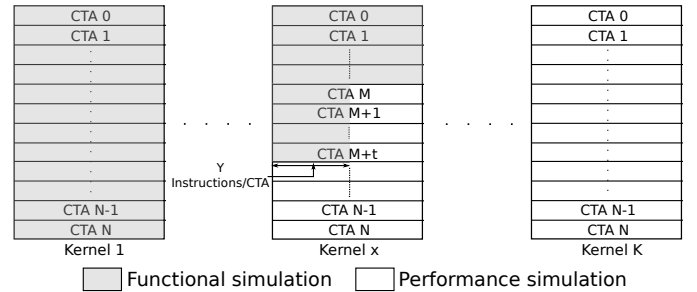

Fig. 4. Checkpointing in GPGPU-Sim: Instead of running entire the application in the Performance mode, the user can run the application in the Functional simulation mode until some point, as shown in Figure 4, and save the necessary data to resume in files. Then the user can resume from this point in Performance simulation mode. Parameters to define checkpoint position such as x, M, t and y, as shown in Figure 4, can be defined by the user in GPGPU-Sim config file.

We support checkpointing at bound kernel boundaries and a CTA boundary within a kernel. At the end of each kernel, we save the computational results to the GPU's global memory. This makes it possible to resume execution from any kernel which has been executed before the checkpoint. For a checkpoint at a specific CTA M within a kernel x, all the kernels with kernel_id $< x$ are executed normally and the state of the GPU's global memory is saved to a file. For kernel x any of the $M - 1$ CTAs before the desired checkpoint point are executed normally. However, for CTAs M to $M + t$, y
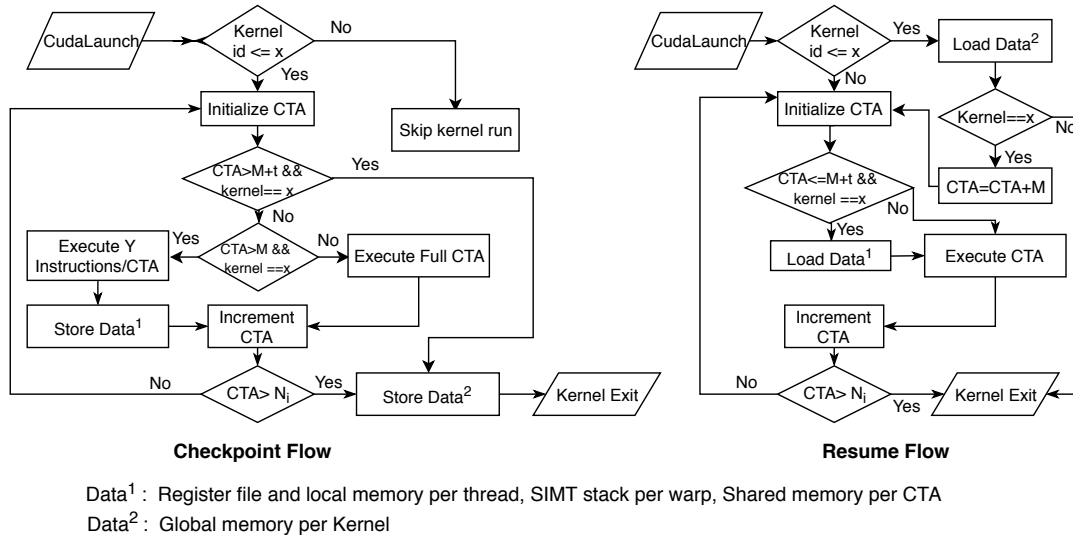
Checkpoint Flow

Resume Flow

Data^1 : Register file and local memory per thread, SIMT stack per warp, Shared memory per CTA

Data^2 : Global memory per Kernel

Fig. 5. Block diagram for Checkpoint and resume flow. Here parameters x, t, M and Y can be configured by the user and explained in figure 4. N_i is the total number of CTA in the specific kernel.

instructions ($y > x$) per CTA are executed. In order to resume from that point, we also need to checkpoint the register data and local memory data for each active thread, the SIMT stack (which is used to handle branch divergence within a warp [3]) for each active warp, and the shared memory for each CTA. All kernels with kernel_id $> x$ are not executed.

To resume at a given checkpoint, all kernels with a kernel_id $< x$ are skipped but the GPU global memory is restored for each kernel since the program might call `cudaMemcpy` between two kernels and perform computation on this data. For kernel x, all CTAs $< M$ are skipped for computation. CTAs M to $M + t$ are initialized and the register data, local memory, SIMT stack, and shared memory are restored for the corresponding threads, warps and CTAs.

IV. CORRELATION

During the process of updating GPGPU-Sim to run machine learning workloads (Section III), we also used a 32-bit floating-point version of MNIST to correlate GPGPU-Sim's execution time with a GeForce GTX 1050. We use MNIST to perform the correlation because it is relatively simple and uses a wide variety of cuDNN layers such as LRN and Winograd. Additionally, MNIST contains self-checking code at the end of the application, which helps ensure the functional correctness of our implementation.

We correlated GPGPU-Sim's performance with real GPUs by comparing the number of GPU cycles with those reported by NVProf. Figures 6 and 7 show the overall correlation for MNIST and the correlation for select kernels, respectively. We selected these kernels because they show the largest discrepancies. For the kernels we do not show, GPGPU-Sim achieves very high correlation with the real GPU. Overall, GPGPU-Sim provides a correlation of 72%. Inspecting the per-kernel results shows that the overall discrepancy is heavily affected by a few kernels such as CGEMM, Winograd, and
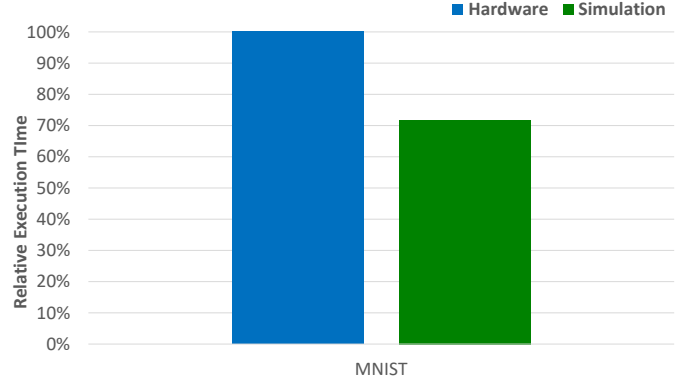

Fig. 6. Correlating execution time for MNIST.

LRN. Thus, improving support for these kernels will make the overall correlation even better.

A. Power Consumption

Figure 8 breaks down MNIST's power consumption into 6 key categories: core, L1 cache, L2 cache, NOC, DRAM, and Idle. As expected for relatively computationally intensive CNNs like MNIST, on average the core (in particular the ALUs) consume 65% of the power. However, on average Idle power consumes a further 25% of the total power. This represents a tuning opportunity for future architectural exploration, which is enabled by this work.

V. CASE STUDIES

A. Methodology

In this section, we study another simple cuDNN program from the NVIDIA examples, conv_sample. We choose conv_sample because it performs forward, backward data, and backward filter convolutions, which are common machine learning operations. Using conv_sample, we iterated over
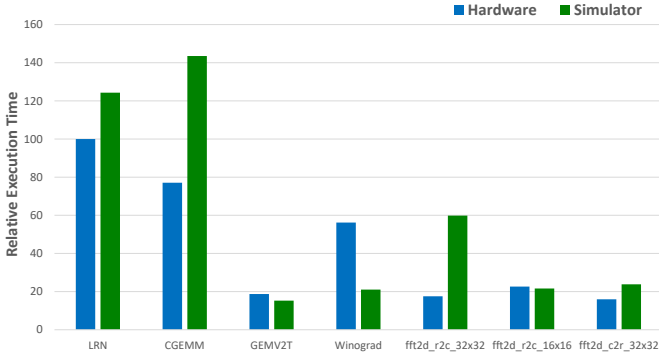
Fig. 7. Select per-kernel, execution time correlation results for MNIST. These results are a subset of the kernels in Figure 6.
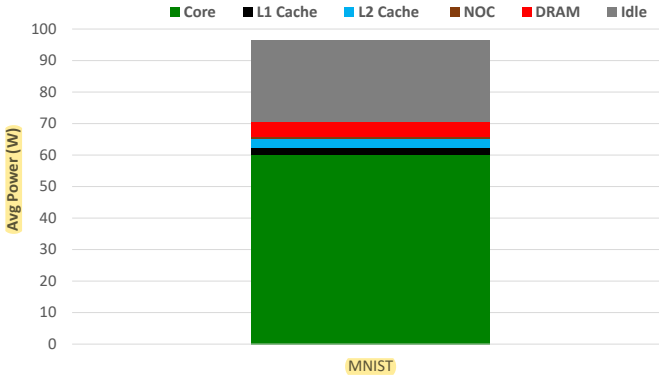

Fig. 8. Average power consumption for a 32-bit floating-point version of MNIST, subdivided into 6 components of the simulated GPU.

Fig. 9. Forward Convolution (FFT) DRAM Efficiency Plot
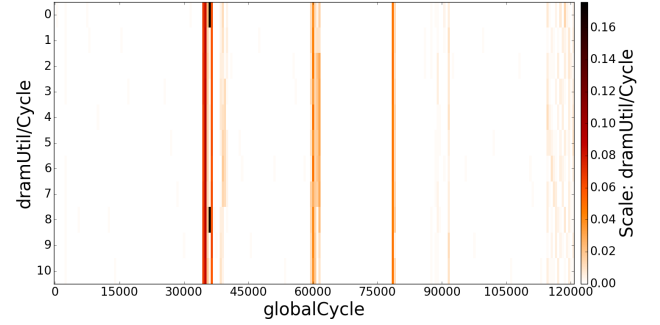
Fig. 10. Forward Convolution (FFT) DRAM Utilization Plot

the various cuDNN algorithms available for each type of convolution. For forward convolution, we ran FFT, FFT Tiling, GEMM, Implicit GEMM, Winograd, and Winograd Nonfused. For backward data convolution, we ran Algorithm 0, Algorithm 1, FFT Tiling, Winograd, and Winograd Nonfused. For backward filter convolution, we ran Algorithm 0, Algorithm 1, Algorithm 3, FFT, FFT Tiling, and Winograd Nonfused. For all of these different approaches, we model a NVIDIA Pascal GeForce GTX1080Ti in GPGPU-Sim.

We studied each algorithm using AerialVision [32], a tool that plots metrics per bank/shader per cycle. In this case study, we plotted DRAM efficiency, global IPC and per shader IPC, and warp divergence – details that are enabled by using simulators like GPGPU-Sim. DRAM efficiency and utilization is DRAM bandwidth utilization when there is a pending request waiting to be processed and two times the number of read and write commands per command cycle [3]. The y-axis for each is the bank number. The global IPC shows the total number of instructions committed per cycle from all the shader streaming multiprocessors, or cores. The shader IPC breaks this down further and shows the number of instructions being committed per shader core. To show how IPC varies across shaders, in the graph the y-axis is the shader core number.
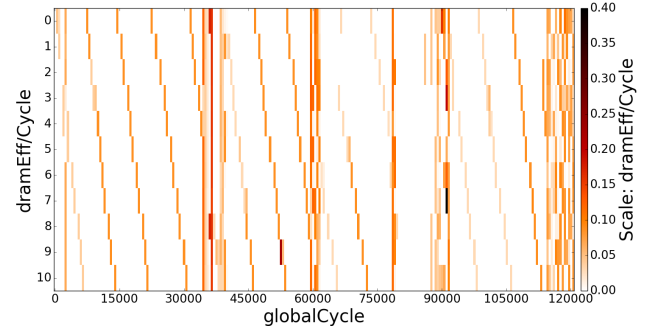
Warp divergence is plotted as a breakdown of the number of warps that are issued for execution. W0 means idle, and W1 through W32 are warps. Visually, the more layers there are, the more warp divergence there is.

B. DRAM Efficiency and Utilization

Figure 9, 11, 12, 13, and 14 shows the DRAM efficiency and utilization, for each convolution approach, as a sequence of DRAM banks. For FFT, we see that most of the DRAM banks show high memory efficiency, interspersed with periods of parallel efficiency. However, FFT also has a mix of serial and parallel efficiency patterns. In the serial sections, FFT is unable able to parallelize memory bank accesses. This phenomenon
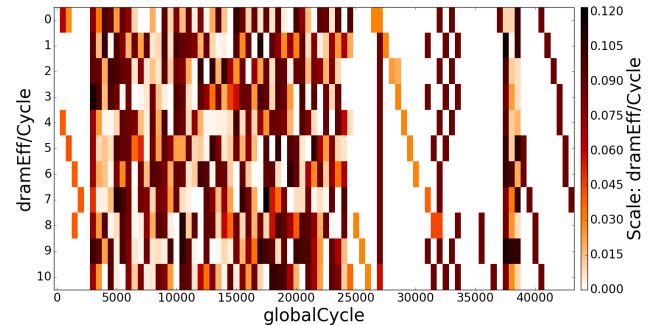

Fig. 11. Forward Convolution (GEMM) DRAM Efficiency Plot
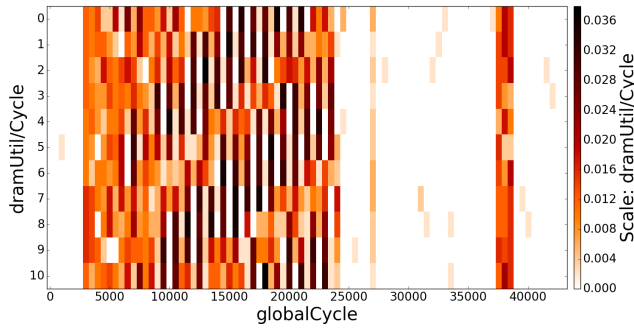
Fig. 12. Forward Convolution (GEMM) DRAM Utilization Plot
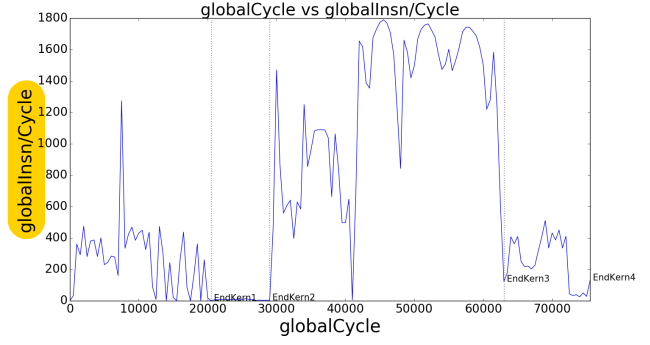

Fig. 15. Forward Convolution (Winograd Nonfused) Global IPC Plot
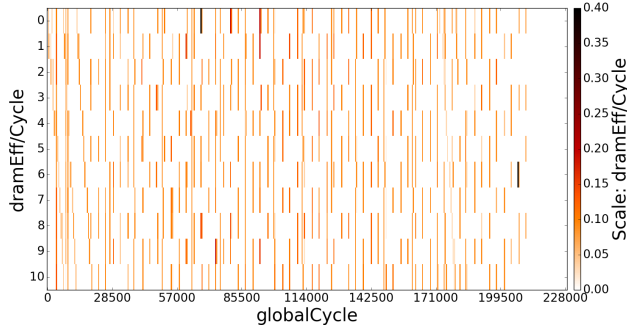

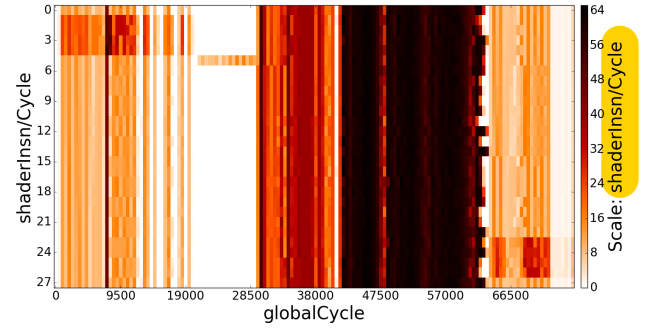Fig. 13. Backward Filter Convolution (Algorithm 0) DRAM Efficiency Plot

Fig. 16. Forward Convolution (Winograd Nonfused) Shader IPC Plot

is known as bank camping. However, bank camping is less of an issue for other approaches like forward convolution with the GEMM algorithm and the backward filter convolution with either algorithm 0 or 1. More generally, some algorithms can make more efficient use of all the banks at the same time, and for all approaches the memory access patterns tend to go through phases of high and low efficiency.

C. Global and Shader IPC

The Winograd Nonfused algorithm has the highest IPCs for all three types of convolution. Furthermore, Figures 15, 16, 18, and 19 show that the forward convolution and backward data convolution implementations are balanced across all the
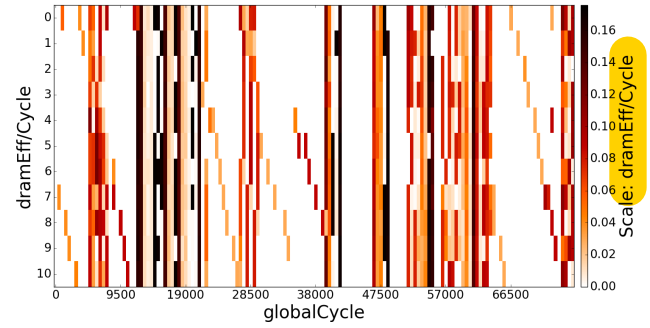

Fig. 17. Forward Convolution (Winograd Nonfused) DRAM Efficency Plot
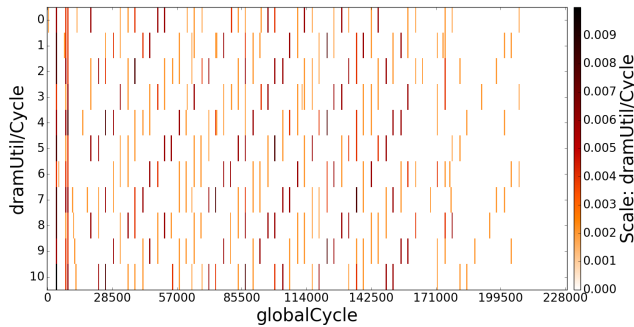
shader cores and thus achieve high per shader IPCs. Although the backward filter convolution version of Winograd Nonfused, shown in Figures 20 and 21, still has the highest IPC, only some of the cores are being used due to load imbalance. However, for the active cores, it commits many instructions per cycle. In general, the algorithms exhibit several clear phases. For example, in Figure 16, only one core is actively committing instructions for a portion of the cycles and at other times, all cores are committing a lot of instructions quickly. Thus, these results show that there are opportunities to save/reduce power by turning off cores during the phases they are not used.
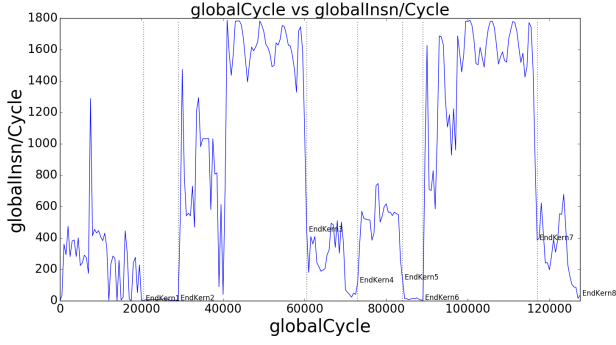

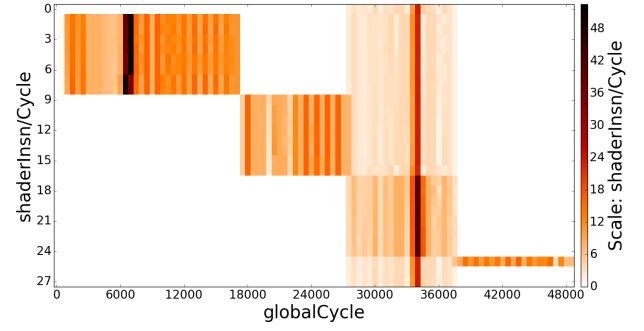Fig. 14. Backward Filter Convolution (Algorithm 0) DRAM Utilization Plot

Fig. 18. Backwards Data Convolution (Winograd Nonfused) Global IPC Plot

Fig. 19. Backwards Data Convolution (Winograd Nonfused) Shader IPC Plot

Fig. 20. Backwards Filter Convolution (Winograd Nonfused) Global IPC Plot
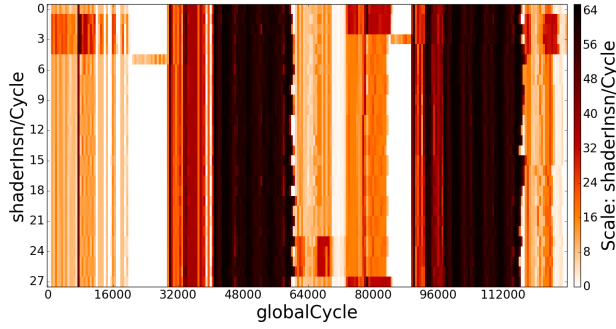

Fig. 21. Backwards Filter Convolution (Winograd Nonfused) Shader IPC Plot
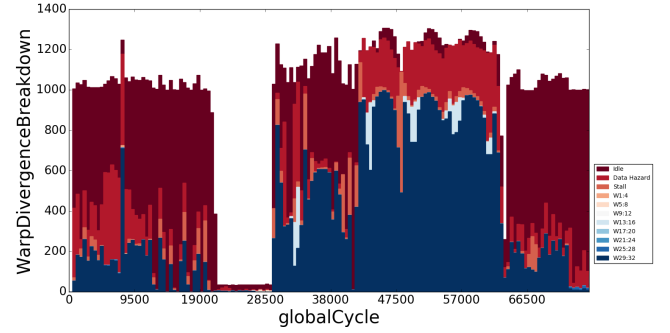

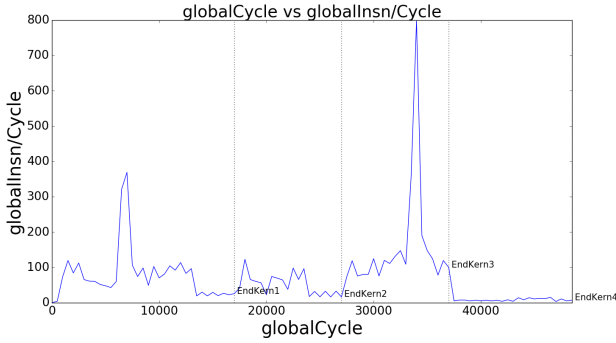Fig. 22. Forward Convolution (Winograd Nonfused) Warp Divergence Plot

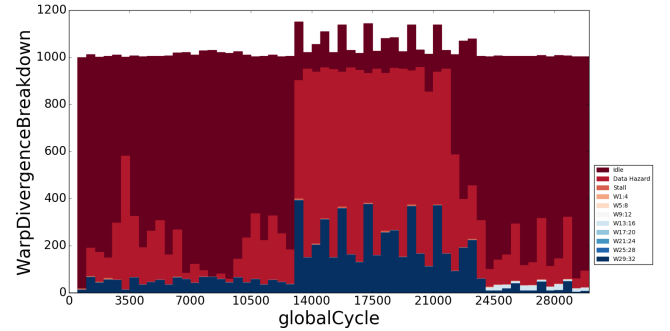
Fig. 23. Forward Convolution (Implicit GEMM) Warp Divergence Plot

D. Comparing DRAM Efficiency to IPC

Figures 16 and 17 show that when Winograd Nonfused's IPC is highest, the memory efficiency is low, indicating that there are phases that the program is compute bound. For example, from around cycle 60000 to 70000, the execution is likely compute bound. This demonstrates how GPGPU-Sim (and AerialVision) can be used to identify regions of interest in applications and how the GPGPU-Sim's statistics can enable detailed comparisions.

E. Warp Divergence

In general, warp divergence is not an issue for any of the algorithms we tested – likely because these algorithms are usually compute bound and have few branches. The forward convolution component of the Winograd Nonfused algorithm has the most significant warp divergence, as shown in Figure 22, where up to two warps are executing at the same time. However, this has a negligible impact on the IPC, since forward convolution with Winograd Nonfused is actually one of the fastest algorithms. Nevertheless, this represents another piece of information that GPGPU-Sim provides and can be used to optimize other machine learning algorithms.

In Figure 23, we see that a majority of the warp breakdown is taken up by data hazards and idle warps. Comparing this to the IPC plots, Figures 24 and 25, we see that the low IPC despite the good load balance in the early part of the kernel can
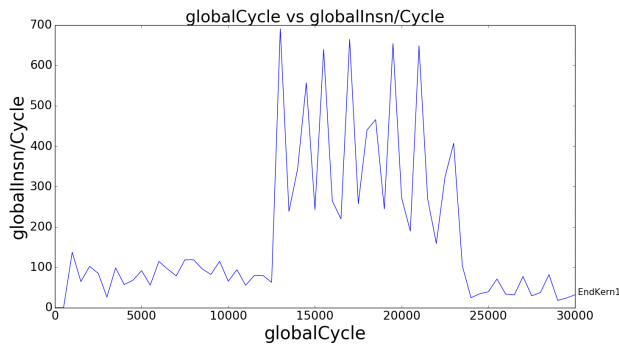
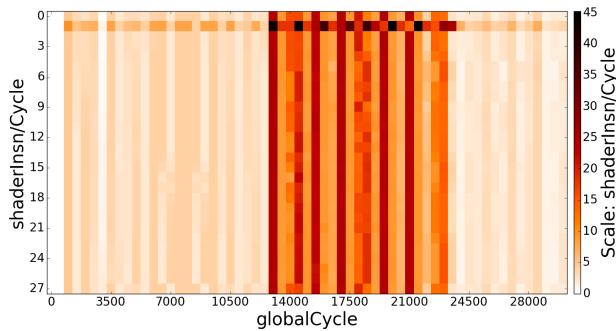Fig. 24. Forward Convolution (Implicit GEMM) Global IPC Plot

Fig. 25. Forward Convolution (Implicit GEMM) Shader IPC Plot

be attributed to this idle warp breakdown. Thus, this represents another opportunity for optimization or power savings.

VI. FUTURE WORK

In addition to getting TensorFlow to run in GPGPU-Sim and adding complete FP16 support, the ability to capture the inputs to kernels requires further effort. For example, our new GPGPU-Sim model can be enhanced to work with double pointer arguments. Resolving these issues will allow us to extract specific kernels, run them individually on hardware, and study them using higher-level tools like NVProf.

VII. CONCLUSION

In this paper, we described the changes we made to the GPGPU-Sim simulator [3], [4] to enable it to run PyTorch by running PTX kernels included in NVIDIA's cuDNN [5] library. We use the resulting modified simulator, which we plan to make available publicly with this paper, to study deep learning workloads and analyze their behavior. With our changes to GPGPU-Sim's Functional simulation model, we find that GPGPU-Sim's Performance model running a cuDNN enabled implementation of LeNet for MNIST reports results within 30% of real hardware. Using GPGPU-Sim's AerialVision performance analysis tool we observe that cuDNN API calls contain many varying phases and appear to include potentially inefficient microarchitecture behavior such as DRAM partition bank camping, at least when executed on GPGPU-Sim's current performance model. Since most deep neural networks deployed today are trained using GPUs via high-level frameworks such as TensorFlow [1] and PyTorch [2], this work has the potential to enable significant microarchitectural research into current deep neural networks.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI 16, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," 2017.

[3] T. M. Aamodt, W. W. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, A. Bakhoda *et al.*, "Gpgpu-sim 3.x manual," 2012.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS. IEEE, 2009, pp. 163–174.

[5] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, vol. abs/1410.0759, 2014.

[6] "Nvidia tesla v100 gpu architecture," NVIDIA Corp., Tech. Rep., August 2017.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[9] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[11] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[12] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[13] N. Compute, "Ptx: Parallel thread execution isa version 2.3," *Dostopno na: http://developer. download. nvidia. com/compute/cuda*, vol. 3, 2010.

[14] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[15] J. D. Andrew Kerr, Duane Merrill and J. Tran, "Cutlass: Fast linear algebra in cuda c++," 2017.

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks." in *CVPR*, vol. 1, no. 2, 2017, p. 3.

[22] N. Fujimoto, "Faster matrix-vector multiplication on geforce 8800gtx," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[23] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 451–460, 2010.

[24] V. Mnih, "Cudamat: a cuda-based matrix class for python," *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, vol. 4, 2009.

[25] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 317–324.

[26] J. Li, S. Ranka, and S. Sahni, "Strassen's matrix multiplication on gpus," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, 2011, pp. 157–164.

[27] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[28] NVIDIA, "Profiler's user guide," https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2018.

[29] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 209–218.

[30] S. A. Mojumder, M. S. Louis, Y. Sun, A. K. Ziabari, J. L. Abellan, J. Kim, D. Kaeli, and A. Joshi, "Profiling DNN Workloads on a Volta-based DGX-1 System," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2018.

[31] H. Zhu, A. Phanishayee, G. Pekhimenko, B. Schroeder, B. Zheng, M. Akrout, A. Pelegris, and A. Jayarajan, "Benchmarking and Analyzing Deep Neural Network Training," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2018.

[32] A. Ariel, W. W. Fung, A. E. Turner, and T. M. Aamodt, "Visualizing complex dynamics in many-core accelerator architectures," in *IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS. IEEE, 2010, pp. 164–174.

[33] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 78–91.

[34] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[35] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[36] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic Deep Neural Network Computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. New York, NY, USA: ACM, 2017, pp. 382–394. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123982

[37] M. Mathieu, M. Henaff, and Y. Lecun, "Fast training of convolutional networks through ffts," in *International Conference on Learning Representations (ICLR2014)*, April 2014.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.